JSFを信じない人のために: JSFに関するFUDをクリアーする

JavaServer Facesは思ったよりも簡単です

Rick Hightower (webserv@us.ibm.com)
CTO
LearningPattern

2005年 2月 03日

必要不可欠の技術であるにも関わらず、JSF（Java ™ Server Faces）に関しては、理不尽なFUD（Fear, Uncertainty, and Doubt）が起きています。噂によると、JSF開発は難しく、一部の主流な手法よりも多くを要求し、動かすためにはWYSIWYGツールが必須だと言うのです。この4回構成の新シリーズでは、developerWorksに頻繁に投稿しているRick Hightowerが、事実とFUDを切り分けます。自分が何をしているかをよく理解すれば、実際にはJSFの方が、StrutsのようなMVC Model 2フレームワークよりも容易なのです。

だいぶ以前から、J2EEのJSF（JavaServer Faces）技術に関してFUD、つまりFear, Uncertainty, and Doubtが飛び交っています。そろそろ、これを止めるべき時であり、少なくともバランスの取れた見方をすべき時だと思います。JSFに関する第1の誤解は、JSF開発にはWYSIWYGのドラッグ・アンド・ドロップ・ツールが必要だというものです。2番目の誤解は、JSFがStrutsのようなMVC Model 2フレームワークをサポートしないというもの、そして最後に、最も広く行き渡っている誤解は、とにかくJSF開発は難しいというものでしょう。

この4回シリーズでは、できるだけ現実的な方法で、3つの誤解全てを解消しようと思います。実際のところ、JSF開発が難しいと思っている人は、正しいやり方をしないために難しいと思ってしまうのです。幸いこれは、簡単に修正できます。今月はまず、JSFのアーキテクチャー的な概要と、MVCとJSFの基本を示すような実際例から始めることにします。その前に、ちょっと時間を使って、JSFに関するFUDと事実を切り分けることにしましょう。

FUDを信じないで下さい！

先に書いた通り、JSFに関しては3つの大きな誤解がありますが、その第1は、作業のためにWYSIWYGツールが必要だというものです。これは全くナンセンスです。Swing開発者の多くはSwingアプリケーションを開発するためにWYSIWYGを使わないのと同じように、JSFアプリケーションを構築するためにWYSIWYGエディターは必要ありません。実際、WYSIWYGツールを使わないJSF開発は、StrutsやWebWorkのような伝統的Model 2フレームワークを使った開発よりも、ずっと簡単なのです。なぜそうなのかの詳細は、この記事の後の方で説明しますが、とりあえず、次を読んでください。つまり、『WYSIWYGツールを使わなくても、JSF開発はStrutsよりもずっと簡単なのです！』

JSFに関する第2の誤解は、JSFがModel 2アーキテクチャーをサポートしない、というものです。これは実のところ、一部は事実です。実はModel 2というのは、MVC（Model-View-Controller）を、サーブレット上に構築されるWeb開発用に水で薄めたものなのです。Model 2は主にステートレス・プロトコル（HTTP）を対象にしていますが、JSFは、より表現力豊かなMVCモデルをサポートしており、昔からのGUIアプリケーションに、より近いのです。MVCにベースを置いているため、JSFフレームワークの実装は他のフレームワークよりも構築が難しくなっていますが、利点として、JSF実装のための実作業の多くは既にできあがっています。ですから、実質的な努力は少なくて済み、逆に得るところはずっと多いのです。

このシリーズについて

この4回シリーズ の記事は、JSF（JavaServer Faces）技術に対するFUDを解消することを目的としており、順を追って、分かりやすいフォーマットで実際に体験することを主眼にします。4回の記事を通して、一連の例を使ってJSFの基本的なアーキテクチャーや特徴、機能などを紹介して行きます。一旦JSF流のやり方に慣れてしまうと、Struts Model 2流の開発方法に戻るのが難しくなるでしょう。結局のところ、JSFでのイベント駆動、GUIコンポーネント・モデルを経験した後で、XMLコンフィギュレーションの死の世界に戻ろうとする人はいないと思います。

このシリーズを最大限に利用するためには、JavaプログラミングやJavaBeansコンポーネント（つまりイベント・モデルとプロパティー）、JavaServer Pages技術、JSP Standard Tag Library Expression Language、そしてWeb開発に関する基本概念を理解している必要があります。

JSF開発に関して最も広く蔓延している誤解は、難しい、というものでしょう。私がこれを最も頻繁に聞くのは、JSFについて多くを読みながら自分では実際に試したことのない人達からなので、これは簡単に解消できます。確かに、ライフサイクル・ダイアグラムや図などを含め、JSFの仕様は多岐に渡っているため、それを見ただけで判断しようとすると、尻込みしたくなるのも無理はありません。しかし、こうした仕様はツールを実装する人達のためのものであり、いわゆるアプリケーション開発者のためのものではないことを理解すべきです。先に書いた通り、JSFフレームワークは、アプリケーション開発者にとっては、驚くほど簡単になるように作られているのです。

実際、JSFの持つ、コンポーネント・ベースでイベント駆動のGUI開発モデルは、Javaの世界では多少新しいものですが、他では以前から存在していたものです。AppleのWebObjectsはJSFのアーキテクチャーと似ており、ASP.netも同様です。Tapestryは、オープンソースでJavaベースのWebコンポーネント・フレームワークとしてJSFとは少し異なった手法をとっていますが、同じくWeb GUIコンポーネント・モデルの上に構築されています。

FUDについては、これくらいで十分でしょう。JSFに関する先入観を解消するためには、とにかくJSF技術の中に飛び込んでみることです。ここではJSFを初めて見る人のために、まずアーキテクチャーの概要から始めることにしましょう。

初心者のためのJSF

JSFはSwingやAWTと同様、一連の標準的、再利用可能なGUIコンポーネントを提供する開発フレームワークです。JSFはWebアプリケーション・インターフェースを構築するために使われます。JSFを使った開発には次のような利点があります。

