Web アプリケーション フレームワーク 開発ガイド

Sun Java™ Studio Enterprise 7 2004Q4

Sun Microsystems, Inc.
www.sun.com

Part No. 819-1292-10
2004 年 12 月, Revision A

原典:　*Web Application Framework Developer's Guide*
Part No: 819-0728-10
Revision A

目次

はじめに

このガイドでは、Web アプリケーションフレームワークを紹介し、その内容と仕組み、他の Web アプリケーションフレームワークとの相違点を説明します。

お読みになる前に

このマニュアルを読み始める前に、サーブレットや JavaServlet™ ページ (JSP ページ) などの既存の J2EE Web テクノロジを利用した Web アプリケーションの構築で用いられている概念を理解しておくことを推奨します。

詳しい情報は、以下のリソースから得ることができます。

- Java 2 Platform, Enterprise Edition Specification
  `http://java.sun.com/j2ee/download.html#platformspec`
- J2EE Tutorial
  `http://java.sun.com/j2ee/tutorial`
- Java Servlet Specification バージョン 2.3
  `http://java.sun.com/products/servlet/download.html#specs`
- JavaServer Pages Specification バージョン 1.2
  `http://java.sun.com/products/jsp/download.html#specs`

注 – Sun では、本マニュアルに掲載されている第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関しても一切の責任を負いません。Sun は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスの利用あるいはそれらのものを信頼することによって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

マニュアルの構成

第 1 章「概要とアーキテクチャ」では、Web アプリケーションフレームワークアーキテクチャの概要を説明し、さまざまな部品を組み合わせて Web アプリケーションフレームワークアプリケーションを作成する方法を示します。

第 2 章「アプリケーションの開発」では、後で完全に機能するアプリケーションに組み立てることが可能なアプリケーションコンポーネントの作成および使用方法について詳しく説明します。

第 3 章「プログラミングガイド」では、一般的なプログラミングシナリオを示し、Web アプリケーションフレームワークのいくつかの基本的なオブジェクトの使い方を説明します。

第 4 章「アプリケーションの配備」では、ほとんどの J2EE コンテナでの配備用に Web アプリケーションフレームワークアプリケーションを準備する方法と、Web アプリケーションフレームワークアプリケーションの配備時構成について説明します。

付録 A「障害追跡」では、既知の障害追跡について概説し、それぞれの既知の問題の状態、考えられる原因、考えられる解決策、およびコメントを示します。

書体と記号について

書体または記号*	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例。	.login ファイルを編集します。 ls -a を実行します。 % You have mail.
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表します。	% **su** Password:
AaBbCc123 または**ゴシック**	コマンド行の可変部分。実際の名前や値と置き換えてください。	rm *filename* と入力します。 rm **ファイル名** と入力します。
『 』	参照する書名を示します。	『Solaris ユーザーマニュアル』
「 」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照。 この操作ができるのは「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅をこえる場合に、継続を示します。	% **grep '^#define ** **XV_VERSION_STRING'**

* 使用しているブラウザにより、これら設定と異なって表示される場合があります。

関連マニュアル

Java Studio Enterprise のマニュアルとしては、Acrobat Reader (PDF) 形式のマニュアル、チュートリアルと、HTML 形式のリリースノート、オンラインヘルプ、チュートリアルが提供されています。

オンラインで入手可能なマニュアル

ここで紹介しているマニュアルは、docs.sun.com℠ Web サイト、および Sun Java Studio Enterprise Developers Source ポータルサイト (http://developers.sun.com/jsenterprise) のドキュメントリンクから入手できます。

docs.sun.com Web サイト (http://docs.sun.com) では、インターネットで Sun のマニュアルを参照、印刷、購入することができます。

- 『Sun Java Studio Enterprise 7 2004Q4 リリースノート』- Part No. 819-1302-10

  最新のリリースの変更点や技術的な注意事項を説明しています。

- 『Sun Java Studio Enterprise 7 インストールガイド』 (PDF 形式) - Part No. 819-1300-10

  サポートしている各プラットフォームへの Sun Java Studio Enterprise 7 統合開発環境 (IDE) のインストール方法を説明しています。システム要件やアップグレード方法、サーバー情報、コマンド行スイッチ、インストールされるサブディレクトリ、データベースの統合、アップデートセンターの使用方法などの関連情報も記載されています。

- 『J2EE アプリケーションのプログラミング』- Part No. 819-1298-10

  EJB モジュールや Web モジュールを J2EE にアセンブルする方法を説明しています。また、J2EE アプリケーションの配備や実行についても説明しています。

- Web アプリケーションフレームワークのマニュアル (PDF 形式)
  - 『Web アプリケーションフレームワーク コンポーネント作成ガイド』 - Part No. 819-1284-10

    Web アプリケーションフレームワークのコンポーネントアーキテクチャと新しいコンポーネントの設計、作成、配布工程を説明しています。

  - 『Web アプリケーションフレームワーク コンポーネントリファレンスガイド』 - Part No. 819-1286-10

    Web アプリケーションフレームワークライブラリに提供されているコンポーネントを説明しています。

  - 『Web アプリケーションフレームワーク 概要』- Part No. 819-1288-10

    Web アプリケーションフレームワークとその位置づけ、仕組み、他のアプリケーションフレームワークと異なる点を説明しています。

  - 『Web アプリケーションフレームワーク チュートリアル』 - Part No. 819-1290-10

    Web アプリケーションフレームワークを使用して Web アプリケーションを構築する際の仕組みとその手法を紹介しています。

- 『Web アプリケーションフレームワーク 開発ガイド』- Part No. 819-1292-10

  Web アプリケーションフレームワークを使用し、開発するアプリケーションの構成要素として使用可能なアプリケーションコンポーネントの作成および使用の手順と、そのアプリケーションを大部分の J2EE コンテナに配備する方法を説明しています。

- 『Web アプリケーションフレームワーク IDE ガイド』- Part No. 819-1294-10

  Sun Java Studio Enterprise 7 2004Q4 IDE の各部の概要、および Web アプリケーションフレームワークアプリケーションを開発するためのビジュアルツールの使用方法を重点的に説明しています。

- 『Web アプリケーションフレームワーク タグライブラリリファレンス』- Part No. 819-1296-10

  Web アプリケーションフレームワークのタグライブラリを簡単に紹介し、タグライブラリに提供されているタグに対する包括的な参照を示しています。

チュートリアル

Sun Java Studio Enterprise 7 には、IDE の機能を理解する手助けとなるチュートリアルがいくつか用意されています。これらのチュートリアルにある技術、およびコード例は、そのまま、または編集を加えて、実際のアプリケーションの開発に利用することができます。すべてのチュートリアルで、Sun Java System Application Server への配備例が紹介されています。

チュートリアルは、すべて Developers Source ポータルのリンク「Tutorials & Code Camps」から利用可能です。IDE で「ヘルプ」>「コードサンプルとチュートリアル」>「概要」を選択すると、このサイトにアクセスできます。

- 「クイックスタートガイド」は、Sun Java Studio IDE の紹介をしています。チュートリアルは、Sun Java Studio を初めてご使用になる方や、特定の機能について早く知りたい場合は、このガイドから始めてください。これらのチュートリアルは、単純な Web アプリケーションや J2EE アプリケーションの開発方法、Web サービスの生成方法を説明しています。また、UML モデリング、リファクタリングの導入方法についても説明しています。ガイドを終えるための所要時間は数分です。
- 「チュートリアル」は、Sun Java Studio IDE の特定の 1 つの機能に焦点を当てています。ある機能の詳細に関心がある場合は、これらを実行してみてください。例で説明している機能によって、初めからアプリケーションを構築する場合と、提供されたソースファイルを使用して構築する場合があります。チュートリアルは 1 時間以内で完成できます。
- 「概要ビデオ」は、技術の説明がビデオで提供されています。IDE の視覚的な概要や、特定の機能の詳細説明を見ることができます。概要ビデオにかかる時間は数分です。概要ビデオは、任意の個所で開始、終了することもできます。

