シームレスな JSF、第 1 回: JSF 用にあつらえたアプリケーション・フレームワーク

Seam ならではの JSF ライフ・サイクルの機能強化を探る

Dan Allen (dan.allen@mojavelinux.com)
Senior Java engineer
CodeRyte, Inc.

2007年 4月 17日

JSF (JavaServer Faces) は Java™ Web アプリケーション向けに初めて標準化されたユーザー・インターフェース・フレームワークです。このJSF を拡張する強力なアプリケーション・フレームワークが、Seam です。この 2 つのフレームワークの絶好の相性について、新しい 3 回連載の第1 回目となるこの記事で学んでください。Dan Allen が、コンテキストに依存した状態管理、RESTful な URL、Ajax remoting、適切な例外処理、そして構成上の規約など、Seamが持つ JSF ライフ・サイクルの拡張機能を紹介します。

JSF はその Java Web 標準という位置づけにより、Java Web アプリケーション市場の中心になりつつありますが、JSF を基盤としてアプリケーションを設計することを義務付けられた開発者が増えるにつれ、コア仕様に明確に規定された内容が理解されるようになってきています。JSFは完全な Web アプリケーション・フレームワークとなるように設計されたものではありませんでした。JSF はむしろ、より完全なアプリケーション・フレームワークをビルドするためのより頑健なイベント駆動型API および UI コンポーネント・ライブラリーを提供するためのものです。

JSF のコンポーネント指向アーキテクチャーを補完する拡張機能に関して調査したところ、Shale と Struts 2 はいずれも役不足なことがわかりました。Struts2 を除外した理由は、JSF をより広範な設計に対する付加的なものとしてしか扱わないからです。基本的に JSF をベースとする Shale は有力候補になりそうに思えますが、それにはいくつかの懸念事項があります。その一方、JBoss Seam は JSF の基礎の上に構築された包括的なアプリケーション・フレームワークとして、JSF が中核とする目標を何1 つ犠牲にしていません。

この連載について

「シームレスな JSF」で紹介する Seam は、JSF に申し分なく適合する初のアプリケーション・フレームワークとして、他のどの拡張フレームワークでも補えないようなJSF の主要な欠点を補います。この連載記事を読んで、Seam が JSF を補完するものとして十分かどうか、読者自身が判断してください。

この 3 回連載の記事では、Seam アプリケーション・フレームワークを紹介し、その長所について説明します。そして読者の皆さんが、Java エンタープライズ・アプリケーションを開発するには

JSF と一緒に Seam を使うのが最適であるという確信に至ることを願っています。この連載を読む前に Seam をダウンロードしたいという方は、「参考文献」セクションを参照してください。

待ち望まれていた Seam

Shale について

Shale には山あり谷ありの歴史があります。Shale は Struts と JSF の統合パッケージから発展したものですが、その後、Struts開発者から見放されてしまいました。ところが現在は、完全に JSF 専用の最重要 Apache プロジェクトとして存在しています。

Shale に対する私の懸念は、一連の疎結合サービスであるというその位置づけです。つまり、統合の負担は開発者に課せられます。Shale の Java5 言語拡張は有益ですが、すべて拡張パッケージに追いやられています。命名規則によってビュー・コントローラーを単一のテンプレートに結合するというのも非常に制限があります。また、Shaleのアプリケーション・コントローラーとダイアログ・マネージャーはどちらもアドオンとしては巨大であり、標準 JSF ライフ・サイクルの機能をあまりにも多く組み込みすぎているように見受けられます。

Seam は Shale に期待するすべての機能を、十分に統合された最適なパッケージで提供します。

JBoss Seam に関する記事 (「参考文献」を参照) を 1 ページ読んだだけで、Seam こそ私が捜し求めていたプロジェクトだということがわかりました。厄介な実世界の開発を十分経験してきたGavin King をはじめとする Seam の開発者たちにとっては、Web アプリケーション・フレームワークが初めから難しい課題に取り組まなければならないことは明らかでした。その難問とは、コンテキストに依存した状態管理、RESTfulかつユーザー・フレンドリーな URL、Ajax remoting、適切な例外処理、そして構成上の規約などです。Java 開発者にとってこの上なく幸せなことに、Seamはこのすべての要件以上のものを叶えます。もし読者が JSF を使用していて、まだ Seam について聞いたことがなければ、Seam の参考資料を一読してみることを是非お勧めします(「参考文献」を参照)。Seam に付属するこのマニュアルは、Seam が持つ最高の資産の 1 つです。

Seam が JSF を補完するものとして適していることは明らかですが、激しい競争のなかでは幾分無関心の憂き目に遭ってきています。Shaleと Struts 2 を含め、すでに多数の Web アプリケーション・フレームワークで溢れかえっている市場では、新参者は未熟なものとして扱われます。そのため、Seamには、多数派のなかでその地位を定着させるという必要がまだ残っています。さらに Seam の採用に時間がかかっているもう 1 つの理由として、このフレームワークに関するある種の俗説が、Seamの直接的なメリットに Java 開発者たちに目を開かせるのを妨げているということが挙げられます。

私が打ち砕きたいのは、Seam は EJB3 と併せて使用しなければ有益でない、あるいは Seam でアプリケーションを開発するには EJB3コンテナーが必要だという俗説です。実際、Seam の資料ではこの誤解を「Seam では、コンポーネントが EJB である必要はなく、EJB 3.0準拠のコンテナーがなくても使用できます」と否定しています。EJB3 を併用するのでなければ Seam を使用できないという主張は、Hibernateを使用しなければ Spring は使用できないと言うのと同じことです。どちらの組み合わせにしても相性は抜群ですが、互いに依存することはありません。