- 振る舞いとプレゼンテーションが、きれいに分かれている

- ステートフルな状態に対してコンポーネント・レベルで制御できる
- イベントを容易にサーバー側コードに結びつけられる
- 使い慣れたUIコンポーネントやWeb階層概念を利用できる
- 標準化されたベンダー実装が複数用意されている

典型的なJSFアプリケーションは、次のような要素から構成されています。

- アプリケーションの状態と振る舞いを管理するためのJavaBeansコンポーネント
- イベント駆動開発（昔からのGUI開発の場合と同様、リスナーを経由して）
- MVC風のビューを表現するページ（ページはJSFコンポーネント・ツリーを通してビュー・ルートを参照します）

JSFを使うためには、少しばかり概念的なハードルを超える必要がありますが、苦労するだけの価値はあるものです。JSFの持つ、コンポーネント状態管理や使いやすいユーザー入力検証、きめ細かい、コンポーネント・ベースのイベント処理、拡張が容易なアーキテクチャーなどを利用することによって、Web開発が大幅に単純化されます。これから先のセクションでは、こうした特徴の中で最も重要なものを、詳しく説明して行きます。

コンポーネント・ベースのアーキテクチャー

JSFでは、標準のHTMLで利用できる入力フィールドの全てに対して、コンポーネント・タグを提供しています。また、アプリケーション特有の目的のために、あるいは、例えば3つのドロップダウン・メニューから成るData Pickerコンポーネントなど、複数のHTMLコンポーネントを組み合わせたコンポジットを構成するために、カスタムのコンポーネントを書くこともできます。JSFコンポーネントはステートフルです。コンポーネントがステートフルであるのは、JSFフレームワークによるものです。JSFは、コンポーネントを使ってHTMLレスポンスを作ります。

JSFのコンポーネント・セットには、イベント公開モデル、つまり軽量IoCコンテナーが含まれており、また、一般的なGUI機能のほとんど（つまり交換可能なレンダリングやサーバー側検証、データ変換、ページ・ナビゲーション管理など）に対するコンポーネントも含まれています。JSFのアーキテクチャーはコンポーネント・ベースなため、構成や拡張が非常に容易です。JSF機能の大部分、つまりナビゲーションや管理対象bean（managed bean）ルックアップは、交換可能なコンポーネントで置き換えることができます。このように交換可能なため、WebアプリケーションGUIの構築が非常に柔軟になり、他のコンポーネント・ベース技術を容易にJSF開発の中に採り入れられるようになります。例えば管理対象beanルックアップ用に、JSF組み込みのIoCフレームワークの代わりに、より機能の完全な、IoC/AOP Springフレームワークで置き換えることができます。

JSFとJSP技術

JSFアプリケーションのユーザー・インターフェースは、JSP（JavaServer Pages）ページから成り立っています。それぞれのJSPページには、GUI機能を表すJSFコンポーネントが含まれています。JSPページ内部でJSFカスタム・タグ・ライブラリーを使って、UIコンポーネントを描画し、イベント・ハンドラーを登録し、コンポーネントをバリデーターと関連付け、コンポーネントをデータ・コンバーターと関連付け、等々を行います。

とは言っても、JSFは本来的にJSP技術に結びついているわけではありません。実際、JSPページが使うJSFタグは、コンポーネントを表示するために単にコンポーネントを参照するだけです。これ

に初めて気がつくのは、JSPページを修正してJSFコンポーネントの属性を変更し、そのページを再ロードしても何も起こらないのを見た時でしょう。これはタグが、現在の状態でのコンポーネントを参照するためです。従って、コンポーネントが既に存在していれば、カスタム・タグはその状態を変更しません。コンポーネント・モデルを使うと、コントローラー・コードがコンポーネントの状態を変更でき（例えばテキスト・フィールドを使用不可にできます）、そのビューが表示される時には、コンポーネント・ツリーの現在の状態が表示されます。

典型的なJSFアプリケーションではJavaコードは全く必要なく、UIにはJSTL EL（JSP Standard Tag Library, Expression Language）コードがほとんど必要ありません。先にも書いた通り、JSFでアプリケーションを構築したり組み立てたりするためのIDEツールは豊富にあり、JSFのGUIコンポーネントに対しては、サードパーティーのマーケットが成長しつつあるようです。また、WYSIWYG ツールを使わずに、JSFをコード化することもできます。

JSFとMVC

MVCについて

MVC（model-view-controller）アーキテクチャーは、一連の設計パターンを提供します。これを利用することによって、GUIベースのアプリケーションを構築、実行する上での関心領域を分離することができます。モデルは、アプリケーションに対するビジネス・ロジックとパーシスタンス・コードをカプセル化します。モデルは、できるだけビュー技術と無関係にすべきです。例えば、SwingアプリケーションでもStrutsアプリケーションでも、JSFアプリケーションでも、同じモデルが使えるべきです。ビューは、モデル・オブジェクトを表示し、プレゼンテーション・ロジックのみを含むべきです。ビューの中には、ビジネス・ロジックやコントローラー・ロジックを含むべきではありません。コントローラーは（関連ロジックと共に）、ビューとモデルの間の仲介者として動作します。コントローラーはモデルとやり取りし、表示のためにモデル・オブジェクトをビューに渡します。MVCアーキテクチャーでは、コントローラーが常に新しいビューを選択します。

JSFは、ここ数年に渡るJavaプラットフォームでのWeb開発技術の進化の中で得られた教訓の成果です。この流れはJSP技術に始まっています。JSP技術は便利なのですが、HTML（そしてHTML風の）ページにJavaコードを混在させることが、非常に容易にできてしまいます。次のステップはModel 1アーキテクチャーでした。このアーキテクチャーでは、開発者はバックエンド・コードの大部分をJavaBeansコンポーネントに押し込み、後で `<jsp:useBean>` タグを使ってWebページの中にJavaBeansコンポーネントをインポートするのです。これは単純なWebアプリケーションではうまく行くのですが、Java開発者の多くは、JSP技術の中に静的なincludeなどのC++機能を取り込むことを嫌いました。そこでModel 2アーキテクチャーが導入されたのです。