オンラインヘルプ

Sun Java Studio Enterprise 7 IDE には、オンラインヘルプが用意されています。ヘルプキー (Microsoft Windows 環境では F1 キー、Solaris オペレーティング環境では Help キー) を押すか、「ヘルプ」>「ヘルプ (すべて)」を選択して開くことができます。ヘルプの項目と検索機能が表示されます。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは以下の表に示す場所から参照することができます。

マニュアルの種類	アクセシブルな形式と格納場所
マニュアルとチュートリアル	形式: HTML 場所: `http://docs.sun.com`
チュートリアル	形式: HTML 場所: Developers Source ポータル (`http://developers.sun.com/jsenterprise`) のリンク「Examples & Code Camps」
リリースノート	形式: HTML 場所: `http://docs.sun.com`

第1章

概要とアーキテクチャ

この章では、Web アプリケーションフレームワークアーキテクチャの概要を説明し、さまざまな部品を組み合わせて Web アプリケーションフレームワークアプリケーションを作成する方法を示します。

Web アプリケーションフレームワークの基礎となるテクノロジは、JATO と呼ばれます。JATO という用語は、このマニュアルでときどき参照することがあり、特にクラスや他のプログラム名で使用されます。Web アプリケーションフレームワークと JATO という名前は同等と考えてください。

概要

Web アプリケーションフレームワークとは

Web アプリケーションフレームワークは、エンタープライズ Web アプリケーション開発用に作られた、標準に基づく強力で成熟した J2EE Web アプリケーションフレームワークです。Web アプリケーションフレームワークでは、表示フィールドやアプリケーションイベント、コンポーネント階層、ページ主体の開発手法などの既成の概念と、Model-View-Controller および Service-to-Workers パターンに基づく最新技術を結集した設計手法とが一体化されています。

Web アプリケーションフレームワークが対象とする開発者

Web アプリケーションフレームワークは、主として、中大規模および巨大規模の Web アプリケーションを構築する J2EE 開発者のニーズに応えることを意図しています。Web アプリケーションフレームワークは小規模の Web アプリケーションにも使用でき、実際そのように利用されていますが、その主な利点は、小規模ではそれほどすぐに明らかになりません。Web アプリケーションフレームワークは、アプリケーションを長期間維持して、多数の変更を加え、適用範囲を拡張する場合、特にその能力を発揮します。簡単に言えば、Web アプリケーションフレームワークは、エンタープライズアプリケーションの開発支援の面で卓越しています。

Web アプリケーションフレームワークは、再利用可能なコンポーネントのコア機能を提供するため、簡単に Web アプリケーションに組み込むことが可能な、既成のコンポーネントを提供することを望む開発者に適しています。さらに、Web アプリケーションフレームワークは、垂直型 Web 製品を構築するためのプラットフォームとして非常に適しています。特に、そうした拡張性能によって、ユーザーとオリジナルの開発者は、既存の垂直型機能を拡張したり、活用したりするための明確な方法を手にすることができます。

Web アプリケーションフレームワークができること

Web アプリケーションフレームワークは、最新技術を結集した J2EE 設計パターンを利用したエンタープライズ Web アプリケーションを構築する開発者を支援します。Web アプリケーションフレームワークは設計パターンに基づく骨組みを提供し、企業の設計者は自身のアーキテクチャの他の部分をはめ込むことができます。Web アプリケーション開発者は簡単な開発手法を見出し、企業設計者は綿密に定義された方法で他のエンタープライズ層およびエンタープライズコンポーネントと統合する、明確に記述された設計手法を見出します。

Web アプリケーションフレームワークは、高低両レベルのインフラストラクチャおよび設計パターンを提供することによって開発者が再利用可能なコンポーネントを構築する支援をします。開発者が定義したコンポーネントは、ネーティブコンポーネントであるかのように Web アプリケーションフレームワークと対話するオブジェクトです。コンポーネントは自由に組み合わせて、1 つのアプリケーション全体、異なるアプリケーション間、さらには複数のプロジェクトや企業にわたって再利用できます。

まとめとして、Web アプリケーションフレームワークは、J2EE 開発初心者が Web アプリケーション開発に初めて取り組むのに役立ちます。また、上級の J2EE 開発者は、他のフレームワークで実現不可能な高度な機能を開発するための強力なツールキットを利用することによって、その能力を高めることができます。

Web アプリケーションフレームワークができないこと

Web アプリケーションフレームワークはエンタープライズ層のフレームワークではありません。このことは、EJB や Web サービス、その他の種類のエンタープライズリソースの作成を直接支援するわけではないことを意味します。Web アプリケーションフレームワークはエンタープライズアプリケーション開発向けに作られていますが、正しくはエンタープライズ層リソースのクライアントであり、それらリソースを利用するための形式的な仕組みを提供します。

注 – Web アプリケーションフレームワークのより広範な概要については、『Web アプリケーションフレームワーク 概要』を参照してください。

Web アプリケーションフレームワークアーキテクチャの 3 つの層

Web アプリケーションフレームワークでは、その主要アーキテクチャ基盤の 1 つとして、実績のある MVC (Model-View-Controller) パターンを使用しています。元の MVC パターンは、Smalltalk によるステートフルなクライアント側アプリケーションの作成を支援するために開発されました。このパターンは、J2EE Web 層で主としてステートレスに使用できるように適合しています。Web アプリケーションフレームワークの MVC パターンは、本格的で完全なパターンです。このため Web アプリケーションフレームワークは、MVC への準拠とされていながら実際には相当の部分でそのパターンの一部しか使用していない他の Web 層フレームワークとは大きく異なっています。

以下に、このマニュアルで層と呼ぶ MVC アーキテクチャの 3 つの部分と、Web アプリケーションフレームワークにこれらの層がどのように組み込まれているかを簡単に説明します。

モデル層

MVC におけるモデルとは、プレゼンテーションに中立なデータの調停者です。このデータは、所定の表現をサポートするように調整することも、アプリケーション固有のデータ構造を表すように調整することもできます。

Web アプリケーションフレームワークでは、データの表現方法への依存を最小限に抑えながら、アプリケーションデータを表すモデルを作成する方向を採用しています。その代わりに、データの形式に最適なビューが選択されます。

モデルの種類

Web アプリケーションフレームワークには、以下のような多数のモデルの種類があります。

モデルの種類	説明
Model (モデル)	最も一般的な種類のモデル。すべてのモデルコンポーネントは、最終的にこの最低限のインタフェースを実装する必要があります。このインタフェースは、フィールド値の取得および設定機能という最も基本的なモデルの動作を指定します。
ContextualModel (コンテキストモデル)	名前付きコンテキストにデータを保持できる種類のモデル。コンテキストの定義は、モデル固有のものになります。
DatasetModel (データセットモデル)	明確に区別されるデータ行を含む種類のモデル。DatasetModel は、反復子 (iterator) のようなメソッドによって特定の行に位置付けることができます。
MultiDatasetModel (マルチデータセットモデル)	個別化された DatasetModel で、ContextualModel と同様、この種類を使用すると、名前付きデータセットにアクセスできます。
ExecutingModel (実行モデル)	バッキングストアのデータを取得または更新するために実行できる種類のモデル。
RetrievingModel (検出モデル)	特にデータ取得操作をサポートする ExecutingModel のサブタイプ。(この種類は SQL に関係しているように見えますが、SQL 固有ではありません。)
InsertingModel (挿入モデル)	特にデータ挿入操作をサポートする ExecutingModel のサブタイプ。(この種類は SQL に関係しているように見えますが、SQL 固有ではありません。)
UpdatingModel (更新モデル)	特にデータ更新操作をサポートする ExecutingModel のサブタイプ。(この種類は SQL に関係しているように見えますが、SQL 固有ではありません。)
DeletingModel (削除モデル)	特にデータ削除操作をサポートする ExecutingModel のサブタイプ。(この種類は SQL に関係しているように見えますが、SQL 固有ではありません。)
TreeModel (ツリーモデル)	デカルト構造ではなく汎用的な階層構造にデータを格納するモデル。このデータ構造に対する反復処理を提供します。
BeanAdapterModel (Bean アダプタモデル)	バッキングデータストアとして 1 つ以上の JavaBean を使用するモデル。
ObjectAdapterModel (オブジェクトアダプタモデル)	バッキングデータストアとして任意の Java オブジェクトグラフを使用するモデル。
QueryModel (照会モデル)	SQL 文を使い、バッキングデータストアとして RDBMS を使用する JDBC ベースのモデル。

