高度なFaceletsプログラミング

カスタム・ロジック・タグとEL機能を作成する

Richard Hightower (rhightower@arc-mind.com)
Developer
ArcMind Inc.

2006年 5月 09日

もし皆さんが、国際化は難しいと考えているのであれば、ぜひ考え直してください。この記事は、好評を博した「FaceletsはぴったりとJSFにフィットします」のフォローアップです。今回は著者のRichard Hightowerが、JSF（Java Server Faces）とELとの隙間を埋める橋渡しとして、より高度な方法を紹介します。Webページを『容易に』国際化し、構成コンポーネントにカスタムのロジック・タグを追加し、そしてFacelets開発にメタプログラミングを取り入れる方法を学びましょう。

Faceletsは、JSF開発コミュニティーにとって、正にかゆいところをかいてくれるもののようです。私がFaceletsに関して書いた最初の記事に対して、非常に多くの肯定的な反響がありました。この記事では、Faceletsを使うとさらに面白いことができることをお見せしようと思います。Jacob Hookomや、Faceletsの開発に関わるすべての人に称賛を！

前回の記事では、Faceletsの背景にある概念を紹介し、またHTMLスタイルのテンプレートや再利用可能構成コンポーネントを作成、操作する方法を説明しました。この記事では、前回の記事で導入したものと同じ例やコンポーネントを使いながら、前回の議論の上に話を進めます。まず、FaceletsのEL（expression language）機能を使って苦労なく国際化を行う方法を説明します。次に、合理的なデフォルトの作り方とカスタム・タグの作り方を説明します。そして最後に、Faceletsで軽量メタプログラミングを行う方法について説明します。

この記事で使用している例の多くは前回の記事を基にしているため、まず皆さんには前回の記事を読むようにお勧めします。また、先に進む前に、この記事で使用するサンプル・コードをダウンロードし、Facelets（とTomahawk）をインストールした方がよいでしょう。

国際化が容易に

Faceletsを扱うのであれば、国際化は障害になりません。最初の練習として、前回の記事で使用したフィールド構成コンポーネントを拡張する方法を説明します。この拡張によって、テキストとして表示すべき日付やブール表現、その他あらゆる種類のJava型を扱えるようにします。また、フィールドに対するラベルを国際化するために、合理的なデフォルトを使う方法についても説明します。

前回の記事を思い出して欲しいのですが、フィールド構成コンポーネントは、（もしラベルが渡されない場合には）デフォルトのラベルとしてフィールド名を使います。これを下記に示します。

```
...
<!--  The label is optional. Use fieldName as label if label missing. -->
    <c:if test="${empty label}">
        <c:set var="label" value="${fieldName}" />
    </c:if>
```

これを変更したい、という場合を考えてみましょう。つまりデフォルト・ラベルとしてfieldNameをロードする代わりに、Facesと関連付けられた、リソース・バンドルにあるfieldNameをフィールド構成コンポーネントが見つけるようにしたい、とします。そのためには、タグとEL機能を定義するタグ・ライブラリーを作成します。まず、リソース・バンドルにあるfieldNameを見つけるEL機能を作成します。もしEL機能がリソース・バンドルの中でラベルを見つけられない場合には、与えられたfieldNameのラクダ記法（camel case）に基づいてfieldNameを生成しようとします。リスト1は、EL機能の使い方の一例を示しています。

リスト1. EL機能の使い方の例

```
<html xmlns="http://www.w3.org/1999/xhtml"
      ...
      xmlns:arc="http://www.arc-mind.com/jsf/core"
      xmlns:t="http://myfaces.apache.org/tomahawk">


<!--  The label is optional. Generate it if it is missing. -->
    <c:if test="${empty label}">
        <c:set var="label"
               value="${arc:getFieldLabel(fieldName,namespace)}" />
    </c:if>
```

EL機能を作成する

リスト1では、Facesと関連付けられた、リソース・バンドルにあるfieldNameを探すためにarc:getFieldLabel(fieldName,namespace) というEL機能を使っています。getFieldLabel() メソッドは、リソース・バンドルの中にあるラベルを実際に探します。EL機能を作成するための手順は次の通りです。