基本的にModel 2アーキテクチャーは、Webアプリケーション用にMVCを水で薄めたものです（MVCについて、を見てください）。Model 2アーキテクチャーでは、コントローラーはサーブレットで表現され、表示はJSPページに委任されます。Strutsは単純化したModel 2実装であり、アクションがサーブレットに取って代わっています。Strutsでは、アプリケーションのコントローラー・ロジックは、そのデータ（ActionFormsで表現されます）と分離されています。Strutsに対する主な不満は、オブジェクト指向と言うよりはプロシージャー的な面に対するものです。WebWorkとSpring MVCは、あまりプロシージャー的でないようにStrutsを改善した、Model 2アーキテクチャーを持つものですが、どちらもStrutsほど広く受け入れられてはいません（一部の人は、成熟していないとも主張しています）。それにどちらも、JSFが提供するようなコンポーネント・モデルを提供していません。

大部分のModel 2フレームワークの真の問題は、イベント・モデルが単純すぎる（基本的に非常に規模を縮小したMVCです）ため、開発者の作業負担が大きくなりすぎることです。表現力豊かなイベント・モデルを使えば、大部分のユーザーが期待するようなやり取りを、簡単に作ることができます。また大部分のModel 2フレームワークでは、JSP技術と同様、HTMLレイアウトやフォーマットが、一見コンポーネントのように動作するGUIカスタム・タグと、あまりにも容易に混在できるようになっています。そして（Strutsなど）一部のModel 2アーキテクチャーでは、振る舞いと状態の分離に失敗しているため、多くのJava開発者にはCOBOLでプログラミングをしているように思えてしまうのです。

表現力を改善したMVC環境

JSFはコンポーネント・モデルを提供し、また大部分のModel 2実装よりも表現力豊かなMVC環境を提供しています。基本的にJSFは（相変わらずステートレスなプロトコルですが）、Model 2アーキテクチャーよりも、真のMVCプログラミング環境に近いのです。JSFはまた、Model 2フレームワークよりも、よりキメの細かいイベント駆動GUIを作りやすくなっています。JSFには、メニュー・アイテム選択やボタン・クリックなど、イベント・オプションが豊富にありますが、大部分のModel 2は、もっと単純な、「要求受信（request received）」に頼っています。

StrutsとJSF

Strutsフレームワークは、JavaプラットフォームでのWebフレームワークの開発における必然的な進化でした。StrutsはModel 2フレームワークの限界を押し広げました。Strutsで使われている考え方は、JSTLやJSFにと、進化を遂げました。Model 2開発モデルに対応するために生まれた多くのプロジェクトは、Strutsで最初に展開された考え方を、（通常は大幅に改善を加えて）使用しました。さらにJSFプロジェクトの中には、（アーキテクチャー的には多様ですが）StrutsのDNAがまだ数多く残っています。例えば、JSFでは相変わらずTilesを使うことができ、Strutsのバリデーター・フレームワークを使ってクライアント側JavaScriptを生成することもできます。一部には、StrutsとJSFを統合しようという動きさえあります。とは言え私の意見では、Strutsを必要とする場面の大部分で、JSFはStrutsに取って代わることができます。（少なくとも現状のStrutsに関する限り）。Shaleと言われる、Strutsの次期主要バージョンでは、JSFのためにStrutsコアの一部を捨て去っています。このシリーズを最後まで読み、自分のプロジェクトでStrutsに置き換わるものとしてJSFが使えるかどうか、皆さん自身で判断してみてください。

JSFのイベント・モデルはよく調整されているため、アプリケーションとHTTPとが詳細に結びつくことが少なく、開発作業も単純になります。また、JSFは伝統的なModel 2アーキテクチャーの改善も図っており、プレゼンテーションとビジネス・ロジックをコントローラーから除外しやすく、ビジネス・ロジックをJSPページから除外しやすくなっています。実際、単純なコントローラー・クラスはJSFと全く結びついておらず、テストがしやすくなっています。真のMVCアーキテクチャーとは異なり、1つ以上のビューポートで解決すべき多くのイベントをJSFモデル階層が発行することは、あり得ません。繰り返しますが、あくまでもステートレス・プロトコルを扱っているので、これは不必要なのです。ビューを変更したり更新したりするシステム・イベントは、ほとんど常に（『常』にと言うべきかも知れません）、ユーザーからのリクエストなのです。

JSFでのMVC実装の詳細

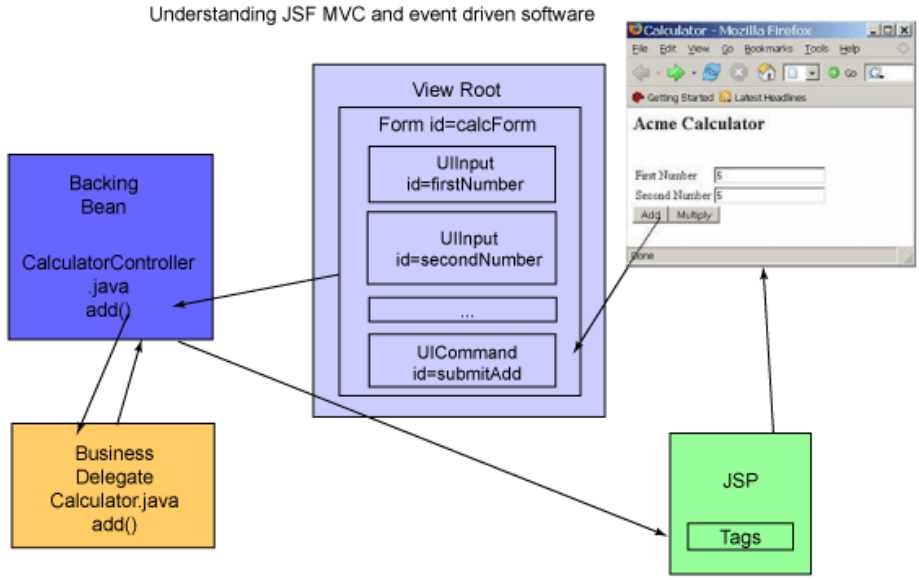
JSFのMVC実装では、マッピングを行うバッキングbean(backing beans) は、ビューとモデルの間の仲介を行います。このためバッキングbeanでは、ビジネス・ロジックとパーシスタンス・ロジックを制限することが重要です。一般的な方法としては、ビジネス・ロジックをアプリケーション・モデルに委任する方法です。この場合、ビューがモデル・オブジェクを表示できるとこ