EJB3 とは何か

この後すぐに説明するように、Seam は貴重なフックとコンポーネント管理プロセスでデフォルトの JSF ライフ・サイクルを拡張します。さらに、Seam は EJB3 とまったく切り離して使用するこ

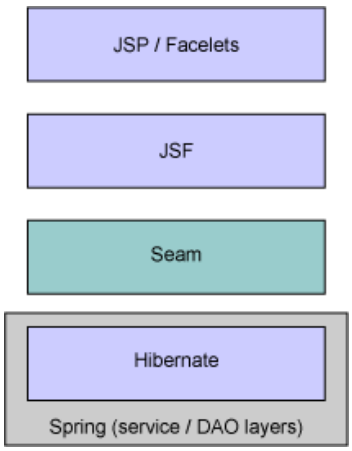
ともできます。ただし、Seam は EJB3 と同様、JDK 5 アノテーション・メタデータに依存してコンポーネントを宣言するので、Seamを使うには Java 5 準拠の JVM が必要だという点を念頭に置いてください。図 1 に、Seam POJO 実装のアプリケーション・スタックを示します。

JBoss Seam と JSR 299

JBoss Seam は、EJB3 および Spring Bean などのビジネス・コンポーネントとの JSF の統合を改善するために設計されたオープン・ソースのアプリケーション・フレームワークです。Seamは Web 環境のさまざまなコンテキストでコンポーネントを管理し、JSF アプリケーションを開発するのに必要な XML 構成をほとんど排除します。このプロジェクトは、Hibernateを作り出した Gavin King の独創的な考えから生まれたもので、現在は JBoss ラボでホストされています。

最近 JBoss が JCP に提出した、Seam の背後にあるコンセプトを標準化するという提案は、JSR 299: Web Beans として承認されました。JSFが管理する Bean コンポーネント・モデルを EJB3 コンポーネント・モデルと統一させることを目的としたこの仕様は、Web ベース・アプリケーションのプログラミング・モデルを大幅に単純化しています。

図 1. Seam POJO アプリケーション・スタック



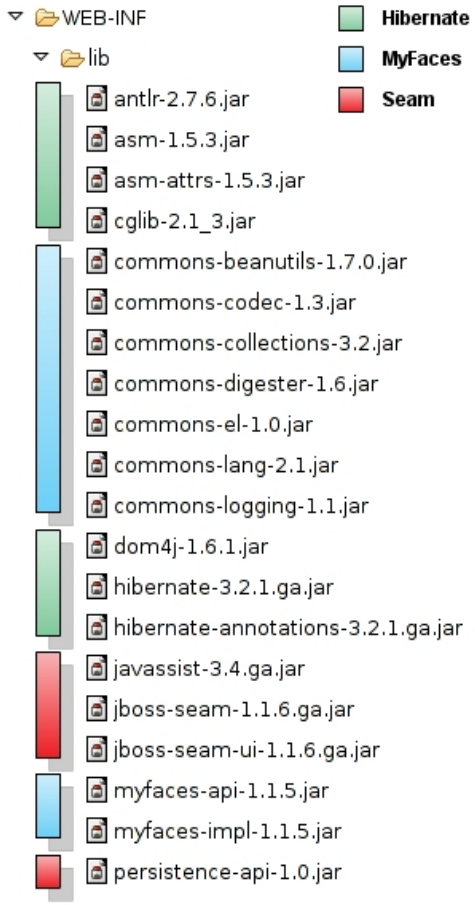
実際は、EJB3 jar や記述子ファイルを参照しなくても、Seam が持つ能力のほとんどを活用できます。Seam を POJO と併用すれば、フレームワークが引き続きコンポーネントのインスタンス化を完全に制御するので、特殊な構成は一切必要ありません。Seamは EJB3 のメカニズムに依存せずに、Java 5 アノテーション処理の大部分を操作します。EJB3 コンテナーに依存する限られたアノテーションのセットは、EJB3コンテナー環境に特化されているため、場合によっては EJB3 に結合せずに Seam を現行の IT 投資に統合することが一層のコスト効果につながることもあります。Seamを実際にどのように使用するかは、単に好みの問題です。

Seam を使用するための構成

膨大な数の Java フレームワークがある一方、1 日の作業時間は限られているという場合、統合しにくかったとしたら Seam に勝ち目はありません。幸い、Seamをプロジェクトに追加するのはわけないことです。JSF ライフ・サイクルはそのまま Seam アプリケーションの中心になるので、再教育に時間を費やす必要もありません。ただ単に4つの jar ファイルを追加してサーブレット・リスナーと JSF フェーズ・リスナーの両方を登録し、空の java プロパティー・ファイルに入れておけばいいだけの話です。このセットアップが完了すれば、固有のJSF アプリケーションを管理 Bean ごとに Seam にシフトできます。

Seam を使い始めるためにまず必要な作業は、必要な jar ファイルをプロジェクトに追加することです。Hibernate を使用していなかったり、最新バージョンへのアップグレードがまだ済んでいなかったりする場合は、セットアップの追加ステップとして、Hibernate 3.2 の配布の jar をその多数の依存関係と併せて組み込んでください。Seam はデータの妥当性検証に Hibernate アノテーションを使用するので、主要なHibernate jarの他に該当する拡張 jar も組み込む必要があります。Seam の配布から必要なライブラリーは jboss-seam.jarと jboss-seam-ui.jar、そして Javassist (Java のロード時反映システム) と Java PersistenceAPI という 2 つのサポート・ライブラリーです。図 2 のプロジェクト・ツリーに、Seam プロジェクト内に蓄積された一連の jar を示します。この図に示す追加ライブラリーのほとんどは、JSFのMyFaces 実装をサポートします。

図 2. Seam プロジェクトに含まれるライブラリー



Seam の構成

次のステップは、サーブレット・リスナー・クラスを web.xml ファイルにインストールすることです (リスト 1 を参照)。このリスナーは、アプリケーションがデプロイされるときにSeam を初期化します。