1. TagLibraryクラスを作成する
2. Javaユーティリティー・クラスを作成する
3. getFieldLabel() メソッドを登録する
4. TagLibraryクラスを登録する
5. facelets-taglib記述子ファイルを登録する

以下のセクションでは、これらのステップを1つ1つ説明し、また新しいEL機能の使い方も説明します。

ステップ1. TagLibraryクラスを作成する

EL機能とロジック・タグは、Facelets TagLibraryクラスの中で定義されます。TagLibraryクラスを作成するためには、AbstractTagLibraryをサブクラス化する必要があります。これをリスト2に示します。

リスト2. AbstractTagLibraryをサブクラス化する

```
package com.arcmind.jsfquickstart.tags;

import com.sun.facelets.tag.AbstractTagLibrary;

import java.lang.reflect.Method;
import java.lang.reflect.Modifier;


/**
 * JsfCoreLibrary is an example for IBM developerWorks (c).
 * @author Rick Hightower from ArcMind Inc. http://www.arc-mind.com
 */
public final class JsfCoreLibrary extends AbstractTagLibrary {
    /** Namespace used to import this library in Facelets pages  */
    public static final String NAMESPACE = "http://www.arc-mind.com/jsf/core";

    /**  Current instance of library. */
    public static final JsfCoreLibrary INSTANCE = new JsfCoreLibrary();
    ...
    ...
```

タグ・ライブラリーが定義できると、それに対してEL機能とロジック・タグを追加することができます。

ステップ2. Javaユーティリティー・クラスを作成する

次に、getFieldLabel() という静的メソッドを持つJavaユーティリティー・クラスを作成する必要があります。このユーティリティー・メソッドはFaceletsと何ら特別な関係は持っておらず、単なる通常の静的なJavaメソッドにすぎません。これをリスト3に示します。

リスト3. getFieldLabel() を持つJavaユーティリティー・クラス

```
package com.arcmind.jsfquickstart.tags;

import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;

import javax.faces.context.FacesContext;

/**
 * Functions to aid developing JSF applications.
 */
public final class JsfFunctions {
 /**
  * Stops creation of a new JsfFunctions object.
  */
 private JsfFunctions() {
 }

 /**
  * Get the field label.
  *
  * @param fieldName
  *            fieldName
  * @param formId
  *            form id
  * @return Message from the Message Source.
  */
 public static String getFieldLabel(final String fieldName,
   final String formId) {
```

```
 Locale locale = FacesContext.getCurrentInstance().getViewRoot()
   .getLocale();
 String bundleName = FacesContext.getCurrentInstance().getApplication()
   .getMessageBundle();

 ResourceBundle bundle = ResourceBundle
                                      .getBundle(bundleName, locale, getClassLoader());

 /** Look for formId.fieldName, e.g., EmployeeForm.firstName. */

 String label = null;
 try {
  label = bundle.getString(formId + fieldName);
  return label;
 } catch (MissingResourceException e) {
  // do nothing on purpose.
 }

 try {
  /** Look for just fieldName, e.g., firstName. */
  label = bundle.getString(fieldName);
 } catch (MissingResourceException e) {
  /**
   * Convert fieldName, e.g., firstName automatically becomes First
   * Name.
   */
  label = generateLabelValue(fieldName);
 }

 return label;

}

private static ClassLoader getClassLoader() {
 ClassLoader classLoader = Thread.currentThread()
   .getContextClassLoader();
 if (classLoader == null) {
  return JsfFunctions.class.getClassLoader();
 }
 return classLoader;
}

/**
 * Generate the field. Transforms firstName into First Name. This allows
 * reasonable defaults for labels.
 *
 * @param fieldName
 *            fieldName
 *
 * @return generated label name.
 */
public static String generateLabelValue(final String fieldName) {
 StringBuffer buffer = new StringBuffer(fieldName.length() * 2);
 char[] chars = fieldName.toCharArray();

 /* Change firstName to First Name. */
 for (int index = 0; index < chars.length; index++) {
  char cchar = chars[index];

  /* Make the first character uppercase. */
  if (index == 0) {
   cchar = Character.toUpperCase(cchar);
   buffer.append(cchar);

   continue;
  }
```

```
   /* Look for an uppercase character, if found add a space. */
   if (Character.isUpperCase(cchar)) {
    buffer.append(' ');
    buffer.append(cchar);

    continue;
   }

   buffer.append(cchar);
  }

  return buffer.toString();
 }
}
```