モデルの種類	説明
StoredProcModel (ストアドプロシージャモデル)	データの取得と設定に RDBMS ストアドプロシージャを使用する JDBC ベースのモデル。
WebServiceModel (Web サービスモデル)	データの取得と設定に JAX-RPC 経由の Web サービスを使用する、個別化された ObjectAdapterModel。
CustomModel (カスタムモデル)	開発者独自のデータ格納メカニズムを実装できる抽象モデルの種類。
CustomTreeModel (カスタムツリーモデル)	開発者が階層データ用に独自の記憶領域を実装できる抽象モデルの種類。
SimpleCustomModel (カスタム単純モデル)	すぐに使用できる格納場所とデータセット機能を提供する具体モデル。

ビュー層

MVC のビューとは、モデルのデータを表示する、プレゼンテーションに固有の手段です。モデルとビューの間には、モデルの変更はそれに接続されているすべてのビューに自動的に反映され、ビューに表示されたデータの変更は関連付けられたモデルに自動的に反映される、という関係があります。

Web アプリケーションフレームワークでは、ビューはモデルデータを表すものと基本的に定義されます。

Swing、Visual Basic、Delphi、または Powerbuilder でのクライアント側アプリケーションの作成に精通している場合は、Web アプリケーションフレームワークのビューという概念を簡単に理解できるでしょう。これらの環境では、開発者は子コンポーネントを含むフレーム、ウィンドウ、およびダイアログを作成します。これらの子コンポーネントは、他の GUI ウィジェットの GUI ウィジェットまたはコンテナで、任意のレベルに入れ子にできます。

Web アプリケーションフレームワークビューコンポーネントは、ほとんど同じようなものを示します。クライアント側アプリケーションに、表示フィールド、パネル、ツリー、および複雑なサブコンポーネントなど、さまざまな種類の GUI ウィジェットがあるのと同様に、これらと同じ役割を担うさまざまな種類の Web アプリケーションフレームワークビューコンポーネントが存在します。Web アプリケーションフレームワークには、表示フィールドとして動作する個別化されたビュー、他のビューを含むことが可能なコンテナ、および複雑なビューコンポーネントとして動作できるこれらの組み合わせが装備されています。

ビューの種類

Web アプリケーションフレームワークビューであるオブジェクトは、実際には `com.iplanet.jato.view.View` インタフェースまたはその派生インタフェースの 1 つを実装するオブジェクトです。

以下の表に、Web アプリケーションフレームワークで利用できる主なビューの種類を示します。

名前	説明
View (ビュー)	最も一般的な種類のビュー。すべてのビューコンポーネントは、最終的にこの最小インタフェースを実装する必要があります。このインタフェースは、特定のビュー動作を指定しません。したがって、View インタフェースには、有用なインスタンスはまずありません。
ContainerView (コンテナビュー)	他のビューを含むことができる種類のビュー。このインタフェースは、子ビューコンポーネントの管理に固有のメソッドを追加します。
TiledView (タイルビュー)	その子ビューコンポーネントを多数の繰り返しタイル (繰り返し領域) に表示できる特殊な種類のビュー。タイルの例としては、表の行や列、タブ付きコンポーネントのタブがあげられます。作成されるタイルのレイアウトは想定されません。このインタフェースでは、単にタイルの繰り返しという概念がエンコーディングされています。
TreeView (ツリービュー)	ツリー形式で情報を表示するのに役立つ種類の ContainerView。
(ビュー Bean)	ビューコンポーネント階層の最上位、つまりルートとして機能できる、個別化された ContainerView。
DisplayField (表示フィールド)	関連付けられた値を持つ特殊なビュー。DisplayField の値はページ上に表示することができ、通常はユーザーによりアプリケーションに送付されます。DisplayField には、BooleanDisplayField、ChoiceDisplayField、および CommandField といったいくつかの特殊な種類があります。
CommandField (コマンドフィールド)	生成される応答内でユーザーまたはユーザーエージェントが起動可能な要素を表す特殊な DisplayField。たとえば HTML では、ボタンやリンクは CommandField によってサーバー側 Web アプリケーションフレームワークアプリケーションで表されます。CommandField が起動されると、新しい要求がサーバーに送信され、その要求に応答してサーバー上で対応するイベントまたはオブジェクトが起動します。

ページと ViewBean

前述したように Web アプリケーションフレームワークには、利用可能な既存の J2EE 規格とテクノロジが有効に採用されています。Web アプリケーションフレームワークで有用な J2EE テクノロジの 1 つが、JSP (Java Server Page) テクノロジです。JSP はアプリケーション開発者が使用するのに最適で、これを使うとすばやく簡単に作成と変更を行うことができます。

Web アプリケーションフレームワークでは、アプリケーション開発者がアプリケーションのページを作成するための主たる方法として JSP が採用されています。ただし、既存の JSP テクノロジベースと Web アプリケーションフレームワークの間には、何らかのつながりがなければなりません。この役割は ViewBean が果たします。ViewBean はある意味で、JSP テクノロジと Web アプリケーションフレームワークテクノロジの接点と言えます。

以下の項では、ViewBean と JSP および Web アプリケーションフレームワークの関係を説明します。

ViewBean とその JSP との関係

JSP を使って J2EE Web アプリケーションを作成したことがある場合は、一般にヘルパー Bean と呼ばれているものを使って JSP の複雑さに対処した経験があるでしょう。ヘルパー Bean は通常、JSP 内では保守とデバッグが困難な Java コードと複雑なデータ構造を JSP から排除する目的で JSP と組み合わせて使用されます。これによって、こうした複雑な機能を JSP 内ではなくヘルパー Bean に置きます。ヘルパー Bean は、*<jsp:useBean>* タグを介して JSP と関連付け、JSP により使用されるようにします。

JSP は本来保守とデバッグが困難であるため、同様に Web アプリケーションフレームワークアプリケーション内でも JSP の複雑さを避けることが望まれます。このために Web アプリケーションフレームワークではヘルパー Bean パターンが採用されており、有用で生産性の高いフレームワークを実現するためのより高度な機能セットをサポートするように拡張されています。

具体的には、Web アプリケーションフレームワークは ViewBean を使って JSP の複雑さに対処し、アプリケーション開発者がアプリケーション指向の Java コードを挿入する場所を提供します。ViewBean は JSP ページの範囲に配置され、他のヘルパー Bean とまったく同じように利用できるため、ViewBean は他のヘルパー Bean と同様に従来の Bean 指向のテクニックを使って利用できます。ただし、ViewBean は、ヘルパー Bean とまったく同じではありません。

最も大きな違いは、ViewBean のほとんどの使用先 (例 : Web アプリケーションフレームワーク JSP タグ) は、Bean のようなプロパティを使ってアプリケーションデータにアクセスしない、という点です。Web アプリケーションフレームワークタグは、この代わりに Web アプリケーションフレームワーク固有のビュー API を使って、ビューの子コンポーネントの階層と対話します。このアプローチによって、より充実した対話が実現することに加え、はるかに高いパフォーマンスも実現します。これは、大規模なレベルに拡張することが必要とされる企業アプリケーションには、明らかな利点となります。

ViewBean とそのビューとの関係

ViewBean は JSP ヘルパー Bean と同様の役割を果たすことに加え、Web アプリケーションフレームワークビューコンポーネント階層のルートビューとして機能する特殊なビューコンポーネントでもあります。つまりこれらは、他のビューコンポーネントを含む最上位のビューコンポーネントであり、したがって親は持ちません。Swing などのクライアント側開発テクノロジでは、ViewBean はウィンドウ、フレーム、またはダイアログコンポーネントのように機能します。

ViewBean インタフェースは、ContainerView インタフェースから拡張されます。つまり ViewBean には、ハードディスク上のルートディレクトリに他のディレクトリが含まれるように、他のビューコンポーネントを入れ子にして含めることができます。ViewBean インタフェースには、標準 ContainerView の全動作に加え、ビューコンポーネント階層のルートとしてのその役割に固有のメソッドが追加されています。