ろでは、バッキングbeanはモデル・オブジェクトもマップします。もう一つの選択肢は、ビジネス・ロジックを、モデルとして動作するファサード（facade）であるビジネス処理代行（Business delegate）に入れる方法です。

JSP技術とは異なり、JSFのビュー実装はステートフル・コンポーネント・モデルです。JSFビューは、ビュー・ルートとJSPページという、2つの部分からなっています。ビュー・ルートというのは、UIの状態を保持するUIコンポーネントの集合です。JSFコンポーネントはSwingやAWT同様、複合設計パターン（Composite design pattern）を使ってコンポーネントのツリーを操作します（単純に言うと、コンテナーはコンポーネントを含み、コンテナーはコンポーネントです）。JSPページはUIコンポーネントをJSPページにバインドし、それによってフィールド・コンポーネントをバッキングbeanのプロパティー（というよりも、むしろプロパティーのプロパティー）に、ボタンをイベント・ハンドラーやアクション・メソッドにバインドできるようにします。

下記は、MVCの視点から見たアプリケーションの例です。このアプリケーションの詳細について、この先で説明します。

図1. MVCの視点から見たアプリケーションの例



前置きはこのくらいで十分として、JSFにとりかかることにしましょう。

JSFの例

この記事のこれから先では、JSFでアプリケーションを実際に作る上での手順に焦点を当てます。このサンプル・アプリケーションは、JavaServer Faces技術を非常に単純に示すものです。このアプリケーションから、次のことを学ぶことができます。

- デプロイのためのJSFアプリケーションを配置する方法
- JSF用のweb.xmlファイルを構成する方法
- アプリケーション用のfaces-config.xmlを構成する方法
- Model beans（別名バッキングbean）の書き方
- JSP技術を使ってビューを構築する方法

- カスタムのタグ・ライブラリーを使って、ビュー・ルートにコンポーネント・ツリーを構築する方法
- フォーム・フィールドのデフォルト妥当性検証

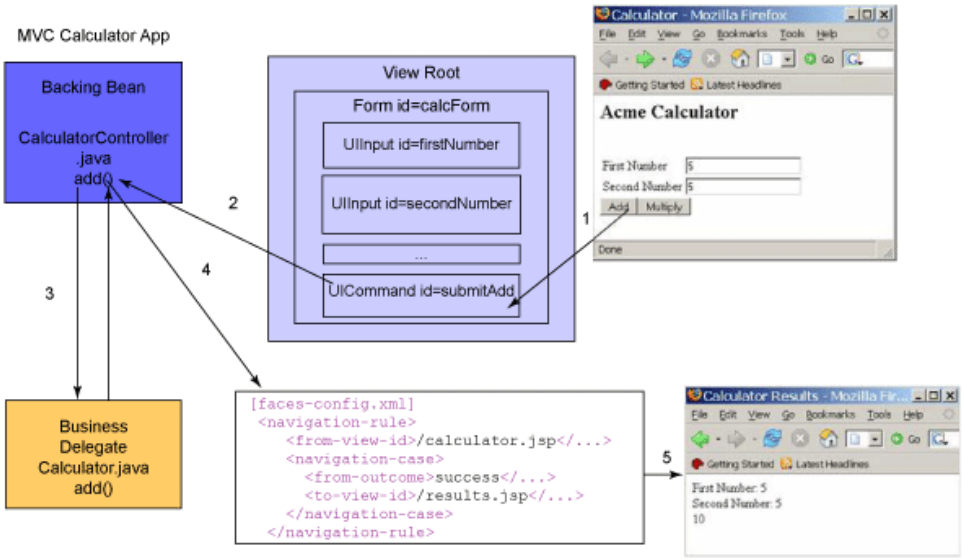
Mavenでビルドする

サンプルの電卓アプリケーション用のビルド環境は、デフォルトでMavenです。Mavenのビルド・システムはAntに似ています。ここで使った例では、MavenのWebアプリケーション用のデフォルト・レイアウトを使いました。従ってアプリケーションのJavaクラス・ファイルは、src/javaの下の、プロジェクト・ルートにあります。Webアプリケーション・ファイルは、src/webappの下に含まれています。ここには、JSPページやfaces-config.xml、web.xmlなどを含めた、私のWebアプリケーション専用のファイルを置いています。プロジェクト・フォルダーのルートにあるproject.xmlファイルは、Mavenがwarファイルをビルドし、パッケージするために何が必要か、どこに必要なファイルがあるかを記述しています。project.propertiesファイルには、さらにプロパティーがあり、ローカル環境特有のものに対してどうすべきか、それらをどこで見つけるべきかをMavenに伝えます。私は、Mavenを使いたくないというAntユーザーのために、Antのbuild.xmlも生成しておきました。Antを使うように選択した場合、サンプルのビルドや環境の設定に関する細かな手順などは、build.xmlファイルと、 参考文献 のセクションを見てください。