リスト3に示すgetFieldLabel() メソッドは、JSFに関連付けられた、リソース・バンドルの中のフィールドを探します。もしそのフィールドが見つからない場合には、ラクダ記法のストリングを分割して先頭を大文字にします。つまり「firstName」というフィールド名は、「First Name」のようになります。これによって合理的なデフォルトが得られ、しかも、もしこれをオーバーライドしたい場合には、リソース・バンドルの中に別のデフォルトを設定することもできます。

ステップ3. getFieldLabel() メソッドを登録する

リスト2に示すAbstractTagLibraryクラス（JsfCoreLibrary）の中に、ベース・クラスを持つgetFieldLabel() メソッドを登録し、EL機能として使用できるようにする必要があります。これをリスト4に示します。

リスト4. EL機能としてJsfCoreLibrary.javaを登録する

```
    public JsfCoreLibrary() {
        super(NAMESPACE);

        try {
            Method[] methods = JsfFunctions.class.getMethods();

            for (int i = 0; i < methods.length; i++) {
                if (Modifier.isStatic(methods[i].getModifiers())) {
                    this.addFunction(methods[i].getName(), methods[i]);
                }
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

ご覧の通り、リスト4は単にリフレクション（reflection）を使って、JsfFunctionクラスの全メソッドに対して繰り返しを行っています。そして、addFunction() という継承されたメソッドを使って、JsfFunctionの全静的メソッドを追加しています。

ステップ4. TagLibraryクラスを登録する

これで、EL機能と、それを含むTagLibraryクラスが定義できました。次に、新しいTagLibraryクラスを登録し、Faceletsがそれを見つけられるようにする必要があります。そのためには、jsf-core.taglib.xmlという新しいfacelets-taglibファイルの中で、新しいライブラリー・クラスを宣言します。これをリスト5に示します。

リスト5. jsf-core.taglib.xml

```
<?xml version="1.0"?>

<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "facelet-taglib_1_0.dtd">

<facelet-taglib>
 <library-class>
           com.arcmind.jsfquickstart.tags.JsfCoreLibrary
      </library-class>
</facelet-taglib>
```

ステップ5. facelets-taglib記述子ファイルを登録する

Faceletsがタグ・ライブラリーを見つけるようにするためには、そのタグ・ライブラリーをWEB-INF/web.xmlファイルに登録する必要があります。これをリスト6に示します。

リスト6. 新しいタグ・ライブラリーをFaceletsに登録する

```
 <context-param>
  <param-name>facelets.LIBRARIES</param-name>
  <param-value>
   /WEB-INF/facelets/tags/arcmind.taglib.xml;
   /WEB-INF/facelets/tags/jsf-core.taglib.xml
  </param-value>
 </context-param>
```

ここで、init-param facelets.LIBRARIESに渡されるセミコロン区切りのリストにjsf-core.taglib.xmlを追加していることに注意してください。構成コンポーネントは、それぞれ独自のtaglibファイルの中に入ります。またEL機能とFaceletsロジック・タグは、同じtaglibファイルの中に入ります。

構成コンポーネントの中のEL機能を使う

EL機能を定義し、それをFacelets taglib記述子ファイルの中に置き、またtaglib記述子ファイルをweb.xmlに登録したので、このタグを使い始めることができます。フィールド構成コンポーネントの中でarc:getFieldLabelを使うと、リソース・バンドルの中にあるラベルを見つけることができます。これをリスト7に示します。

リスト7. EL機能を使う

```
<!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:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:arc="http://www.arc-mind.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:t="http://myfaces.apache.org/tomahawk">

  THIS TEXT WILL BE REMOVED
  <ui:composition>

    <c:if test="${empty label}">
        <c:set var="label"
               value="${arc:getFieldLabel(fieldName,namespace)}" />
    </c:if>
       ...
```

ここで、この「arc」ライブラリーの名前空間をxmlns:arc="http://www.arc-mind.com/jsf/core"というHTML要素の中で規定することによって「arc」ライブラリーをインポートしていることに注意してください。この名前空間は、TagLibraryクラスの中で宣言されている名前空間（この場合はJsfCoreLibrary）と一致する必要があります。これを下記に示します。