こうした属性があるため、ViewBean は、主としてアプリケーションのページと考えることができます。クライアント側アプリケーションが多数のウィンドウやダイアログから構成されているのと同様に、Web ベースのアプリケーションは多数のページから構成され、これらの各ページには ViewBean が関連付けられています。アプリケーション内の Web アプリケーションフレームワークページの数は、ViewBean オブジェクトの数と同じになります。

以下のような理由から、ViewBean は標準 ContainerView と比べて特別なものになっています。

- ViewBean は、(*<jato:useViewBean>* タグを介して) JSP と直接関連付けることができる唯一のビューコンポーネントです。
- ViewBean はアプリケーションの 1 つのページを抽象化したものであるため、ページ固有のアプリケーション値の格納といったページ固有の機能を処理するロジックとメソッドを備えています。
- ViewBean は、アプリケーション開発者がクライアントに対する応答の生成を開始するためのメソッドを提示します (これらは要求のコントローラとして機能します)。
- ViewBean は、アプリケーションクライアントで表示されてクライアントがアクセスできる、アプリケーション内の唯一のコンポーネントです。

ページレットと ContainerView

前述したように、ビュー層は通常、任意に入れ子になったビューコンポーネントの階層から構成されます。これらのコンポーネントの多くは、TiledView や TreeView、またはデータグリッド、入力フォーム、ヘッダーやフッターを始めとする他の上位コンポーネントなど、多様な ContainerView の 1 つになります。これらすべてを総称して、ページレット (ページの一部) と呼びます。ViewBean はページに類似していますが、ContainerView はページレットに類似しています。

ページレットは、それ自身の JSP フラグメントが関連付けられている場合も、その親ページ (またはページレット) を使って生成する場合もあります。ページレットで JSP フラグメントを使用する場合は、完全な JSP のように型への特定の依存を宣言する代わりに、フラグメントとそれに対応するページレット間の関連付けは、名前によって行われます。このため、アプリケーションの必要に応じて、複数の JSP フラグメントにより単一のページレットを動的に生成できます。これによって、きわめて柔軟に動的にページを生成することが可能になります。

コントローラ層

コントローラは、モデルとビュー間の動作、およびアプリケーション全体の動作を調整します。モデルおよびビューコンポーネントは多くの場合、既成の種類のインスタンスになりますが、一般に、コントローラはアプリケーションに合わせてカスタムに作成されます。

一部の J2EE アーキテクチャでは、コントローラの役割は、モノリシッククラス、マッピング、または非常に中央集中型のアプローチによる他の情報の格納によって実行されます。このアプローチは、初心者がアプリケーションを視覚化するには便利ですが、保守性、およびコンポーネント化された要素からアプリケーションを構成する能力という点で重大な欠点があります。このため Web アプリケーションフレームワークでは、本質的な分散アプローチを採用しています。このアプローチでは、コントローラの役割が、特定のタスクやアプリケーションコンポーネントにより密接に関連したコントローラオブジェクトとロジックユニットにさらに細かく分割されています。

前述のビューおよびモデル層とは異なり、少なくとも大局的な意味では、Web アプリケーションフレームワークには、MVC パターンのコントローラの役割に対応するオブジェクトやインタフェースはありません。この理由については、さまざまな Web アプリケーションフレームワーク機能を示す際に概説していきます。ただし現時点では、Web アプリケーションフレームワークでは、パターンのコントローラの役割を実行するために分散アプローチを採用しており、これによって再利用可能なコンポーネントを作成したり時間をかけてコンポーネントをアプリケーションにまとめる機能など、多数の利点が提供されているということを理解すれば十分です。

Web アプリケーションフレームワークの MVC と従来の MVC との違い

従来の (ステートフルなクライアント側の) MVC アーキテクチャでは、一般的に Publish/Subscribe イベントモデルを使ってオブジェクトを相互に関連付けます。しかし、このアプローチはステートフルオブジェクトを必要とし、一般に Web 層では、多数のユーザーへの拡張性がありません。J2EE Web 層の要求のライフサイクルは明確に記述されていて予測可能であり、またスケーラビリティが基本的な要件であるため、Web アプリケーションフレームワークではステートフルなサーバー側オブ

ジェクトを避けています。このアプローチでは、MVC オブジェクトを相互に関連付ける方法としてイベントリスナーを避けることが求められます。Web アプリケーションフレームワークでは MVC の層間の関係は維持されていますが、これらの関係は従来のイベントリスナーの関係ではありません。

第2章

アプリケーションの開発

この章では、後で完全に機能するアプリケーションに組み立てることが可能なアプリケーションコンポーネントの作成および使用方法について詳しく説明します。

アプリケーションの作成

Web アプリケーションフレームワークアプリケーションとは

Web アプリケーションフレームワークのアプリケーションは、基本的に J2EE の Web 層アプリケーションと同じです。要約すれば、Web アプリケーションフレームワークアプリケーションとは、Web アプリケーションフレームワーク固有の機能を備えた J2EE Web 層アプリケーションです。このため Web アプリケーションフレームワークアプリケーションでは、その基盤として標準の WAR (Web Application aRchive: Web アプリケーションアーカイブ) 構造を使用しています。したがって Web アプリケーションフレームワークアプリケーションを作成するときは、Web アプリケーションフレームワーク固有の WAR ファイルを作成し、そのアプリケーションの単一の `ServletContext` を暗黙に定義することになります。あらゆる Web アプリケーションフレームワークアプリケーションはそれぞれに、Web アプリケーションフレームワーク固有のデータを含む 標準的な `web.xml` ファイルを持ちます。

Web アプリケーションフレームワークアプリケーションは、最終的には WAR ファイル構造の J2EE アプリケーションであるため、同じアプリケーション内で他の J2EE 機能を使用できます。たとえば、非 Web アプリケーションフレームワーク JSP やサーブレットを、同じ WAR ファイルでまったく問題なく使用できます。J2EE 仕様で許容されている制限された範囲内で、こうした非 Web アプリケーションフレームワークアプリケーションコンポーネントは Web アプリケーションフレームワークア

プリケーションコンポーネントと相互運用することが可能です。詳細は、71 ページの「Web アプリケーションフレームワークアプリケーションとの相互運用」を参照してください。

Web アプリケーションフレームワークアプリケーションはまた、特定機能を提供するために対話する一連の関連コンポーネントを記述する論理エンティティでもあります。より具体的に言えば、アプリケーションとは、単一の WAR ファイルという 1 つの単位として配備される、ページ、ビュー、モデル、およびアプリケーション固有のコードのセットです。通常、アプリケーションの一部を変更した場合は、(J2EE 仕様に従い WAR ファイルとして) アプリケーション全体を再配備することが必要になります。アプリケーション WAR ファイルには、そのアプリケーションが機能するために必要な関連するすべてのライブラリ、クラス、およびリソースが含まれます。ただし、J2EE コンテナにより提供されるライブラリは含みません。

アプリケーションレベルのエンティティ

一般に、各 Web アプリケーションフレームワークアプリケーションは、Web アプリケーションフレームワーク実行時ライブラリ、`web.xml` ファイル、および作成された WAR ファイルに加えて、アプリケーションサーブレットを持ちます。このサーブレットは、アプリケーションの各種モジュールサーブレットのベースクラスとして機能するため (この後で説明)、アプリケーションレベルのイベント処理、初期化、または各要求に関連する他のタスクを実行する機会を提供します。

モジュール

現実のアプリケーションは、わずかに異なる機能要件を持つさまざまな部分から構成されて非常に大規模になる場合があるため、Web アプリケーションフレームワークにはモジュールというアプリケーションのサブユニットが追加されています。Web アプリケーションフレームワークアプリケーションは、1 つのユニットとして配備される 1 つ以上の Web アプリケーションフレームワークモジュールから構成されます。すべての Web アプリケーションフレームワークアプリケーションは、最低 1 つのモジュールを持つ必要があります。