リスト 1. Seam サーブレット・リスナーの構成

```
<listener>
  <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
</listener>
```

次に、JSF フェーズ・リスナーを faces-config.xml ファイルに追加します (リスト 2 を参照)。このリスナーは、Seamを標準 JSF ライフ・サイクルに統合します (Seam がこのライフ・サイクルに織り込む機能拡張の概要については、図 3 を参照してください)。

リスト 2. Seam フェーズ・リスナーの構成

```
<lifecycle>
  <phase-listener>org.jboss.seam.jsf.SeamPhaseListener</phase-listener>
</lifecycle>
```

最後に、空の seam.properties ファイルをクラスパスのルートに配置して Seam にロードするよう命令します (リスト 3 を参照)。この空のファイルがJVM クラス・ローダー最適化として使用されて Seam がコンポーネントを検索するクラスパスの領域が絞り込まれるため、ロード時間が大幅に短縮されます。

リスト 3. Seam プロパティー・ファイル

```
# The mere presence of this file triggers Seam to load
# It can also be used to tune parameters on configurable Seam components
```

当然のことながら、この最小限のセットアップでは Seam が持つ多くの機能を使用できません。上記の手順は、Seam を基本的なレベルで使用する場合、Seamのフットプリントはほんのわずかであることを示すためのものです。例えば、Seam にはその機能の適用範囲を JSF ライフ・サイクルの枠を超えて拡張するサーブレット・フィルターが組み込まれています。このサーブレット・フィルターは、JSF以外の要求の統合、リダイレクトによる対話の伝搬、そしてファイルのアップロード管理を目的として使用できます。EJB3 統合をはじめとする追加機能を制御する構成ファイルについての説明は、「参考文献」で紹介している Seam 参考資料に記載されています。

Seam による機能強化

Seam による管理 Bean の開発は、一般的な JSF 構成プロセスと比べると驚くほど簡単です。Bean を JSF ライフ・サイクルに公開するには、たった1 つのアノテーション、`@Name` をクラス定義の上に配置すればいいだけです。あとは、Seam がコンポーネントの可視性とライフ・サイクルを制御してくれます。とりわけ素晴しいのは、このBean を faces-config.xml ファイルで定義する必要がないという点です。

リスト 4 には、`@Name` アノテーションの他にも `@DataModel`、`@DataModelSelection`、`@In`、`@Out`、および `@Factory` が記載されています。これらのアノテーションが、ビュー・テンプレートと Seam コンポーネントとの間の双方向の変数フローを可能にします。

Seam の専門用語では、このアクションは bidirectional injection (双方向インジェクション) を略して、bijection(バイジェクション) と呼ばれます。プロパティー・データがアウトジェクトされると、このデータは名前によって直接ビューに表示できるようになります。データがコンポーネントにインジェクトされるのは、再ポスト時、またはコンポーネントが初期化される際のいずれかです。後者はよく知られた制御の反転(IoC) パターンの実装で、委譲オブジェクト内での結合に使用できます。従来の IoC と Seam のバイジェクションとの大きな違いは、バイジェクションでは長期スコープ内のコンポーネントが短期スコープ内のコンポーネントへの参照を持つことができるという点です。このような接続が可能な理由は、Seamはコンテナーの起動時だけでなく、コン

ポーネントが呼び出されるたびに依存関係を解決するためです。バイジェクションは、ステートフルなコンポーネント開発の土台となります。

もちろん、リスト 4 の POJO Bean では Seam が動作する仕組みを表面的に表すのがやっとです。その他の Seam の実装方法については、この連載の今後の記事で説明していきます。

リスト 4. 一般的な Seam POJO Bean

```
@Name("addressManager")
public class AddressManagerBean {

    @DataModel
    private List<Address> addresses;

    @DataModelSelection
    @Out( required = false )
    private Address selectedAddress;

    @Factory( value = "addresses" )
    public void loadAddress() {
        // logic to load addresses into this.addresses
    }

    public String showDetail() {
        // no work needs to be done to prepare the selected address
        return "/address.jspx";
    }

    public String list() {
        return "/addresses.jspx";
    }
}
```

Spring のインジェクション

既存の Spring コンテナーが管理するサービス層オブジェクトの投資を有効に活用するには、関連ビジネス・ロジックを処理するすべての SpringBean を Seam コンポーネントにインジェクトする必要があります。それにはまず、Spring フレームワークに付属のカスタム変数リゾルバー(「参考文献」を参照) によって処理される、Spring と JSF の統合が構成済みであることを確認しなければなりません。このブリッジが用意できていれば、あとは`@In` Java 5 アノテーションに値のバインディング表現を組み合わせて Spring Bean の注入を受け取る Seam コンポーネントのプロパティーを指定するだけで、Springを Seam に統合できます (リスト 5 を参照)。ちなみに、Seam の今後のバージョンには Spring のカスタム名前空間が含まれる予定なので、値のバインディング表現は必要なくなります。

リスト 5. Spring Bean のインジェクション

```
@Name("addressManager")
public class AddressManagerBean {
    @In("#{addressService}")
    private AddressService addressService;
}
```

上記のセットアップ例では、軽量コンテナー (この例では Spring) を使用して構成されたステートレス・サービス層とデータ・アクセス (DAO)層を使用できます。EJB3 は必要ないため、デプロイメントのターゲットは、基本サーブレット・コンテナーであれば何にしても構いません。

Seam と JSF の実装を初めて目にしたところで、今度は JSF を使用する際に私が直面した問題と、その問題を Seam が解決する仕組みについて少々詳しく説明することにします。