```
public final class JsfCoreLibrary extends AbstractTagLibrary {
    /** Namespace used to import this library in Facelets pages  */
    public static final String NAMESPACE = "http://www.arc-mind.com/jsf/core";
```

ここまでのステップは5つしかありませんが、かなりの大仕事に思えるかも知れません。しかし、一旦基本的なステップを行ってしまえば、同じような苦労を毎回繰り返すことなくロジック・タグやEL機能を追加できるようになります。すべてのフィールド・ラベル名には、合理的なデフォルトが付けられます。アプリケーションが、皆さんの国や母国語しかサポートしていないのであれば、通常はリソース・バンドルにエントリーを追加する必要はありません。そして特別なラベルが必要な場合、あるいはそのアプリケーションを別の言語や場所で使用したい場合には、単にリソース・バンドルにエントリーを追加するだけでよいのです。

Faceletsは、私が面倒と思っていた国際化の問題、つまり必要無さそうなことのために特別なコードを大量に書く必要がある、という問題を解決してくれます。今やそうしたことが、無料になるのです。では次に、ブール表現や日付、テキスト・コンポーネントを自動的に生成するようにフィールド・タグを変更したい場合にはどうするのかを見て行きましょう。

カスタムのロジック・タグを作成する

まず、field.jspタグにメタプログラミング機能を追加します。そのためには、ある特定な値バインディングの型がブール型（isBoolean）なのか、何らかのテキスト形式（isText）なのか、あるいは日付（isDate）なのかを伝えるFaceletsロジック・タグを作成します。次に、このタグを使って適当なコンポーネントを描画します。

リスト8は、こうしたタグ（field.xhtml）の使い方を示しています。

リスト8. field.xhtml

```
<!--  Initialize the value binding -->
<arc:setValueBinding var="vb" valueBinding="#{entity[fieldName]}"/>

<!--  If the value binding is a string, display an inputText field. -->
<arc:isText id="vb">
    <h:inputText id="#{fieldId}"  value="#{entity[fieldName]}"
                 required="${required}" styleClass="fieldInput">
          <ui:insert />
    </h:inputText>
</arc:isText>

<!--  If the value binding is a boolean, display a
      selectBooleanCheckbox field. -->
<arc:isBoolean id="vb">
         <h:selectBooleanCheckbox id="#{fieldId}"
                  value="#{entity[fieldName]}" required="${required}"/>
</arc:isBoolean>

<!--  If the value binding is a date, display a t:inputDate field. -->
<arc:isDate id="vb">
 <t:calendar id="#{fieldId}" renderPopupButtonAsImage="true"
           renderAsPopup="true"
                          value="#{entity[fieldName]}"
                          required="${required}"
           styleClass="fieldInput" >
                <ui:insert />
      </t:calendar>

</arc:isDate>

...
```

リスト8のfield.xhtmlコードは、もし値バインディングが何らかの形式のテキストであれば、inputTextを描画します。もし値バインディングがブール型の場合には、selectBooleanCheckboxを描画します。そして最後に、もし値バインディングが日付の場合には、Tomahawkカレンダー・コンポーネントを描画します（囲み記事、「Tomahawkについて」を見てください）。

Tomahawkについて

ここで挙げた例を完成するために、私はアプリケーションの中にTomahawkサポートをインストールしました。TomahawkはオープンソースのJSFコンポーネント・プロジェクトであり、MyFacesの一部です。またMyFacesはApacheの一部です。このインストールのためには、適当なJARファイルをダウンロードしてTomahawk Facelets taglibを登録する必要があり、またweb.xmlにJavaScript生成サポートをインストールする必要もあります。MyFaces Tomahawkのダウンロードとインストールに関しては、参考文献を見てください。

次に、ある特定なコンポーネントにバインドされている型を取得するために、値バインディングを作成します。値バインディングは、コンポーネントがバインドされている値の型に関する情報を持っています。JSFを使ったことのある人であれば、値バインディングには慣れているはずです。例えば#{Employee.firstName} は恐らくストリング型と等価な値バインディング・タグですが、#{Employee.age} タグは恐らく整数型と等価です。次のセクションでは、Faceletsを使って値バインディング・タグを作成する方法を学びます。

値バインディング・タグを作成する

Faceletsの中の値バインディングを検索するタグを作成するためのステップは、次の4つです。

1. TagHandlerをサブクラス化するクラスを作成する
2. コンストラクターに属性を登録する
3. applyメソッドをオーバーライドする
4. JsfCoreLibraryクラスの中にタグを登録する

以下のセクションでは、値バインディング・タグを構築するための各ステップを説明し、その後でタグの使い方を説明します。

ステップ1. TagHandlerをサブクラス化するクラスを作成する

Facelets TagHandlerを使って、タグ・ボディーにあるコンポーネントをコンポーネント・ツリーに追加するかどうかを判断するロジックを注入します。Faceletsテンプレートの最終目的は、JSFコンポーネント・ツリーを作成することであることを忘れないでください。

リスト9は、SetValueBindingHandlerタグがTagHandlerをサブクラス化する様子を示しています。SetValueBindingHandlerは単に変数を定義し、その変数をロジック・タグが再利用できるように、その変数をFaceletsスコープに置きます。

リスト9. SetValueBindingHandlerタグがTagHandlerをサブクラス化する