Web アプリケーションフレームワークのモジュールと J2EE のモジュールを混同しないようにしてください。J2EE では、モジュールという用語は J2EE Web アプリケーション全体を示す場合があります。たとえば、J2EE Web アプリケーションや WAR ファイルが Web モジュールという名称で示されている場合もあります。このマニュアルでは、「アプリケーション」はすべて厳密には J2EE Web アプリケーションまたは WAR ファイル全体を意味し、「モジュール」はすべて Web アプリケーションフレームワーク固有のモジュールを意味します。このモジュールに類似するものは、J2EE にはありません。

物理的には、モジュールは、アプリケーション WAR ファイルの WEB-INF/classes ディレクトリ (この項では以降、パスはこのディレクトリへの相対パスとして示します) にある単一の Java パッケージです。モジュールパッケージは開発者が指定する任意のパッケージで、任意の数のサブディレクトリを含むことができます。たとえば、main と com.mycompany.main の両方とも、モジュールパッケージに指定できます。

モジュールパッケージと他の任意のパッケージの最も重要な違いは、アプリケーションのクライアントは、モジュールパッケージ内の ViewBean にのみアクセスできる、ということです (詳細は以下を参照)。これは主として、アプリケーションの web.xml ファイルのサーブレットマッピングを介して行われます。このマッピングでは、モジュールのモジュールサーブレット (以下に説明) を URL パスにマップします。通常、この URL パスは単純名で、完全限定名ではありません。たとえば、モジュールパッケージが実際には com.mycompany.main であっても、外部クライアントは /main という URL パスによってのみ識別できます。これによって URL の長さを短縮でき、アプリケーションユーザーが使用する際の複雑さが緩和されます。URL マッピングが、モジュールパッケージと類似のものを共有する必要はありません。前述の例では、/foo という URL パスも指定できます。

クライアントは (クラス名ではなく) 論理ページ名のみを要求できます。この名前は、モジュールサーブレットによって ViewBean クラス名に自動的にマップされます。その結果、外部クライアントは ViewBean にのみ直接アクセスできる、ということになります。セキュリティなどの理由から、モジュールパッケージ内のすべての ViewBean は、そのパッケージ内に直接置く必要があります。サブパッケージ内の ViewBean にはアクセスできません (これらの ViewBean にアクセス可能にするには、サブパッケージをモジュールとして宣言する必要があります)。ただしこれらは、アプリケーションにより他の方法で (例 : サブクラスとして) 使用できます。

モジュールサーブレット

各モジュールは、そのモジュール内のオブジェクトに対するすべてのクライアント要求を処理するモジュールサーブレットを 1 つ備えています。モジュールサーブレットはフロントコントローラとして機能して、すべてのクライアント要求をインターセプトし、アプリケーションコンポーネントに要求を送信する前に必要に応じてイベントを起動します。たとえば、モジュールサーブレットは、要求の開始と終了、セッション開始とタイムアウト、およびさまざまなエラー状態に関連したイベントを備えています。これらのイベントについては、26 ページの「要求の処理」を参照してください。

モジュールサーブレットは、通常アプリケーションのアプリケーションサーブレットから派生します。アプリケーションサーブレットはスーパークラスであるため、モジュールは必要に応じてアプリケーション全体に指定された動作を無効にしたり追加したりすることができます。これは、たとえば特定のモジュールが、セッションタイムアウトといった特定イベントの処理に異なる要件を持っている場合に役立ちます。モジュール作成者は、必要に応じてモジュールの動作を個別化しながら、アプリケーション全体にわたる共通のフロントコントローラ動作も利用できます。

モジュールサーブレットは、必ずしもアプリケーションのアプリケーションサーブレットベースクラスから派生する必要はありません。その代わりに、Web アプリケーションフレームワークの com.iplanet.jato.ApplicationServletBase クラスから直接派生することも可能です。ただし、派生を避ける必要性はほとんどなく、Web アプリケーションフレームワーク IDE は、アプリケーションサーブレットベースクラスから派生したモジュールサーブレットを自動的に作成します。

パッケージ構造

一般に、Web アプリケーションフレームワークアプリケーションの WEB-INF/classes ディレクトリは、以下のようになります。

```
/Base application package (複数ディレクトリの深さになる場合もある)
<Application servlet>
     <Other application classes/objects/files>
     /<module 1>
        <Module servlet>
        <Other module classes/objects/files>
     /<module 2>
        <Module servlet>
        <Other module classes/objects/files>
     ...
     /<module n>
        <Module servlet>
        <Other module classes/objects/files>
```

各モジュールは、アプリケーション全体の自己完結型サブパッケージです。あるモジュールのページやオブジェクトは、「モジュール間ナビゲート」と単純なクラス参照の両方または一方を使って、他のモジュールのオブジェクトと対話できます。モジュール間ナビゲーションについては後述します。

WAR ファイルの作成

Web アプリケーションフレームワークアプリケーションは、他の J2EE Web 層アプリケーションと同様に、標準 WAR ファイルにパッケージ化されています。Web アプリケーションフレームワークアプリケーションには、WAR ファイル内の特定ファイルの場所について、いくつかの前提条件があります。以下に、Web アプリケーションフレームワーク WAR ファイルの推奨される配置を示します。

```
/[base application package]
    /[module 1]
        [Module JSP files]
    /[module 2]
        [Module JSP files]
    ...
    /[module n];
        [Module JSP files]
/[other static resources]
/WEB-INF
    web.xml (アプリケーション配備記述子)
    /classes
        /[base application package]
            [Application servlet class]
            [Other application classes/objects/files]
            /[module package 1]
                [Module servlet]
                [Other module classes/objects/files]
            /[module package 2]
                [Module servlet]
                [Other module classes/objects/files]
            ...
            /[module package n]
                [Module servlet]
                [Other module classes/objects/files]
    /lib
        [Web Application Framework runtime jar file]
        [other Web Application Framework component libraries]
        [other application jar/zip files]
    /tld
        /com_iplanet_jato
            jato.tld (Web アプリケーションフレームワークタグライブラリ記述子)
```

JSP ファイル用とモジュールクラス用に 1 つずつ、2 つの並行するディレクトリ階層が存在します。これら 2 つのディレクトリ階層は並行に置く必要がありますが、厳密に言えば、並行にする必要はありません。具体的に説明すると、それぞれの ViewBean は、開発時にその JSP ピアの URL で構成されますが、この URL は任意に指定できます。ただし、別のグループが開発したモジュールの名前が衝突するのを防ぐために、前述の並行ディレクトリ構造を維持する必要があります。

コンポーネントライブラリの使用

以前のバージョンの Web アプリケーションフレームワークは特定の種類のアプリケーションコンポーネントに対する充実したサポートを備えていましたが、バージョン 2.0 ではこのサポートに一定の形が与えられて、ほとんどの種類のフレームワークオブジェクトに拡張されています。バージョン 2.0 からは、ビュー、モデル、コマンド、およびサポートするクラスは、すべてコンポーネントとして指定できます。この指定により、これらのオブジェクトは IDE で調べて操作できるようになっており、Web アプリケーションフレームワークアプリケーションを非常に高い生産性でビジュアルに開発することが可能となっています。

Web アプリケーションフレームワークでは一連の基本コンポーネントが定義されており、Sun Microsystems を始めとする開発企業はコアフレームワークの機能と能力を大幅に拡張する追加コンポーネントライブラリを提供していく予定です。独自の再利用可能なコンポーネントおよびコンポーネントライブラリの作成については、『Web アプリケーションフレームワーク コンポーネント作成ガイド』を参照してください。

Web アプリケーションフレームワークコンポーネントとは

その登場以来、Web アプリケーションフレームワークアプリケーションは特定の種類のオブジェクトに対してコンポーネントモデルをサポートしてきました。ただし、以前のコンポーネントモデルは、開発者が各コンポーネントの API を学び、自分のアプリケーションでそのコンポーネントを利用するためのコードを書くことを前提にしていました。かっては、このレベルの機能で十分であり、生産性の面で競合製品 (現在も依然としてコンポーネントモデルがない) に比べて優れていましたが、Web アプリケーションフレームワークのバージョン 2.0 では、あらゆる種類の主な Web アプリケーションフレームワークアプリケーションオブジェクト (ビュー、モデル、コマンド) を包含し、機能豊富な IDE を使って Web アプリケーションフレームワークアプリケーションの開発が可能になるまでに、コンポーネントモデルが大幅に拡張されています。