JSF の復習

Seam が JSF にもたらすメリットを十分に認識するには、Web ベースのプログラミングでよく使用されている他の方法と JSF がどのように違うのかを理解しなければなりません。まず、JSFは従来のモデル・ビュー・コントローラー (MVC) アーキテクチャーを実装する Web フレームワークの 1 つですが、JSF を他と差別化しているのは、JSFが並外れて豊富なパターン実装を採り入れるという点です。JSF における MVC 実装は、Struts、WebWork、Spring MVC などのフレームワークで使用されている「PUSH型 MVC」として知られるモデル 2 に比べると、従来の GUI アプリケーションのスタイルにより近いものがあります。この、後に挙げたフレームワークはアクション・ベースとして分類される一方、JSFはコンポーネント・モデルをベースとする新しいフレームワーク群のメンバーです。

この違いは、アクション・ベースのフレームワークは「PUSH 型」モデルを使用し、コンポーネント・フレームワークは「PULL 型」モデルを使用することを考えると把握しやすくなります。コンポーネント・フレームワーク内のコントローラーは、ページの表示要求に対して(アクション・ベースのフレームワークのように) 積極的役割を果たすのではなく、要求ライフ・サイクルの目立たないところでビュー内からデータを提供するメソッドを呼び出します。さらに、ページ上の要素(コンポーネント) は、イベントが発生するとサーバー・サイド・オブジェクトからメソッド呼び出しをトリガーできるようなイベントにバインドされるため、同じビューの再表示、あるいは別のページへの遷移が行われることになります。したがって、コンポーネント・フレームワークはイベント駆動型としても分類できます。イベントとの通信を行う上で基礎として使用される要求・応答プロトコルは、コンポーネント・フレームワークでは抽象化によって取り払われます。

イベント駆動型の手法では、ビューのレンダリングを準備するために前もって単一のメソッドの中で済ませておかなければならない作業が減るというメリットがあります。コンポーネント・フレームワークでの呼び出しは、UIイベントあるいは解決された値のバインディング表現のいずれかによる直接的な結果だからです。

アプリケーションが半分ぐらいまでできてくると、通常はどのページでも関係のない多くの作業が必要になってきます。この情報のすべてを単一のアクションあるいは一続きのアクションで管理しようとすると、メンテナンスはたちまち悪夢と化します。その結果、開発者は自分でも気付かずに、コードをオブジェクト指向モデルからプロシージャー型プログラミングのスタイルにしてしまうことがよくあります。コンポーネント・フレームワークはそれとは対象的に、作業を分離し、より自然な形でオブジェクトの役割と責任を果たします。

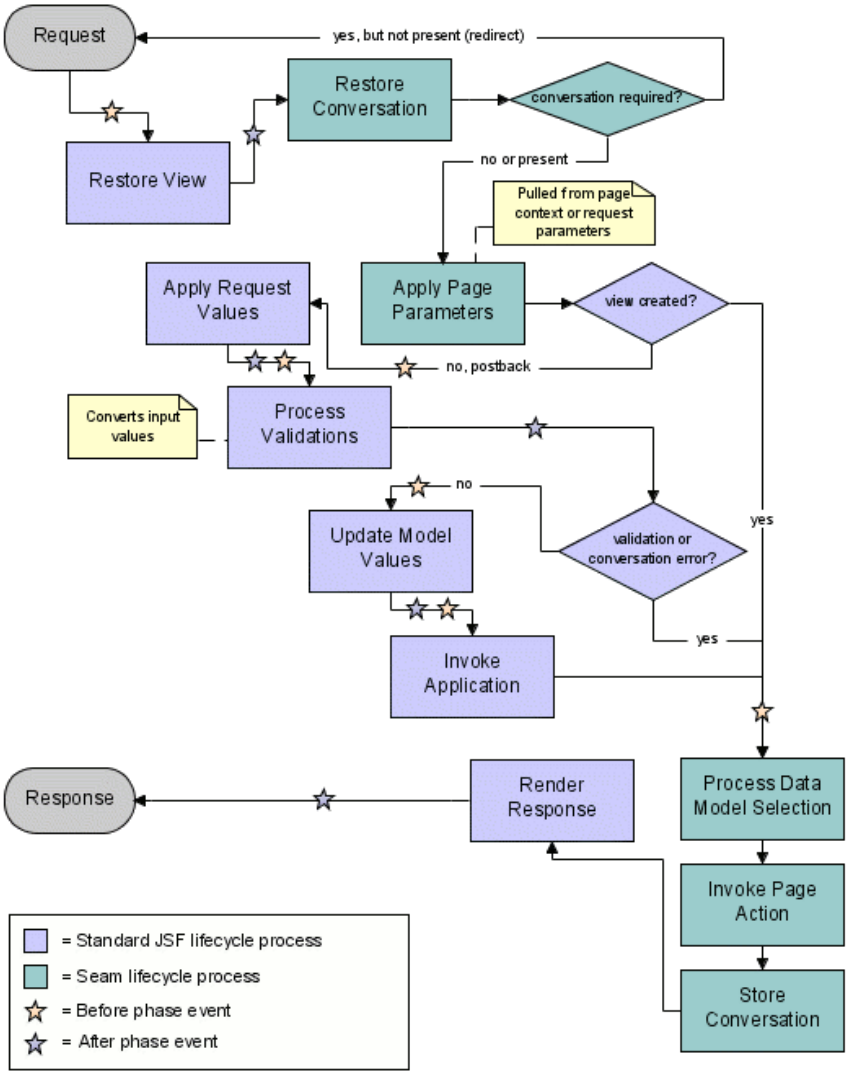
Seam と JSF

JSF とコンポーネント・フレームワークの基礎知識についての説明はこれで十分として、実際問題は、多くの Java 開発者が最近発見したように、JSFへの移行はまるで順風満帆というわけではないということです。アプリケーションにアクション・ベースの Web 世界を強要しようとしなければならなくなるという現実をはじめ、たいていの場合は、コンポーネント・モデルの採用にはまるで新しい一連の問題が伴います。JSFがアクション・ベースのフレームワークのように動作すればいいだけの場合もありますが、少なくともすべての要求に対して実行するフェーズ・リスナーを