```
package com.arcmind.jsfquickstart.tags;

import com.sun.facelets.FaceletContext;
import com.sun.facelets.tag.TagAttribute;
import com.sun.facelets.tag.TagConfig;
import com.sun.facelets.tag.TagHandler;


import javax.el.ValueExpression;
import javax.faces.component.UIComponent;


/**
 * Maps value binding and type so other tags can access it.
 */
public final class SetValueBindingHandler extends TagHandler {
```

ステップ2. コンストラクターに属性を登録する

Faceletsタグを書く際には、タグそのものによって属性が定義されるのであり、XMLファイルの中で定義されるのではないことを理解することが重要です。SetValueBindingHandlerのコンストラクターは、コンストラクターの中に2つの属性（varとvalueBinding）を定義し、登録します。これをリスト10に示します。

リスト10. コンストラクターの中に属性を登録する

```
/**
 * Maps value binding and type so other tags can access it.
 */
public final class SetValueBindingHandler extends TagHandler {
    /**   The name of the new variable that this tag defines. */
    private final TagAttribute var;

    /**   The actual value binding expression. */
    private final TagAttribute valueBinding;
```

```
    /**
     * Constructor. Set up the attributes for this tag.
     *
     * @param config TagConfig
     */
    public SetValueBindingHandler(final TagConfig config) {
        super(config);
        /* Define var and valueBinding attributes. */
        this.var = this.getRequiredAttribute("var");
        this.valueBinding = this.getRequiredAttribute("valueBinding");
    }
```

ステップ3. applyメソッドをオーバーライドする

次に、SetValueBindingHandlerの中のapplyメソッドをオーバーライドします。通常は、タグ・ボディーの中で定義されたコンポーネントをコンポーネント・ツリーに追加するかどうかをプログラム的に判断するためにapplyメソッドをオーバーライドします。しかしSetValueBindingHandlerは、渡された値バインディングに対して、単純に、ある1つの変数（varを設定するための任意の変数、例えば「vb」など）を定義します。リスト11は、applyメソッドをオーバーライドするためのコードを示しています。

リスト11. applyメソッドをオーバーライドする

```
/**
 * Maps value binding and type so other tags can access it.
 */
public final class SetValueBindingHandler extends TagHandler {
...
...

    /**
     * Apply.
     *
     * @param faceletsContext faceletsContext
     * @param parent parent
     *
     * @throws IOException IOException
     */
    public void apply(final FaceletContext faceletsContext, final UIComponent parent) {
     /* Create the ValueExpression from the valueBinding attribute. */
        ValueExpression valueExpression =
            this.valueBinding.getValueExpression(faceletsContext, Object.class);

        /* Get the name of the new value. */
        String tvar = this.var.getValue(faceletsContext);
        Class type = valueExpression.getType(faceletsContext);

        /* Put the value binding into the FaceletsContext where
         * we can retrieve it from other components.
         */
        faceletsContext.setAttribute(tvar, valueExpression);