このサンプルは、単純な電卓アプリケーションです。アプリケーションを作成する上での目標は、2つの数字を入力できるページをエンド・ユーザーに示すことです。従ってこのページには、2つのテキスト・フィールドと2つのラベル、2つのエラー・メッセージ位置、そしてSubmitボタンがあります。テキスト・フィールドは、数字を入力するためのものです。ラベルは、テキスト・フィールドにラベルを付けるためのものです。エラー・メッセージ位置は、テキスト・フィールドに対する検証の結果や、データ変換でのエラー・メッセージを表示するためのものです。JSPページは3つあります。index.jspは単にcalculator.jspにリダイレクトします。calculator.jspは上記のGUIを表示し、results.jspは結果を表示します。 `CalculatorController` という管理対象beanは、calculator.jspとresults.jspに対するバッキングbeanとして動作します。

図2は、電卓アプリケーションの2番目のMVCビューを示しています。このアプリケーションのソースは、このページの先頭または最後にある Code アイコンをクリックすればダウンロードすることができます。

図2. サンプル・アプリケーションの、2番目のMVCビュー



アプリケーションを構築する

JSFで電卓アプリケーションを構築するには、下記をする必要があります。

1. サンプル・アプリケーションのsrc/webapp/WEB-INFディレクトリーにある、web.xmlファイルとfaces-config.xmlファイルを集める
2. FacesサーブレットとFacesサーブレット・マッピングをweb.xmlファイルの中で宣言する
3. web.xmlファイルの中でfaces-config.xmlファイルを規定する
4. どのbeanがJSFで管理されるかを、faces-config.xmlファイルの中で宣言する
5. ナビゲーション・ルールをfaces-config.xmlファイルの中で宣言する
6. モデル・オブジェクト `Calculator` をビューする
7. `CalculatorController` を使って `Calculator` モデルに伝える
8. index.jspページを作る
9. calculator.jspページを作る
10. results.jspページを作る

ステップ1は単なる設定なので省略し、他の各ステップを詳しく説明しましょう。

Facesサーブレットとサーブレット・マッピングを宣言する

Facesを使うためにはまず、下記のように、web.xmlファイルの中にFacesサーブレットをインストールする必要があります。

```
<!-- Faces Servlet -->
<servlet>

<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup> 1 </load-on-startup>
</servlet>
```

これは、一般的なweb.xmlディスクリプターと非常に似ていますが、自分独自のサーブレットを規定する代わりに、JSFサーブレットに制御を渡してリクエストを処理する点が異なっています。f:viewを使う、JSPファイルへの全リクエストは、このサーブレットを通す必要があります。従って、マッピングを追加し、そのマッピングを通して、JSF対応のJSP技術のみをロードする必要があります。これを次に示します。

```
<!-- Faces Servlet Mapping -->
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/calc/*</url-pattern>
</servlet-mapping>
```

上記は、 `/calc/` にマップする全リクエストをFacesサーブレットに送って処理するように、Facesサーブレット・コンテナーに伝えています。これによって、JSFがJSFコンテキストとビュー・ルートを初期化できるようになります。

faces-config.xmlファイルを規定する

facesコンフィギュレーション・ファイルにfaces-config.xmlという名前を付け、Webアプリケーションの WEB-INFディレクトリーに置くと、Facesサーブレットがそれを取り上げ、自動的にそれを使います（それがデフォルトのため）。あるいは、web.xmlファイルの中の初期化パラメーター（javax.faces.application.CONFIG_FILES）でカンマ区切りのファイル・リストを引き数として、1つ、またはそれ以上のアプリケーション・コンフィギュレーション・ファイルをロードすることもできます。ごく単純なJSF Webアプリケーションを除いて、皆さんは恐らく2番目の手法を使うと思います。

bean管理を宣言する

次に、どのbean がJSFのGUIコンポーネントで使われるのかを宣言します。このサンプル・アプリケーションには、管理対象beanは1つしかありません。これはfaces-config.xmlの中で、次のように構成されます。

```
<faces-config>
...
<managed-bean>
<description>
The "backing file" bean that backs up the calculator webapp
</description>
<managed-bean-name>CalcBean</managed-bean-name>
<managed-bean-class>com.arcmind.jsfquickstart.controller.CalculatorConroller</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>
```

上記のconfigは、 `CalcBean` というbeanをJSFコンテキストに追加したい、ということをJSFに対して伝えています。管理対象beanの呼び方は任意です。beanを宣言したら、次のステップは、アプリケーションに対する高位レベルのナビゲーション・ルールを宣言することです。

ナビゲーション・ルールを宣言する

この単純なアプリケーションで必要なこととしては、calculator.jspページからresults.jspページへのナビゲーション・パスを確立することだけです。これを下記に示します。

```
<navigation-rule>
<from-view-id>/calculator.jsp</from-view-id>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/results.jsp</to-view-id>
</navigation-case>
</navigation-rule>
```

上記は、アクションが /calculator.jspビューから論理結果「success」を返したら、ユーザーを /results.jspビューに進めると言っています。

モデル・オブジェクトをビューする

ここでの目標はJSFを使い始める手がかりを示すことなので、モデル・オブジェクトは非常に単純にしています。このアプリケーションのモデルは、1つのモデル・オブジェクトの中に含まれています。これをリスト1に示します。

リスト1. 電卓アプリケーションのモデル・オブジェクト

```
package com.arcmind.jsfquickstart.model;
/**
* Calculator
*
* @author Rick Hightower
* @version 0.1
*/
public class Calculator {
//~ Methods ------------------------------------------------------------
/**
* add numbers.
*
* @param a first number
* @param b second number
*
* @return result
*/
public int add(int a, int b) {
return a + b;
}
/**
* multiply numbers.
*
* @param a first number
* @param b second number
*
* @return result
*/
public int multiply(int a, int b) {
return a + b;
}

}
```