使用するという手段に頼らなければ、これはネイティブJSF で実行可能なオプションではありません。

JSF の主要な欠点としては他にも、HTTP セッションへの大幅な依存 (特にひと続きのページに渡ってデータが伝播する場合)、いい加減な例外処理、ブックマーク・サポートの欠如、そしてXML Hell な構成などが挙げられます。Seam は JSF に自然に統合しつつも、JSF 仕様委員会が断念したか、あるいは単に見過ごしていた新しい機能を同時に織り込むことで、これらの問題の多くを解決します。Seamのフレームワークが促進する極めて簡潔で、読みやすさの点でも再使用可能性の点でも優れたコードには、これらの問題を管理する上で決まって漏れ込んでくる「結合」ロジックが一切ありません。図3 を見ると、アプリケーションのコードを簡潔にするのに役立つ、JSF ライフ・サイクルにおける Seam の拡張ポイントのほとんどがわかります。

図 3. Seam のライフ・サイクル拡張機能



ここからは、JSF 開発の一般的な課題に適用される、上記の拡張機能をいくつか取り上げて検討します。

手のかからない構成

Seam は、非常に実用的な Java 5 アノテーションの使用例となります。Seam のデプロイメント・スキャナーは、seam.propertiesファイルが含まれるすべてのアーカイブを調べ、`@Name` アノテーションでマークされたクラスに対して Seam コンポーネントを作成します。Java 言語にはコード・レベルでメタデータを追加するための共通構文がなかったため、多くのXML 構成が考案されましたが、それよりも優れたソリューションが登場することになったのは、Java 5 仕様へのアノテーションの追加がきっかけです。ほとんどのバッキングBean は特定アプリケーション内で使用するために開発されているため、これらの Bean の構成をクラス自体ではない別のファイルに「抽象化」する理由はありません。操作するファイルが1 つ減るのも付加的なメリットです。Seam は、Bean を JSF ライフ・サイクルに統合するのに役立つ、完全なアノテーション・カタログを提供します。リスト 4 では、これらのアノテーションをいくつか使用しています。

ページ・アクションと RESTful な URL

アクション・ベースのフレームワークではそれぞれの要求を前もって処理しますが、コンポーネント・フレームワークではこれを頼りにすることはできません。この代償によって影響されるユース・ケースは、ほんの少数を挙げるだけでもRESTful URL、ブックマーク・サポート、URL パターンによるセキュリティー、そしてページ・フロー検証などがあります。要求を前もって処理できないことは間違いなく、JSFの使用方法を学ぶ開発者にとって主な混乱の種の 1 つです。JSF ベンダーによっては、開発者ツールで onPageLoad 機能 (「参考文献」を参照) を提供することによって対処しているところもありますが、これはコア仕様の一部ではありません。

ユーザーが例えばブックマークから直接、項目の詳細画面を要求した場合に、一般的にどうなるかを考えてみてください。JSF コントローラーは受動的な動作をするため、ページのレンダリングが始まったが最後、ターゲット・データがないことが明らかになったとしてもユーザーを論理フローの先頭にリルートすることができなくなります。残された選択肢は、値が空白で、その他にもレンダリングの欠陥が考えられるページを表示することだけです。

まず直感的に思い付く方法は、ページの主要なバッキング Bean に「事前レンダリング」するメソッドを実装するというものでしょう。ただしコンポーネント・フレームワークでは、バッキングBean とページの関係は必ずしも 1 対 1 である必要はありません。それぞれのページはバッキング Bean の組み合わせに依存することが可能で、組み合わせに含まれるそれぞれのBean にしても、複数の異なるページで使用できます。その代わりに必要となるのは、単一のビュー ID (/user/detail.jspx など)を、それに対応するビュー・テンプレートがレンダリング用に選択されたときに呼び出す必要がある 1 つ以上のメソッドに関連付けるための何らかの手段です。フェーズ・リスナーを使用することもできますが、その場合には、現行のビューとフェーズに対して機能を実行するかどうかを決定するカスタム・ロジックが必要になります。このソリューションでは冗長なロジックが多くなるだけでなく、最終的にはコンパイルしたJava コードに (アプリケーションで最も不確定なコンポーネントになりがちな) ビュー ID をハードコーディングするはめになります。

救いの手となるページ・アクション

レンダリングの不備を発生前に阻止する手段となるのが、Seam のページ・アクションです。ページ・アクションを指定するには、ページが入力されるとRender Response (レンダリング応答) フェーズの直前に実行されるメソッド・バインディングを使用します。メソッド・バインディングは、/WEB-INF/pages.xml構成ファイルでどのビュー ID に対しても任意の数だけ構成できます

(あるいはページごとに定義を分割することもできます。その場合、ビュー・テンプレートに隣接するファイルに定義を配置し、ファイル名を複製してファイル拡張子を*.page.xml に置き換えます)。ページ・アクションに XML が適しているわけは、ビュー ID が変わりやすいためです。Apply RequestValues (要求値の適用) フェーズで JSF が値のバインディングによってポスト・データをモデル・オブジェクトにマッピングするのと同じく、Seamはページ・アクションの実行前に値のバインディングによって任意の要求パラメーターをモデル・オブジェクトにマッピングできます。こうした要求パラメーターのインジェクションの構成は、ページ・アクションのXML 宣言内にネストされます。ページ・アクションのメソッド呼び出しがヌル以外のストリング値を返すと、Seam はそれをナビゲーション・イベントとして扱います。そのため、本格的アクション・ベースのフレームワークに移行しなくても、その最も独特な機能をエミュレートすることが可能なのです。Seamには、アプリケーションで共通して使用される豊富な組み込みページ・アクションのセットが含まれています。このセットには、対話が確立されたことを検証するアクション、対話を開始、ネスト、終了できるアクション、発見された例外を処理するアクション、そしてクレデンシャルが正しいことを確認するアクションが含まれます。