        /* Cache the type so we don't have to look it
         * up in each tag. */
        faceletsContext.setAttribute(tvar + "Type", type);
    }
```

リスト11は、2つの変数、varとvar + "Type"、を定義します。次に、これらの変数を単純にfaceletsContextの中に置きます（詳細については、リスト11に書き入れた私のコメントを見てください）。

ステップ4. JsfCoreLibraryクラスの中にタグを登録する

この例に対しても、前の例の場合と同じようにJsfCoreLibraryを使うことができます。TagLibraryはEL機能とロジック・タグの両方を含むことができます。そこで、この練習の最後のステップでは、addTagHandlerというスーパークラスを呼んで、JsfCoreLibrary にSetValueBindingHandlerタグを登録します。これをリスト12に示します。

リスト12. JsfCoreLibraryにタグを登録する

```
/**
 * JsfCoreLibrary is an example for IBM developerWorks (c).
 * @author Rick Hightower from ArcMind Inc. http://www.arc-mind.com
 */
public final class JsfCoreLibrary extends AbstractTagLibrary {
    /** Namespace used to import this library in Facelets pages  */
    public static final String NAMESPACE = "http://www.arc-mind.com/jsf/core";

    /**  Current instance of library. */
    public static final JsfCoreLibrary INSTANCE = new JsfCoreLibrary();

    /**
     * Creates a new JsfCoreLibrary object.
     *
     */
    public JsfCoreLibrary() {
        super(NAMESPACE);

        this.addTagHandler("setValueBinding", SetValueBindingHandler.class);
        ...
    }
```

Faceletsの良いところは、同じタグをJSPで書く場合と比較すると、Faceletsでタグを開発した方がずっと少ないXMLで済むことです。さらに多くのタグを開発し、JsfCoreLibrayに登録することも容易にできます。このタグの名前がsetValueBindingであることに注意してください。

カスタムのロジック・タグを使う

これでsetValueBindingというタグが定義できたので、このタグを使い始めることができます。これをリスト13に示します。

リスト13. setValueBindingタグを使う

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:arc="http://www.arc-mind.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core"
      xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:t="http://myfaces.apache.org/tomahawk">

  THIS TEXT WILL BE REMOVED
  <ui:composition>
  ...
    <!--  Initialize the value binding -->
    <arc:setValueBinding var="vb"
                         valueBinding="#{entity[fieldName]}" />
```

web.xmlの中に何かをインポートしたり、何かを設定したりする必要はありません。いったん（例えば「arc」のような）タグ・ライブラリーを定義すれば、XMLファイルを更新したりする

ことなく、そのタグ・ライブラリーにタグや機能を追加することができます。つまりXMLの地獄（XMHell）を回避できるのです！

Faceletsによるメタプログラミング

Faceletsに関するお楽しみの（この記事での）最後として、ちょっとしたタグを作成しましょう。このタグを使うと、もし値バインディングが特定な型の場合には、ボディーの中で定義されるコンポーネントを追加できるのです。

この魔法のタグ、IsTypeHandler() は、あるコンポーネントをコンポーネント・ツリーに追加すべきかどうかを、選択的に判断します。ボディーの中で定義されるコンポーネントをコンポーネント・ツリーに追加すべきである場合には、このタグはthis.nextHandler.apply(faceletsContext, aParent) を呼びます。またisTypeHandler() は、this.nextHandler.apply(faceletsContext, aParent) を呼んでタグ・ハンドラーのボディーの中で定義されるコンポーネントをコンポーネント・ツリーに追加すべきかどうかを判断するために、isType() という抽象メソッドを呼びます。リスト14は、このすべてに関するベース・クラスを示しています。

リスト14. 型を処理するためのベース・クラス

```
package com.arcmind.jsfquickstart.tags;

import com.sun.facelets.FaceletContext;
import com.sun.facelets.tag.TagAttribute;
import com.sun.facelets.tag.TagConfig;
import com.sun.facelets.tag.TagHandler;

import java.io.IOException;

import javax.faces.component.UIComponent;


/**
 * Is the current field a boolean.
 */
public abstract class IsTypeHandler extends TagHandler {
    /**   */
    private final TagAttribute id;

    /**
     * Create tag.
     *
     * @param config TagConfig
     */
    public IsTypeHandler(final TagConfig config) {
        super(config);
        this.id = this.getRequiredAttribute("id");
    }

    /**
     * Is the current field a boolean.
     *
     * @param faceletsContext ctx
     * @param aParent parent
     *
     * @throws IOException IOException
     */
    public void apply(final FaceletContext faceletsContext, final UIComponent aParent)
        throws IOException {
        /* Get the name of the value binding. */
        String tid = this.id.getValue(faceletsContext);
        Class type =
```

```
          (Class) faceletsContext.getVariableMapper().resolveVariable(tid + "Type")
                                                 .getValue(faceletsContext);

        /* If the type is a boolean, process the body of the tag.
         */
        if (isType(type)) {
            this.nextHandler.apply(faceletsContext, aParent);
        }
    }

    /**
     *
     *
     * @param type type
     *
     * @return true if this is the correct type.
     */
    protected abstract boolean isType(Class type);
}
```