バージョン 2.0 では、コンポーネントとは、メタデータ情報と連携してサポートされている各種コンポーネントクラスの 1 つを意味します。このメタデータは、`ComponentInfo` クラスという Web アプリケーションフレームワーク固有のクラスにカプセル化されます。設計時には、開発環境によって ComponentInfo を検査し、使いやすいビジュアルな形でコンポーネントを表示できます。

`ComponentInfo` クラスに格納されているメタデータは、Sun Java Studio Enterprise 7 IDE のような開発環境でコンポーネントを自動的に使用できるように意図されています。一方で開発者は、`ComponentInfo` クラスを定義しないで、アプリケーションの各種コンポーネントを手動で作成し、使用することもできます。

Web アプリケーションフレームワークコンポーネントには、2 つの種類があります。1 番目の種類は、配布可能コンポーネントと呼ばれます。配布可能コンポーネントはコンポーネントライブラリにパッケージ化され、通常はコンポーネントセットの一部

として配備されます。配布可能コンポーネントには明示的な ComponentInfo クラスが関連付けられ、特に他の開発者が使用できるように配備するものとして開発されます。

これとは対照的に、2 番目の種類の Web アプリケーションフレームワークコンポーネントは、アプリケーション固有コンポーネント、または配布不可コンポーネントと呼ばれます (この場合の配布可能という用語を J2EE の配布可能と混同しないように注意してください。J2EE の場合の配布可能は複数 VM 上で動作するアプリケーションを意味しますが、ここでは他の開発者への配布を意味します)。これらのコンポーネントは、それらが定義されたアプリケーション内でのみ再利用できます。通常これらには、明示的な ComponentInfo は関連付けられておらず、現在のアプリケーションの最上級オブジェクトとして存在します。たとえば、アプリケーションで ContainerView または Model を構築する場合は常に、そのアプリケーション内で使用するための配布不可コンポーネントを構築することになります。IDE はこれらのコンポーネントを直接操作する方法を認識しているため、開発者は追加作業を一切行わないで同じアプリケーション内でこれらを使用できます。ただし、配布不可コンポーネントを配布可能コンポーネントに変換する場合は、通常は付加的な作業が必要となります (明示的 ComponentInfo クラスの追加など)。

さらに、拡張可能コンポーネントという特殊なカテゴリのコンポーネントもあります。拡張可能コンポーネントは、サブクラス化して新しい種類のコンポーネントを作成できるコンポーネントです。拡張不可コンポーネントでは、そのインスタンスを作成することはできますが、拡張して新しい動作を追加することはできません。現バージョンの Web アプリケーションフレームワークから、すべての配布不可コンポーネントは拡張可能コンポーネントから派生しています (ただし、配布不可コンポーネント自体は、拡張可能コンポーネントではありません)。特定の配布可能コンポーネントのみが、拡張可能コンポーネントとなります。

最初はこれらの説明を理解できなくても、意識する必要はありません。こうした詳細にわたる機能は IDE によって自動的に管理されるため、開発者はアプリケーションの構築に集中できます。アプリケーション開発者は通常、コンポーネントが拡張可能かどうかや ComponentInfo クラスを備えているかどうかを意識する必要は一切ありません。

Web アプリケーションフレームワーク コンポーネントライブラリ

標準の Web アプリケーションフレームワークコンポーネントライブラリには、主要インタフェースと実行時クラス、それに、Web アプリケーションフレームワークアプリケーションの作成に利用できる多数の基本的なコンポーネントが含まれています。このコンポーネントライブラリ は、1 つの JAR ファイルとしてパッケージ化され、アプリケーションの WEB-INF/lib ディレクトリに出現します。

IDE を使った Web アプリケーションフレームワークアプリケーションの作成では、標準のコンポーネントライブラリの最新バージョンが、自動的にアプリケーションの WEB-INF/lib ディレクトリに追加されます。以前のバージョンの IDE ツールセット

で作成したアプリケーションを開くと、Web アプリケーションフレームワーク実行時ライブラリを含めてアプリケーションをアップグレードするよう促されることがあります。

他のコンポーネントライブラリ

Web アプリケーションフレームワークコンポーネントライブラリに加え、他のコンポーネントライブラリも、アプリケーションの `WEB-INF/lib` ディレクトリに配置するだけでアプリケーションに追加できます。IDE はコンポーネントライブラリを自動的に認識してマウントします (これは 1 ～ 2 分かかります)。この後、ライブラリの新しいコンポーネントがアプリケーション内で使用可能になります。

タグライブラリのアンパック

ライブラリ開発者は Web アプリケーションフレームワークコンポーネントライブラリの一部として、ライブラリのビューコンポーネントの生成をサポートする 1 つ以上のタグライブラリを提供できます。タグライブラリはコンポーネントライブラリのコンポーネントマニフェストファイルで宣言され、IDE がコンポーネントライブラリを認識すると、そのタグライブラリ記述子 (`.tld` ファイル) がアプリケーションで使用できるようにライブラリ JAR ファイルから自動的にアンパックされます。さらに IDE は、タグライブラリエントリを `web.xml` ファイルに自動的に追加します。

別のライブラリの同名のファイルが衝突しないように、タグライブラリ記述子ファイルはライブラリの名前に基づいてアプリケーションの `WEB-INF/tld` ディレクトリの下の特別な場所にアンパックされます。このスキーマでは、ドット (.) が下線 (_) に置き換えられて、ライブラリ名がディレクトリ名に変換されます。たとえば、コンポーネントライブラリの内部ライブラリ名が `com.iplanet.jato` である場合、これはタグライブラリ記述子のアンパック時に `com_iplanet_jato` に変換されます。Web アプリケーションフレームワークコンポーネントライブラリのタグ記述子ファイルは、最終的にはアプリケーションの `WEB-INF/tld/com_iplanet_jato` ディレクトリの下に配置されます。

派生した、タグ記述子の物理ディレクトリ名は、アプリケーションが使用できるように `web.xml` ファイルの論理リソース名に自動的に登録されます。この論理名は、コンポーネントライブラリの作成者が指定します。Web アプリケーションフレームワークコンポーネントライブラリの場合、記述子は `/WEB-INF/jato.tld` リソースとして登録されます。

タグ記述子のアンパック機構は、タイムスタンプを使用して、新しいバージョンのライブラリがアプリケーションに追加されたときに既存ファイルを上書きするかどうかを判断します。この機能により、開発者は 1 つのステップだけでアプリケーションのコンポーネントライブラリをアップグレードすることが可能となっています。

他のファイルのアンパック

IDE はタグ記述子ファイルを自動的にアンパックすることに加え、ライブラリ作成者が必要とする他のファイルについても、コンポーネントライブラリがアプリケーションに追加されたときにアンパックすることができます。こうしたファイルには、JAR ファイル、静的 HTML ファイル、イメージファイル、JSP、または他の任意の種類のファイルがあります。これは IDE で新しいコンポーネントライブラリがアプリケーションに追加されたときに完全に自動的に実行されますが、各コンポーネントライブラリのマニュアルを必ず読み、新しいライブラリの追加後にアプリケーションディレクトリ構造に突然表示される追加ファイルに注意してください。

ページ (ViewBean) の作成

前述したように、アプリケーションの 1 つの論理ページは、通常 ViewBean と関連付けられた JSP の 2 つの部分から構成されています。IDE では、ViewBean の部分が重視されています。なぜなら ViewBean は Java クラスであり、開発者が必要とする種類の作成作業 (コンポーネントの構成など) に非常に適しているからです。このため JSP については論理ページオブジェクトの付随的な部分と考え、以下の節ではまず ViewBean の作成に必要な作業を重点的に説明してから、その JSP セットの作成および管理方法について説明します。

ViewBean クラスの作成

ViewBean は簡単に作成できます。IDE を使用する場合は、IDE で「新規」->「Web アプリケーションフレームワーク (JATO)」->「ページ (ViewBean)」を選択してから、新規ウィザードの ViewBean の一覧から選択するだけで作成できます