ページ・アクションは、JSF に対するブックマーク・サポートを有効にするための重要な鍵です。Seam の作成者たちはこの機能を利用するため、魔法の要求パラメーター、`actionMethod` がページの入力時にメソッド呼び出しをトリガーできるようにしました。さらにいいことには、ブックマークのリンクを作成するために追加作業を行う必要はありません。Seamには、詳細を処理してくれる 2 つのコンポーネント・タグ、`s:link` と `s:button` があるからです。この 2 つのタグは、ネイティブ JSF の `h:commandLink` と `h:commandButton` にそれぞれ対応します。その違いは、Seam のコンポーネント・タグがアセンブルするリンクは、JSF を示す `HTTP POST` フォーム送信モデルを使用するのではなく、`HTTP GET` 操作を使用して要求を発行するという点です。したがって、Seam が作成するリンクはブックマークを作成するのに「わかりやすく」、開発者にとって遥かに使いやすくなるように考慮されています。

ページ・アクションを使用すると気付くかもしれませんが、ロケーション・バーの URL は次に控えているページではなく、現在表示中のページに対応します(URL が次のページに対応するという事態になるのは、JSF が構成するフォームは、そのフォームを生成した URL に再ポストされるからです。JSFはサーバー・サイドのリダイレクトによって新しいビューに進むため、アクションが行われた後にロケーション·バーが更新されて新しいビューを反映することはありません)。ページ・アクションの柔軟性を実演してみたいなら、ページ・アクションを使って、例えば/faces/product/show/10 という RESTful URL を作成してください。これを作成するには、ページ・アクション・メソッドをビューID「/product/show/*」にマッピングします。ここで、先頭の /faces は JSF サーブレットがマッピングする部分です。このページ・アクション・メソッドは要求URL を調べてデータ型とデータ識別子を判断し、該当するデータをロードしてから適切なテンプレートにナビゲートします。この例で明らかになるのは、JSF要求の URL とビュー・テンプレート間の 1 対 1 のマッピングは必須ではないということです。

ファクトリー・コンポーネント

JSF での最大のフラストレーションの 1 つとして挙げられるのは、ユーザーがトリガーしたアクションまたはアクション・リスナーのメソッド内は別として、JSFではビューのデータが作成される決定的な時点が指定されないという点です。アクション・メソッド内にロジックを配置しても、それがレンダリングの前に実行される保証はありません。ユーザーによってトリガーされたイベントが常にページ・ビューに先行するとは限らないためです。

例えば、JSF アプリケーションへの URL が最初に要求されたらどうなるでしょう。このページにサービス層から取得した一連のデータを表示しなければならないとすると、JSFライフ・サイクルでデータがフェッチされる見込みはありません。ビュー内の値のバインディング表現にマッピングするバッキング Bean のゲッター・メソッド内にロジックを付ければいいと考えるかもしれませんが、その場合、JSFがその表現を解決するたびに追加の呼び出しをトリガーし、それによってサービス層が検索されます。そうなると、ページ上には少数のコンポーネントしかないのに、ゲッター・メソッドが6 回以上呼び出されるという事態も考えられます。これでは明らかに、パフォーマンスの点で最適ではありません。管理 Bean 上で専用プロパティーを使って状態を維持するとしても、この事例に直面するたびに余分な接続を追加する必要が出てきます。そこで1 つのソリューションとなるのが、今まで説明してきたように、Seam のページ・アクションを使用することです。ただし、このタスクはあまりにも一般的なので、Seamではさらに簡単なソリューションを提供しています。

Seam では、ファクトリー・データ・プロバイダーという概念を導入しています。ファクトリー・データ・プロバイダーを指定するのは、`@Factory` Java 5アノテーションです。ファクトリーを構成する方法は何通りかありますが、最終的な結果として、データは最初に要求されたときに 1 度だけ作成されることになります。Seamは確実に、同じデータに対する以降の要求が検索メソッドの追加呼び出しをトリガーすることなく、以前に作成された結果セットを返すようにしています。このファクトリー・データ・プロバイダーを対話と組み合わせると、検索するのに手間がかかるデータの短期キャッシュを実装する上で非常に強力な機能になります。JSFが値のバインディング表現を解決する回数を減らそうとしないならば、Seam のファクトリー機能がたいていの場合は役に立ちます。

ステートフルな対話

JSF について一般的に混乱しがちな点は、その状態管理機能です。JSF 仕様では、ページがアクションを受け取った後にどのように「復元」されるか、そしてその間はタイム・イベントがキューに入れられて選択が登録されることを説明しています。仕様で使われている言葉を調べると、コンポーネント·ツリーが再ポスト時に復元される間、これらのコンポーネントが使用するバッキングBean データは復元されないことが分かります。コンポーネントは単に値のバインディング (`#{value}` 構文を使用した EL 式) をストリング・リテラルとして保管し、このストリング・リテラルは基礎となるデータに対して実行時に解決されます。つまり、値がページ・スコープや要求スコープなどの短期スコープに保管されると、JSFライフ・サイクルが Restore View (ビューの復元) フェーズに達するまでには値が消えてなくなるということです。