次に、処理したい型それぞれに対してタグを追加します。これをリスト15に示します。

リスト15. 特定な型を処理する

```
package com.arcmind.jsfquickstart.tags;

import com.sun.facelets.tag.TagConfig;


/**
 * Is the current field a boolean.
 */
public final class IsBooleanHandler extends IsTypeHandler {

    /**
     * Create tag.
     *
     * @param config TagConfig
     */
    public IsBooleanHandler(final TagConfig config) {
        super(config);
    }

    /**
     *
     *
     * @param type type
     *
     * @return true if this is a boolean.
     */
    protected boolean isType(final Class type) {
        /* If the type is a boolean, process the body of the tag.
         */
        if (type == Boolean.class) {
            return true;
        } else if (type.isPrimitive() && "boolean".equals(type.getName())) {
            return true;
        }

        return false;
    }
}

package com.arcmind.jsfquickstart.tags;

import java.util.Date;
```

```
import com.sun.facelets.tag.TagConfig;

/**
 * Is the current field a date.
 */
public final class IsDateHandler extends IsTypeHandler {

    /**
     * Create tag.
     *
     * @param config TagConfig
     */
    public IsDateHandler(final TagConfig config) {
        super(config);
    }

    /**
     *
     *
     * @param type type
     *
     * @return true if this is a boolean.
     */
    protected boolean isType(final Class type) {
        /* If the type is a string, process the body of the tag.
         */
        if (type == Date.class) {
            return true;
        }
        return false;
    }
}

package com.arcmind.jsfquickstart.tags;

import java.math.BigDecimal;
import java.math.BigInteger;

import com.sun.facelets.tag.TagConfig;

/**
 * Is the current field a string.
 */
public final class IsTextHandler extends IsTypeHandler {

    /**
     * Create tag.
     *
     * @param config TagConfig
     */
    public IsTextHandler(final TagConfig config) {
        super(config);
    }

    /**
     *
     *
     * @param type type
     *
     * @return true if this is a boolean.
     */
    protected boolean isType(final Class type) {
        /* If the type is a string, process the body of the tag.
         */
        if (type == String.class
          || type == Integer.class
```

```
          || type == BigDecimal.class
          || type == BigInteger.class
          || type == Character.class
          || type == Long.class
          || type == Short.class
          || type == Byte.class
          || type == Float.class
          || type == Double.class
          || (type.isPrimitive() && !type.getName().equals("boolean"))
          ) {
            return true;
        } else {
         return false;
        }
    }
}
```

リスト15では、タグ・ボディーの中で定義されているコンポーネントをコンポーネント・ツリーに追加するかどうかを判断するために、単純に値バインディングの型を使っています。各タグはisType() メソッドをオーバーライドします。ここではボディー・コンポーネントをツリーに追加するかどうかの判断に単純に型を使っていますが、場合によっては注釈や他の形式のメタデータを使うことも簡単にできます。

当然のことですが、上記のタグをすべてJsfCoreLibraryに『登録』しないと仕事は終わりません。これをリスト16に示します。

リスト16. 型を処理するタグをTagLibraryに追加する

```
public final class JsfCoreLibrary extends AbstractTagLibrary {
   ...
    public JsfCoreLibrary() {
        ...

        this.addTagHandler("isBoolean", IsBooleanHandler.class);
        this.addTagHandler("isText", IsTextHandler.class);
        this.addTagHandler("isDate", IsDateHandler.class);
        ...
```