これで、ビジネス・ロジックが全て設定されました。次のステップは、これをWebアプリケーション・インターフェースに接合することです。

モデルとビューを接合する

コントローラーが目標とするのは、モデルからビューへの接着剤として動作することです。`Controller` オブジェクトの機能の一つは、ビュー技術とモデルを無関係に保つことです。下記を見てもらうと分かると思いますが、コントローラーは、入力収集と結果表示のために使用す

る3つのJavaBeansプロパティーを規定しています。このプロパティーは、 `results` （ 出力 ）と `firstNumber` （ 入力 ）、そして `secondNumber` （ 入力 ）です。 `Controller` はまた、 `Calculator` オブジェクトにある同じ名前のオペレーションに委任する、2つのオペレーションも提示しています。リスト2は、 `CalculatorController` のコードを示しています。

リスト2. CalculatorController

```
package com.arcmind.jsfquickstart.controller;
import com.arcmind.jsfquickstart.model.Calculator;

/**
* Calculator Controller
*
* @author $author$
* @version $Revision$
*/
public class CalculatorConroller {
//~ Instance fields ------------------------------------------------------
/**
* Represent the model object.
*/
private Calculator calculator = new Calculator();
/** First number used in operation. */
private int firstNumber = 0;
/** Result of operation on first number and second number. */
private int result = 0;
/** Second number used in operation. */
private int secondNumber = 0;
//~ Constructors ---------------------------------------------------------
/**
* Creates a new CalculatorConroller object.
*/
public CalculatorConroller() {
super();
}
//~ Methods --------------------------------------------------------------
/**
* Calculator, this class represent the model.
*
* @param aCalculator The calculator to set.
*/
public void setCalculator(Calculator aCalculator) {
this.calculator = aCalculator;
}
/**
* First Number property
*
* @param aFirstNumber first number
*/
public void setFirstNumber(int aFirstNumber) {
this.firstNumber = aFirstNumber;
}
/**
* First number property
*
* @return First number.
*/
public int getFirstNumber() {
return firstNumber;
}
/**
* Result of the operation on the first two numbers.
*
* @return Second Number.
*/
```

```
public int getResult() {
return result;
}
/**
* Second number property
*
* @param aSecondNumber Second number.
*/
public void setSecondNumber(int aSecondNumber) {
this.secondNumber = aSecondNumber;
}
/**
* Get second number.
*
* @return Second number.
*/
public int getSecondNumber() {
return secondNumber;
}
/**
* Adds the first number and second number together.
*
* @return next logical outcome.
*/
public String add() {
result = calculator.add(firstNumber, secondNumber);
return "success";
}
/**
* Multiplies the first number and second number together.
*
* @return next logical outcome.
*/
public String multiply() {
result = calculator.multiply(firstNumber, secondNumber);

return "success";
}
}
```

リスト2で、 `multiply` メソッドと `add` メソッドが「 `success` 」を返すことに注意してください。文字列 `success` は、論理的な結果を意味します。これはキーワードではないことに注意してください。faces-config.xmlでナビゲーション・ルールを規定した時には、文字列 `success` を使いました。従って、加算または乗算操作を実行した後は、アプリケーションはユーザーをresults.jspページに進めます。

これでバッキング・コード(backing code)ができました。次は、アプリケーション・ビューを表す、JSPページとコンポーネント・ツリーを規定します。

index.jspページを作る

このアプリケーションにおけるindex.jspページの目的は、/calculator.jspページを確実にJSFコンテキストの中にロードし、このページが、対応するビュー・ルートを見つけられるようにすることです。index.jspページは次のようになります。

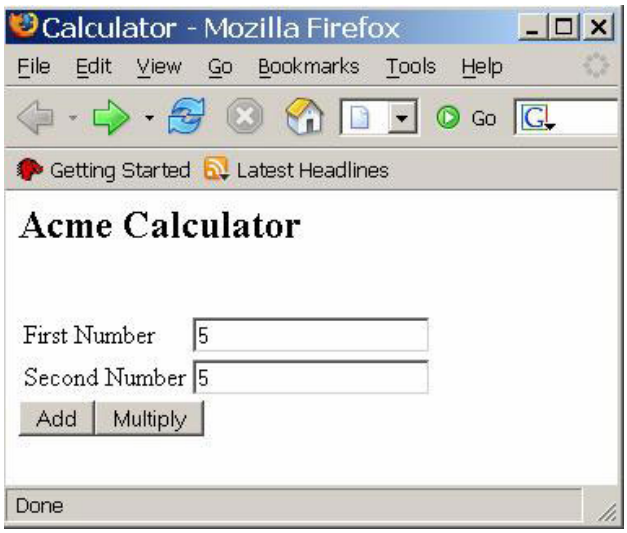
```
<jsp:forward page="/calc/calculator.jsp"
/>
```

このページがすることは、「 calc 」Webコンテキストの下にあるcalculator.jspにユーザーをリダイレクトすることだけです。これでcalculator.jspページはJSFコンテキストの下に置かれ、ここでcalculator.jspページは、そのビュー・ルートを見つけます。

calculator.jspページを作る

calculator.jspページは電卓アプリケーションのビューの核心です。図3に示すように、このページは、ユーザーから2つの数字を受け取ります。

図3. Calculatorページ



このページは複雑なので、どのように構築するのかを順を追って説明しましょう。まず、JSFに対するtaglibを次のように宣言することから始めます。

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

上記はJSPエンジンに対して、 html と core という2つのtaglibを使いたいことを伝えています。 html taglibは、フォームやその他、HTML特有のものを処理するためのタグを全て含んでいます。 core taglibは、論理タグや検証タグ、コントローラー・タグその他、JSF特有のタグを全て含んでいます。

このページを通常のHTMLでレイアウトした後、コンポーネント管理のためにJSFを使おうとしていることを、JSFシステムに対して伝えます。これは <f:view> タグを使って行います。これによって、コンポーネントの管理のためにJSFを使おうとしていることを、コンテナーに伝えます。（このタグを発音する時には必ず、「エフ、コロン、ビュー」と言うようにします。コロンを発音することが非常に重要です！）