値のバインディング・データがコンポーネント・ツリーに保管されない場合に最も顕著なマイナス面は、ゴースト・イベント (ghost event)効果です (「参考文献」を参照)。これは、`UIData` ファミリーの一時親コンポーネントによって引き起こされます。値のバインディング表現によって参照されるモデル・データが使用できなくなったり、あるいはコンポーネント・ツリーが復元される前に変更されると、ツリーの一部が破棄される結果となる場合があります。イベントがこれらの破棄された分岐のいずれかに含まれるコンポーネントからトリガーされた場合、そのイベントは検出されず、その省略は極めてひそやかに行われます(キューの開発者が「どうして私の作ったアクションが呼び出されないんだ?!」と叫ぶ声は別として)。

イベントの損失は例外の条件となりそうですが、JSF ライフ・サイクルでは何の警告も促されません。これらのコンポーネントは基礎となるデータが安定して適切に復元されることに依存しているので、JSFが何かが欠けていることを認識するのは容易ではないのです。

残念ながら JSF 仕様そのものが、「重宝」なスコープとも呼べるセッション・スコープに大部分のバッキング Bean を配置するよう開発者を仕向けています。そうなると、サーバー管理者が「メモリー不足」エラー、クラスター環境のサーバー・アフィニティー、そしてサーバー再起動時のシリアライゼーション例外という形の副次的影響に対処せざるを得なくなります。MyFacesTomahawk プロジェクトでゴースト・イベントのソリューションとして提供しているのは、`t:saveState` タグです。この MyFaces タグでは、データ (バッキング Bean 全体を含む) を単なる値のバインディングとしてではなく、コンポーネント・ツリーの一部として保管できます。しかし、このソリューションはむしろ大ざっぱなもので、隠しフォーム・フィールドを使って要求間で値を転送するという方法とかなり似ています。さらに、ビューとコントローラーの間には密結合も作り出されます。Seamの作成者たちが認識したのは、Java Servlet 仕様の 3 つの組み込みコンテキスト (要求、セッション、アプリケーション) では、データ駆動型Web アプリケーションの十分なスコープ・セットは構成されないということです。そこで、彼らは Seam に対話スコープを導入しました。つまり、ページ・フローの開始点から終了点までを範囲とするステートフルなスコープです。

Seam の対話スコープ

「Seam ではステートフル・コンポーネントの使用を重視する」という Seam 参考ドキュメントでの記述は、Seam の中核となる構想を具体的に説明しています。Webアプリケーションは長いこと、ステートレスであると考えられていました。この考え方は、一部には HTTP プロトコルが持つステートレスな性質が原因となっているかもしれません。たいていのフレームワークは、ページのレンダリングで締めくくる前に1 回限りの処理を提供することによって、この考え方に従いますが、この手法はかなりの障害をもたらします。重要なアプリケーションは決まって、特定の事例を満足させるために長期で実行する対話を必要とするからです。ステートフルなコンテキストが必要なアプリケーションの例には、ストア・チェックアウト・プロセス、プロダクトのカスタマイズ、複数ページからなるフォームのウィザード、それに手順を踏んだ対話動作をベースにしたその他多くのアプリケーションがあります。これらのアプリケーションの一部は、URLパラメーター (RESTful な URL としても知られています) と隠しフィールドを使ってページ間のデータ転送を行うことで上手く切り抜けることができるのも確かですが、この方法は開発者にとってすぐに頭痛の種となります。また、現在では時代遅れでもあります。ほとんどのWeb フレームワークは依然としてステートレス・モデルで動作しているので、カスタム・ソリューションを「ハッキング (詳細調査)」するためにフレームワークの外へ踏み出すことも珍しくありません。

JSF では、HTTP セッションに大幅に依存してステートフルなコンテキストを導入しようとします。実際、JSF コンポーネントの動作は、セッション・スコープを持つバッキングBean を操作すると、期待通りに動作すると言えるほど改善されます。ただし慎重に設計しないで HTTP セッションを過剰に使用すると、深刻なメモリー・リークやパフォーマンスのボトルネック、それにセキュリティー問題の原因となります。その上、タブ切り替え型ブラウザー環境でHTTP セッションを使用すると極めて異常な動作の原因となり、ユーザーの神聖なる Back ボタンの動作に問題を生じることもあります。ただしここで重要な点は、JSFは必要の半分しか満たさないということです。JSF は、ステートフルな UI であり、コンポーネント・ツリーの保管と復元の詳細すべてを処理しますが、データの保管と復元に関する支援はまったく行いません。この契約では、JSFがス

テートフルな UI を提供し、開発者がステートフルなデータを提供することになります。そして運悪く、この 2 つが一致することを確実にするという役目は開発者に課せられます。

Seam が登場する以前は、ステートフルなデータを持つ方法として唯一役に立ったのは、HTTP セッションに頼ることでした。Seam はこの事態を是正し、JSFの状態管理を完全なものにするために、まったく新しい対話スコープを確立しています。JSF ライフ・サイクルに Seam が加わることにより、対話のコンテキストは、それぞれの要求で送信されるトークンによって識別される単一のブラウザー・ウィンドウ(またはタブ) に結合されます。対話スコープは、HTTP セッションの分離したセグメントを使ってページ間でデータを転送します。ここで踏まえておかなければならないのは、Seamの対話パーシスタンスに対する HTTP セッションの使用は完全に透過的だという点です。Seam は不用意にコンポーネントを HTTP セッションにダンプして、いつまでもそこに残しておくことはしません。セッション・データの該当セグメントは、Seamによってそのライフ・サイクルが慎重に管理され、対話が終了すると自動的にクリーンアップされます。Seam は「Web 対話」のそれぞれのページでデータが出入りできるように、新しい宣言型の方法によって巧みにバイジェクションを使用します。このように、Seamの対話スコープは HTTP セッションの限界を克服すると同時に、開発者が HTTP セッションを使用しなくても済むようにします。