型に基づくタグを使う

これでタグ・ライブラリーの中に、さらに3つのタグができました。つまりisBooleanとisText、そしてisDate() の3つです。フィールド・タグの中で、これらのタグを使うことができます。これをリスト17に示します。

リスト17. 最終的なfield.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:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:arc="http://www.arc-mind.com/jsf/core"
      xmlns:c="http://java.sun.com/jstl/core"
 xmlns:fn="http://java.sun.com/jsp/jstl/functions"
      xmlns:t="http://myfaces.apache.org/tomahawk">

 THIS TEXT WILL BE REMOVED
  <ui:composition>
```

```
    <!--  The label is optional. Generate it if it is missing. -->
    <c:if test="${empty label}">
  <c:set var="label"
                  value="${arc:getFieldLabel(fieldName,namespace)}" />
    </c:if>

    <!-- The required attribute is optional,
         initialize it to true if not found. -->
    <c:if test="${empty required}">
        <c:set var="required" value="true" />
    </c:if>


    <h:panelGroup>
        <h:outputLabel id="${fieldName}Label"
                       value="${label}" for="#{fieldName}" />
    </h:panelGroup>

    <!--  Initialize the value binding -->
    <arc:setValueBinding var="vb"
                         valueBinding="#{entity[fieldName]}" />

    <!--  If the value binding is a string,
          display an inputText field. -->
    <arc:isText id="vb">

        <h:inputText id="#{fieldName}"
                     value="#{entity[fieldName]}"
                     required="${required}" styleClass="fieldInput">
            <ui:insert />
        </h:inputText>
    </arc:isText>

    <!--  If the value binding is a boolean, display a
           selectBooleanCheckbox field. -->
    <arc:isBoolean id="vb">
        <h:selectBooleanCheckbox  id="#{fieldName}"
                                  value="#{entity[fieldName]}"
                                  required="${required}" />
    </arc:isBoolean>

    <!--  If the value binding is a date,
          display a t:inputDate field. -->
    <arc:isDate id="vb">
        <t:calendar id="#{fieldName}" renderPopupButtonAsImage="true"
                    renderAsPopup="true" value="#{entity[fieldName]}"
                    required="${required}" styleClass="fieldInput">
              <ui:insert />
        </t:calendar>
    </arc:isDate>

    <!--  Display any error message that are found -->
    <h:message id="${fieldName}Message"
               style="color: red; text-decoration: overline"
               for="#{fieldName}" />

</ui:composition>
 THIS TEXT WILL BE REMOVED AS WELL

</html>
```

これで、もし値バインディングの型がIntegerや
String、BigDecimal、Character、Long、Short、Byte、Float、 Double、short、int、charなどの場合には、フィールド・タグはinputTextフィールドを表示するようになります。またフィールド・

タグは、値バインディングの型がbooleanあるいはBooleanの場合には、チェック・ボックスを表示します。そして値バインディングの型がjava.util.Dateの場合には、Tomahawkカレンダー・コンポーネントを表示します。これは正直、非常にカッコ良いと言えます。

まとめ

この記事では、少しばかり面白いものを紹介しながら、Faceletsの強力な応用について説明しました。FaceletsのEL機能を利用すると、国際化に伴う苦労を軽減することができ、またカスタムのロジック・タグを使えば、Faceletsに組み込まれたロジック・タグを拡張することができます。これらの説明から、前回の記事で使用した単純な例を拡張して、より高度な機能を持たせることがいかに簡単か理解できたと思います。またこの記事では、テキストとして表示すべき日付（Tomahawkカレンダー）やブール表現（チェック・ボックス）、その他様々なJava型を扱えるようにフィールド構成コンポーネントを更新する方法についても触れました。さらに、合理的なデフォルトについても紹介し、これを使ってフィールド・コンポーネントに対するラベルの国際化を行いました。Faceletsで他にどんな面白いことができるか、今後も期待していてください。

ダウンロード

内容	ファイル名	サイズ
Source code	j-facelets2.zip	12KB

著者について

Richard Hightower

Rick Hightowerは、JSFやSpring、Hibernateなどに関するトレーニングの専門会社であるArcMind Incの最高技術責任者です。J2EE開発への極限プログラミングの応用を解説した人気の著書『Java Tools for Extreme Programming』の共著者であり、また『Professional Struts』の共著者でもあります。