<f:view> が無いと、JSFはコンポーネント・ツリーを構築できず、後で、既に作成されたコンポーネント・ツリーを参照することもできません。 <f:view> タグは次のように使います。

```
<f:view>
<h:form id="calcForm">
...    </h:form>
</f:view>
```

上記の第1行は <f:view> を宣言しており、これがJSFによって管理されていることをコンテナーに伝えます。次の行は <h:form> タグであり、JSFに対して、ここでHTMLフォームが欲しいと言っています。描画フェーズの期間中、フォーム・コンポーネント内部に含まれるコンポーネントは、参照され、自身を描画するように言われ、その時点で出力に対して標準HTMLを生成します。

次に、他にどんなコンポーネントがフォームの中で必要かをJSFに伝えます。 <h:form> の中では、 panelGrid を宣言します。 panelGrid は複合コンポーネント、つまり、他のコンポーネントを含むコンポーネントです。 panelGrid は、他のコンポーネントのレイアウトを規定します。 panelGrid は、リスト3に示すように宣言されます。

リスト3. panelGridを宣言する

```
<h:panelGrid columns="3"> <h:outputLabel value="First Number" for="firstNumber" />
<h:inputText id="firstNumber" value="#{CalcBean.firstNumber}" required="true" />
<h:message for="firstNumber" />
<h:outputLabel value="Second Number" for="secondNumber" />
<h:inputText id="secondNumber" value="#{CalcBean.secondNumber}" required="true" />
<h:message for="secondNumber" />
</h:panelGrid>
```

3に設定されている属性カラムは、3つのカラムを持つ格子の中にコンポーネントが配置されることを示しています。ここでは6つのコンポーネントを panelGrid に追加します。つまり2行ということです。各行は、 outputLabel と inputText 、そして message から成ります。ラベルとメッセージは inputText コンポーネントと関連付けられます。従って妥当性検証エラーまたはエラー・メッセージが textField に関連付けられると、メッセージが message コンポーネントに現れます。両方のテキスト・フィールドとも必須です。つまり、送信時に両方の値が入っていないとエラー・メッセージが作られ、制御は、このビュー、つまり /calculator.jspに戻ります。

どちらの inputFields も、値の属性にJSF EL (JavaServer Faces Expression Language) 値バインディングを使っていることに注意してください (例えば、 value="#{CalcBean.firstNumber}" ")。ちょっと見ると、これはJSTL ELとよく似ています。しかし実際には、JSTL ELコードはフィールドを、バッキングbeanプロパティーの対応する値と関連付けるのです。この関連付けは反射的です。つまり firstNumber が100であれば、フォームが表示された時には100が現れます。同様に、ユーザーが有効な値 (例えば200) を送信すると、200が firstNumber プロパティーの新しい値になります。

もっと一般的な (しかし、もう少し複雑な) 方法は、バッキングbeanがプロパティーでモデル・オブジェクトをエクスポーズし、こうしたモデル・オブジェクト・プロパティーをフィールドにバインドする方法です。この手法を使った例は、このシリーズの今後の記事で紹介します。

calcForm は、フィールドだけではなく、 panelGroup 内の2つの commandButton を使った 2つのアクションにも関連付けられています。これを次に示します。

```
<h:panelGroup>
<h:commandButton id="submitAdd" action="#{CalcBean.add}" value="Add" />
<h:commandButton id="submitMultiply" action="#{CalcBean.multiply}" value="Multiply" />
</h:panelGroup>
```

スタイルシートについて

JSFコンポーネントのルック・アンド・フィールは全て、スタイルシート・クラスを使って宣言されます。つまりコンポーネントはそれぞれ、インライン・スタイルを関連付けるためのスタイルと、コンポーネントを styleSheet クラスと関連付けるための styleClass を持ちます。 panelGrid も、スタイルを行とカラムに関連付けるためのスタイル属性を持ちます。今回は簡単にするために、スタイルシートは何も規定していません。

panelGroup は、何かを配置する方法が異なることを除いて、概念的に panelGrid に似ています。コマンド・ボタンは action="#{CalcBean.add}" を使って、ボタンをバッキングbeanのメソッドにバインドします。従って、ボタンでフォームが送信されると、関連付けられたメソッドが呼び出されます（全ての妥当性検証に問題が無い場合）。