例外の処理

Seam の作成者によると、「JSF は例外処理に関しては驚くほど制限されている」ということですが、この点に関する異議はまったくありません。JSF仕様は例外管理を完全に無視していて、その責任をすべてサーブレット・コンテナーに押し付けています。しかし、サーブレット・コンテナーに例外を処理させると、エラー・ページ上に表示できる内容が大幅に制限され、トランザクションのロールバックが不可能同然になるという問題があります。エラー・ページは要求ディスパッチャーの転送後に表示されるため、`FacesContext` はもはやスコープ内にはありません。そのため、ビジネス・ロジックを行うには手遅れです。最後の望みとしては、Servlet API から `javax.servlet.error.*` 要求属性を取得し、エラーの原因を示す情報を検索するしかありません。

ここでも、Seam は的確な宣言型の例外処理を携えて救いの手を差し伸べます。例外管理は、アノテーションまたは構成ファイルのいずれかによって指定されます。アノテーションの場合、`@HttpError`、`@Redirect`、そして `@ApplicationException` を例外クラスの上に配置して、これらの例外クラスがスローされた場合に行うアクションを指定します。一方の XML 構成オプションは、変更を目的としてアクセスすることができない例外クラスに対して使用できます。Seam での例外管理の役目は、HTTP ステータス・コードの送信、リダイレクトの実行、ページ・レンダリングの強制実行、対話の終了、例外が発生したときのユーザー・メッセージのカスタマイズです。JSFは応答のレンダリングが開始されてからはアクションの方針を変更できないので、いくつかの固有の制限が、例外を処理できるかどうかを決定します。ページ・アクションなどの他の Seam 機能を適切に使用することで、応答をレンダリングする前に、すべてではないにしてもほとんどの例外状態に対処できます。

Ajax remoting

最終的な JSF 仕様がリリースされたのは Ajax の人気が爆発した時期とほぼ重なっていたため、JSF フレームワークでは非同期 JavaScriptと部分的ページ・レンダリングの分野ではほとんど支援を提供していません。そのため、2 つのプログラミング・スタイルが互いに調和しないように見え

た時期もありました。結局は、JSFの `PhaseListeners` あるいは `Renderer` コンポーネントを使って部分的なページ更新を処理することを提案したソリューションが広まり始めましたが、それでも早速明白になったのは、JSF がAjax の採用を必要以上に困難にしていることです。ICEfaces などの一部のプロジェクトは、JSF ライフ・サイクルを、(Direct-to-DOMレンダリングとして知られる) ページとサーバー間の通信により適切に設計されたものに完全に置き換えるという手段を採るまでに至りました。

Seam では、JavaScript リモーティング (たいていの場合は Ajax の傘下にグループ分けされる技術) に対して独特の手法を提供しています。これは、DWR(Direct Web Remoting) ライブラリーの手法と大体似かよった手法で、JavaScript にサーバー・コンポーネントで直接メソッドを呼び出せるようにすることによって、直接クライアントとサーバーを結合します。ただし、DWRでは分離したエンドポイントにしかアクセスできない一方、Seam のリモーティングは豊富なコンテキスト依存コンポーネント・モデルにアクセスできるという点でDWR よりも強力です。この相互作用は JSF のイベント駆動型設計に基づくため、Swing パラダイムに一層近くなります。最大の利点は、この機能には追加の開発作業がほとんど必要ないことです。コンポーネントのメソッドに `@WebRemote`という単純なアノテーションを配置すれば、JavaScript にアクセスできるようになります。サーバー・コンポーネントの状態が変更されたら、今度はAjax4JSF コンポーネント・ライブラリーが部分的レンダリングを処理できます。要するに、Seam のリモーティング・ライブラリーによって、JSFの作成者たちが最初から心に描いていた JSF での対話型設計が実現するというわけです。

まとめ

連載「シームレスな JSF」の第 1 回をここまで読めば、Seam を使わずに JSF で開発するのは不合理であるとう意見がこじつけだとは思えなくなるはずです。これをさらに裏付けるには、JSR299, Web Beans の投票結果を見るだけで十分でしょう (「参考文献」を参照)。近い将来のある時点で Seam が正式な標準になり、Java EE スタックが最終的に「Web ベース・アプリケーションを対象とした極めて簡易化されたプログラミング・モデル」を提供することは明らかです。これはJSF 開発者にとっても Seam にとっても朗報ですが、Java 標準にするべきだという主張がないとしても、Seam は JSF を補完するものとして価値があります。

JSF の欠点を埋めるために Seam でまず必要となるのは、ほんのわずかなセットアップ作業だけです。そうしたわずかな作業だけで、Seam はJSF 開発の最も厄介な難問の数々を解決します。作業を上回るメリットがもたらされるのです。この記事で説明した Seam の利点は、その良さを理解する手始めでしかありません。

ダウンロード

内容	ファイル名	サイズ
Barebones Seam project[1]	j-seam1.zip	13KB

注

1. このプロジェクト概要では、Maven 2 のビルド・インフラストラクチャーを使用しています。すべての依存関係は、ビルド実行時にオンデマンドで取り込まれます。

著者について

Dan Allen

現在、CodeRyte, Inc. のシニア Java エンジニアとして活躍する Dan Allen は、熱烈なオープン・ソース擁護者でもあり、ペンギンを目にすると決まって興奮してしまいます。Linux とオープン・ソース・ソフトウェアの世界に魅了されたのは、材料工学の学位で Cornell University を卒業した後です。それ以来、Web アプリケーションの虜になっていますが、この数年はとくに、Spring、Hibernate、Maven 2、そして豊富な JSF スタックなどの Java 関連の技術に重点を置いています。彼の開発経験を辿るには、http://www.mojavelinux.com でブログにサブスクライブしてください。