アプリケーションのクライアントが ViewBean にアクセスできるようにするには、Web アプリケーションフレームワークモジュールパッケージ内で ViewBean を作成する必要があります。モジュールパッケージ外部の ViewBean は、モジュールパッケージ内部の他の ViewBean のスーパークラスとして使用できますが、それ自体にクライアントがアクセスすることはできません。

ウィザードを使って ViewBean を作成するときは、通常はその ViewBean を生成するための、対になる (対応する) JSP ファイルを作成します。この JSP ファイルはアプリケーションのドキュメントルートに置かれ、*<jato:useViewBean>* タグ宣言によって自動的に ViewBean と関連付けられます。JSP へのパスには ViewBean のパッケージ名が反映されるという規則により、`com.mycompany.main` パッケージに配置された ViewBean は、アプリケーションドキュメントルートの `com/mycompany/main` サブディレクトリに作成された JSP を持つようになります。ViewBean の JSP 関連付けについて、以下の項で詳しく説明します。

手動で ViewBean を作成する場合は、アプリケーションのモジュールパッケージの 1 つに `com.iplanet.jato.view.BasicViewBean` または他の `ViewBean` クラスのサブクラスを単に作成するだけです。ただし、手動で作成した ViewBean は IDE では、プレーン Java ファイルとして操作するため、すべてのコンポーネントやイベントハンドラを ViewBean に手動で追加する必要があります。さらに、手動で JSP ファイルを追加して ViewBean に関連付ける必要があります。

命名

前バージョンの Web アプリケーションフレームワークでは、厳密な命名規則を使って要求ページ名を ViewBean クラス名にマップしていました。バージョン 2.1 からはこの制限がなくなり、ViewBean は他のクラスと同じように命名できるようになっています。下位互換性の維持が必要な既存アプリケーションに対しては、`web.xml` の設定を使って厳密な命名規則を使用するように切り替えることができます。詳細については、75 ページの「アプリケーションの配備」を参照してください。

デフォルトでは、IDE によって作成されるアプリケーションでは、厳密な命名規則に従わない ViewBean を使用できます。つまり ViewBean クラスには、希望する名前を自由に指定できます。

コード

IDE で ViewBean クラスをダブルクリックするか、テキストエディタでこれを開くと、ViewBean は通常最初に作成されたときから存在するコードに加え、少なくとも IDE では編集できない保護ブロックのコードを備えていることが分かります。

ViewBean または他の Web アプリケーションフレームワークオブジェクトの保護コードブロックは、別のエディタを使って編集してはいけません。次回 IDE でクラスを変更したときに、その変更内容は上書きされてしまいます。保護領域内の特定スポットにコードを追加する必要がある場合は、追加が必要かどうかを確認するか (ほとんどの場合コードを追加する必要はない)、または ViewBean あるいはその子コンポーネントのプロパティの「コード生成」タブを使って標準的な方法でコードを追加してください。

通常は必要とする他のコードやメソッドを ViewBean に追加できます。追加できない場合は、IDE で視覚的に表現されていないメソッドの一部をオーバーライドしてください。上級テクニックとしてオーバーライドできるメソッドには、`getDisplayURL()`、`mapRequestParameters()`、`setRequestContext()`、`securityCheck()`、`beginChildDisplay()`、および `handleRequest()` があります。

ViewBean クラス内のコードは、一般的にスレッドに対して安全である必要はありません。なぜなら、各要求スレッドは ViewBean の専用コピーを取得するためです。また、ViewBean は要求スコープのオブジェクトであるため、要求間で ViewBean インスタンスにデータを保存しないようにしてください。

JSP の管理

クライアント要求への応答として ViewBean を生成するための最も一般的なテクニックは、ピア JSP を使用する方法です。この JSP には、そのピア ViewBean と自身を関連付けるカスタムの Web アプリケーションフレームワークタグが含まれています。実行時には、JSP と ViewBean が連携動作して、現在の要求に対する動的応答を生成します。この JSP と ViewBean の組み合わせにより、JSP テクノロジの長所であるパワフルなレイアウト機能およびコンテンツ機能が実現する一方で、コードを使わないで JSP を簡単に維持することが可能となっています。

各 JSP は 1 つの ViewBean とだけ関連付けることができますが、1 つの ViewBean は同時に多数の JSP と関連付けることができます。こうした他の JSP は、ViewBean の生成に使用されるコンテンツとレイアウトのバリエーションを含むことができます。この並列コンテンツ機能については、37 ページの「URL と並列コンテンツの表示」で説明します。ただし、この機能はあっても、ViewBean は主にほとんどのアプリケーションで単一の JSP のみを使用します。

JSP は、その対ピア ViewBean との間に、利用関係とよく呼ばれるものを持っています。換言すれば、JSP は応答の生成中、そのピア ViewBean を使用します。一方 ViewBean は、それに関連付けられた JSP を同じようには使用しません。つまり、それらの関係は、釣り合いのとれたものではありません。生成時に、JSP は ViewBean からデータを取得します。ViewBean は JSP によって呼び出されてこのデータを提供しますが、ViewBean が JSP を呼び出すことはありません。生成または表示プロセスという点では、JSP は操縦者として機能します。

JSP を ViewBean と関連付けるには、少なくとも、ViewBean のクラスを呼び出す有効な *<jato:useViewBean>* タグ宣言を行う必要があります。実行時に ViewBean が自身を表示するために転送されるとき、ViewBean は一致する *<jato:useViewBean>* タグを持つ JSP を選択し、(`getDisplayURL()` および `getDefaultDisplayURL()` メソッドの両方または一方を介して) その表示 URL としてアプリケーションのドキュメントルートへの相対パスである JSP の URL を返す必要があります。

Web アプリケーションフレームワーク IDE では、各 ViewBean ノードは「JSP ページ」というサブカテゴリノードを備えています。このノードの下に 1 つ以上の JSP ノードが表示され、これらの JSP はそれぞれ ViewBean のクラス名を呼び出す *<jato:useViewBean>* タグ宣言によって現在の ViewBean と関連付けられます。

関連付けられたこれらの各 JSP は、ViewBean によって管理されることが前提とされます。「管理される」とは、子ビューコンポーネントの追加時に、これらの JSP が ViewBean と自動的に同期化されることを意味します。(JSP を管理する、というこの概念は、単に設計時のみの概念で、実行時についての意味はありません。) この機能の詳細は、以下の項で説明します。

開発者にとってこれらの JSP ノードの最も有用な点は、これによって ViewBean の JSP ファイルを簡単に開けることと、管理されている JSP が複数ある場合は ViewBean の表示用のデフォルトを選択できることです。開発者は、JSP ページカテ

ゴリの任意の JSP ノードを右クリックし、「デフォルトとして設定」を選択することで、その JSP を ViewBean の現在のデフォルトとして選択できます。一度に複数のデフォルトを設定することはできません。

特定の JSP を ViewBean のデフォルトとして設定すると、`setDefaultDisplayURL()` メソッドを介して、その JSP の URL が ViewBean のデフォルト表示 URL として設定されます。ViewBean でこのメソッドを呼び出すコードが、保護コードブロックの 1 つに生成されます。

実行時に ViewBean が実行されると、異なる値を返すように開発者が ViewBean の `getDisplayURL()` メソッドをオーバーライドしない限り、デフォルト JSP を使って ViewBean が生成されます。したがって開発時には、異なるデフォルト JSP を設定して ViewBean を再コンパイルすることで、JSP を切り替えてテスト実行できます。

子ビューコンポーネントの追加

ViewBean は、ページの表示フィールドを表す子ビューコンポーネントをそれに追加しない限り、一般的に役に立ちません。したがって、ページ作成の重要な部分は、子ビューコンポーネントを ViewBean に追加して、モデルデータにアクセスするようにこれらを構成することになります。

ViewBean の「可視コンポーネント」ノードを右クリックして「可視コンポーネント ...追加」を選択することで、子ビューコンポーネントを ViewBean に簡単に追加できます。コンポーネントブラウザが表示され、アプリケーションで現在利用できるビューコンポーネントと、それにマウントされている各アプリケーションライブラリが示されます。または、コンポーネントパレットから、追加するコンポーネントを選択することもできます。