これで、JSFアプリケーションをコーディングする上で一番の難関を乗り越えました。後のステップは楽なものです。

results.jspページを作る

results.jspページは、最後に行われた電卓操作の結果を表示するために使います。これは、リスト4に示すように定義します。

リスト4. results.jspページ

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
...
<f:view>
First Number: <h:outputText id="firstNumber" value="#{CalcBean.firstNumber}"/> <br />
Second Number: <h:outputText id="secondNumber" value="#{CalcBean.secondNumber}"/>
<br />
Result: <h:outputText id="result" value="#{CalcBean.result}"/> <br />
</f:view>
```

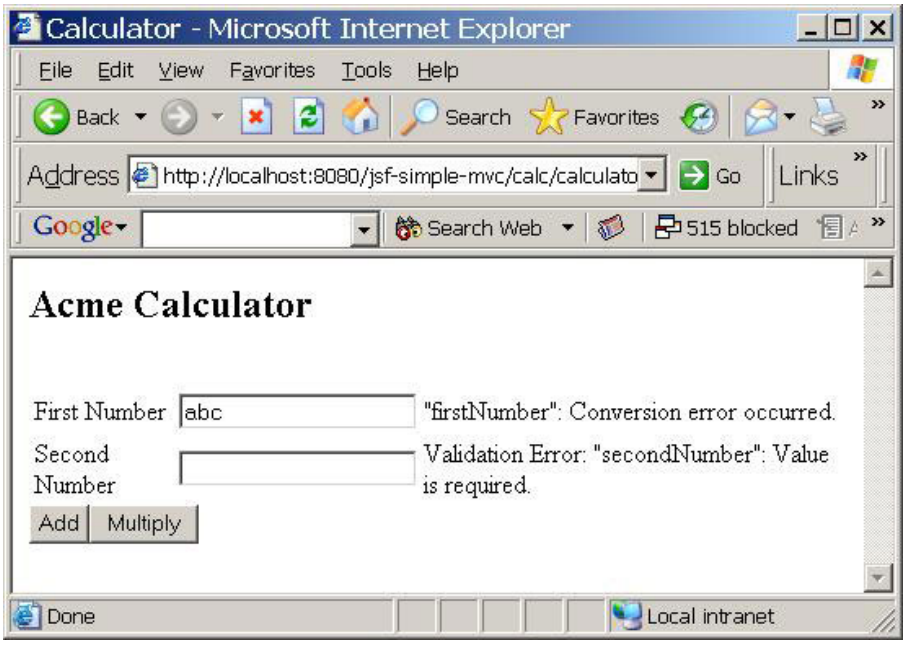
このresults.jspファイルは比較的単純なページで、加算の結果をユーザーに表示します。これは <outputText> タグによって行います。 <outputText> タグには、 id と value 属性があります。 value 属性は、描画の際にはbeanの値を文字列として出力します。 value 属性はJSFを使って、出力値をバッキングbeanプロパティー（先ほどの、 firstNumber と secondNumber 、そして result です）にバインドします。

アプリケーションを実行します！

このアプリケーションを実行するには、warファイルがマップされているページに行きます。そうすると、index.jspファイルがcalculator.jspページをロードします。 firstNumber フィールドまたは secondNumber フィールドに、何か無効なテキスト（例えば「abc」）を入力して送信すると、/calculator.jspビューに戻され、対応したフィールドの隣にエラー・メッセージが表示されます。 firstNumber フィールドまたは secondNumber フィールドを空白にしたまま送信すると、/calculator.jspビューに戻され、対応したフィールドの隣にエラー・メッセージが表示されます。つまりJSFでは、フィールドが必須であることを規定するだけ、そしてフィールドを int プロパティーにバインドするだけで、一部の妥当性検証が、ほとんど自動的に行われるのです。

図4は、このアプリケーションが妥当性検証エラーやデータ変換エラーをどう処理するかを示しています。

図4. 妥当性検証エラーとデータ変換エラー



まとめ

JSFを紹介する今回の記事を読んだだけでは、まだ皆さんも何度か、首をかしげざるを得なかったでしょう。しかし、心配は要りません。最も困難な部分は乗り越えました。JSFの概念的なフレームワークに入り込むだけで、この技術を実装するための戦いの半分以上は終わりなのです。そして、これが苦労するだけの価値があることが、皆さんにもすぐに分かるでしょう。

Strutsを使った方が簡単だと思っている人達のために念のために言うと、ここで構築した、単純なJSFアプリケーションのStruts版を作るためには、少なくとも2倍の努力が必要だと思います。これと同じサンプル・アプリケーションをStrutsで作るためには、2つのボタンに対して2つのアクション・クラスが必要であり、それぞれに独自のアクション・マッピングのセットが必要です。また、少なくとも皆さんがModel 2の勧告に従うとしても、最初のページをロードするためのアクション・マッピングも必要です。またJSFではデフォルトの、エラー処理や妥当性検証を真似するために、Strutsの構成としてバリデーター・フレームワークを使うか、あるいはそれ相当のものを `ActionForm` の `validate` メソッドの中に実装する必要があります。さらに、Struts configの中で `DynaValidatorForm` を宣言するか、あるいは `ActionForm` を作って `validate` メソッドをオーバーライドするか、あるいはバリデーター・フレームワークへのフックを持つ `ValidatorForm` のサブクラスを使う必要もあります。そして最後に、全アクションが使用するような、何らかの転送(forwards)（恐らく各アクションに対して2セット）、あるいはグローバル転送(global forwards)を構成する必要があるでしょう。

Strutsでは、コーディングが倍になるだけではなく、新人開発者がStrutsを学ぶのも大変なのです。なぜ私がそんなことを言うかというと、私自身がStrutsの授業コースとJSFの授業コースの両方を作ったことがあり、その両方を教えたことがあるからです。開発者がJSFを学ぶのは容易ですが、Strutsでは非常に苦労します。Strutsに比べてJSFの設計には、はるかに多くのことが事前に考慮されていると思います。JSFは、より論理的であり、直感的です。Strutsは寄せ集めから進化してきました。JSFは仕様が決められてから作られました。私が考える限り、JSF開発の方がStruts開発よりも、とにかく生産的なのです。

これで、JSFシリーズの第1回は終わりです。次回の記事は、この続きから始めます。次回はJSFリクエスト処理ライフサイクルの主なフェーズについて説明し、サンプル・アプリケーションの各部分が、ライフサイクルのどの部分に対応するのかを解説します。また、即時イベント処理(immediate event handling)の概念も紹介し、JSFに同梱されている多数の組み込みコンポーネントの解説を含めて、JSFのコンポーネント・イベント・モデルの理解を深めます。さらに、JavaScriptとJSFの組み合わせにも触れることにしています。では、次回の記事をお楽しみに！

ダウンロード

内容	ファイル名	サイズ
Source code with JAR files	jsf-simple.zip	2051KB
Source code without JAR files	jsf-simple-no-jars.zip	33KB

著者について

Rick Hightower

Rick Hightowerは、ArcMind Inc.のCTO（最高技術責任者）です。J2EE開発への極限プログラミング応用を解説した本として人気の、Java Tools for Extreme Programmingの共著者であり、Professional Strutsの共著者でもあります。また、Warner OnstineのJSF QuickStartも作っており、このシリーズの内容の一部は、そのコースの中のサンプルに基づいています。彼の連絡先はrhightower@arc-mind.comです。