子ビューコンポーネントは、基本的に 2 種類あります。最初の種類は、`com.iplanet.jato.view.DisplayField` インタフェースを実装し、ビューコンポーネントのツリー上で事実上「葉」として機能するビューです。DisplayField は通常、文字列や整数といった 1 つの基本データを表し、多くの場合はユーザーが変更してそれをサーバーに送付できます。DisplayField ビューコンポーネントの好例は、標準コンポーネントライブラリに装備されているテキストフィールドコンポーネントです。

複雑な DisplayField コンポーネントもあります。DisplayField コンポーネントが ContainerView インタフェースも実装し、自身の子ビューコンポーネントを備えている場合です。開発者は通常、DisplayField としてのこれらのコンポーネントと対話を行い、これらのコンポーネント自体が複合コンポーネントであることはまったく認識しません。

標準コンポーネントライブラリで提供されるさまざまな DisplayField コンポーネントは、すべて ModelReference (モデル参照) およびモデルフィールドバインドによってモデルと関連付けることが可能です。詳細は、51 ページの「値の操作」を参照してください。

2 番目の種類のビューコンポーネントは ContainerView (またはページレット) で、これは実際に TiledView や TreeView、およびサン以外が提供する他の種類といった 2、3 のサブタイプを含んでいます。ContainerView は他のビュー (他の ContainerView を含む) を含むことができる特殊なビューで、複合コンポーネントとして機能します。ビューコンポーネントは ContainerView を通して、任意のレベルに入れ子にできます。これは通常、ビューコンポーネント階層と呼ばれます。クライアント側アプリケーションで ContainerView という場合は、Swing の JPanel のようなパネルコンポーネントに相当します。TiledView コンポーネントは、大まかに言えば表コンポーネントです (ただし、表としてだけ使用されるわけではありません)。また、TreeView コンポーネントは、Swing の JTree のようなツリーコンポーネントに相当します。

さまざまな種類の ContainerView にカスタムのプロパティやメソッドを装備することで、開発者は ContainerView に含まれる複雑なコンポーネントセットを意識したり、これらを使って作業を行わなくても、より簡単に ContainerView と対話できます。ContainerView は多くの場合、アプリケーション固有の配布不可コンポーネントです。つまり、これらは通常、開発者が部分的に実装したり拡張したりできる、現アプリケーション内の型になります。

ContainerView コンポーネントの ViewBean (または他の ContainerView) への追加に関しては、他の種類のビューコンポーネントを追加するのと実質的にまったく同じです。ContainerView は、他のビューコンポーネントと同じように設定できるプロパティを表示します。1 つ違う点は、ContainerView は一般的に、ViewBean の JSP と対話するために、その内部の複雑さを示すことです。具体的に言えば、一般に開発者は ContainerView の子を「見る」ことができ、個々にこれらのコンポーネントの同期をとって配置できます。

ViewBean または他の ContainerView に子ビューコンポーネントが追加された後は、開発者は単にそのプロパティシートを使ってこれを構成する必要があります。各プロパティに入力すると、ViewBean で生成されたコードが更新されて、実行時のコンポーネントが構成されます。さらに、それぞれの子ビューコンポーネントが ViewBean に追加されると、一致する JSP タグが、ViewBean が管理する JSP に追加されます。このタグは最小限に構成され、JSP の静的コンテンツレイアウトの基本的な観点からのみ追加されます。開発者は必要な他の属性をこのタグに追加して、ページレイアウトに合わせて配置する必要があります。詳細は、以下の項を参照してください。

JSP との同期化

子ビューコンポーネントは、ViewBean に追加されても、ViewBean の表示時に必ずしも生成されるわけでも、また生成される必要があるわけでもありません。たとえば、一部のビューコンポーネントは、ViewBean に関連付けられた特定 JSP でのみ現れる必要があったり、モデルフィールドとの関連付けのためにアプリケーションコードでのみ使用されたりします。

子ビューコンポーネントは、ViewBean と関連付けられた JSP における表現で、その ViewBean で生成される表現を持つ必要があります。この表現は通常、コンポーネント作成者が提供するカスタム JSP タグになります。たとえば、IDE で ViewBean にテキストフィールドコンポーネントを追加すると、IDE によって *<jato:textField>* タグが JSP に自動的に追加されて、ViewBean の表示時にそのコンポーネントが生成されるようになります。

したがって、ビューコンポーネントを ViewBean に追加するという概念にもともと備わっているのは、ViewBean が管理する 1 つ以上の JSP を子ビューコンポーネントのセットとの同期を取るという考えです。各 JSP は基本的にそれに関連したフィールドのみを使用する能力を備えているため、開発者はそれぞれの子ビューコンポーネントの JSP 表現を簡単に管理する方法が必要となります。

「ビューと同期」機能は、この JSP 管理を簡単にすることを目的にしています。ViewBean の JSP ノードを右クリックしてこのメニュー項目を選択すると、ダイアログが表示され、ここでその特定の JSP に表示したいそれぞれの子ビューコンポーネントを選択できます。「了解」を押すと、JSP に変更が加えられます。すなわち、削除されたコンポーネントタグが除去され、挿入されたコンポーネントタグが追加されます。その後で JSP ファイル自体を開き、JSP の全体的なレイアウト内で作用するようにこれらのタグを配置できます (挿入されたタグはファイルの最後のほうに追加されます)。

この機能は単なる便利というだけです。JSP ファイルのタグを追加、編集、または削除することで、ViewBean と JSP の同期を手動で取ることができます。実際に、一部のビューコンポーネントは自動同期をサポートしておらず、生成される出力に表示するには手動で同期を取る必要があります。

JSP タグは、JSP のコンテンツの種類に基づいて JSP に追加されます。開発者は、そのコンポーネントがサポートするコンテンツの種類それぞれについてタグテンプレートを選択できます。したがって、各種のコンテンツを使用すると、JSP の独自の用法によっては、同じビューコンポーネントに異なる JSP タグが使用される場合があります。各タグセットがサポートする機能も、コンテンツの種類によって異なる場合があるため注意してください。

IDE からのページの実行

Web アプリケーションフレームワークアプリケーションを記述するとき、多くの開発者は、現在のページを実行してその動作をチェックし、その JSP レイアウトをテストしたいと思うでしょう。手動でアプリケーションをパッケージ化し、コンテナに配備してから適切な URL でブラウザを開かなくても、ViewBean を実行するという 1 回のステップでこのすべてを行うことができます。

最初に、アプリケーションノードから「すべて構築」を選択することによって、少なくとも 1 回アプリケーションを完全にコンパイルしておきます。(ViewBean を実行すると、現在のモジュールのすべてのファイルがコンパイルされます。) 必要な ViewBean を右クリックして、「ページを実行 (再配備)」またはツールバーボタンを

選択します。Web アプリケーションのデフォルトとして選択した J2EE サーバーに対して (Sun Java Studio Enterprise 7 ヘルプを参照)、アプリケーションが自動的に配備され、ブラウザが起動されてその ViewBean に適切な URL が読み込まれます。アプリケーションコードに変更を加えた場合は、「実行 (強制再読み込み)」オプションを使ってページを実行する必要があります。これを行わないと、アプリケーションに変更が反映されません。この唯一の例外は JSP に対する変更で、これは自動的に検出されます。

実行できるのは、Web アプリケーションフレームワークモジュール内の ViewBean だけです。モジュール外部の ViewBean は、アプリケーションクライアントからはアクセスできないことに注意してください。

ページレット (ContainerView) コンポーネントの作成

この節では、コンポーネントライブラリ内に含まれている拡張可能コンポーネントからのビューコンポーネントの作成方法について説明します。拡張可能コンポーネントは、拡張 (サブクラス化) して新しい種類のコンポーネントを作成できるように設計されているコンポーネントです。(拡張可能コンポーネントを基盤として使って) 新しいコンポーネントの種類を定義したら、その多くのインスタンスを所定のアプリケーション内で使用できることがあります。

ContainerView クラスの作成

ContainerViews クラスの作成プロセスは、ViewBean クラスの作成に類似しており、実際に異なる点はありません。唯一の違いは、すべてのページレットと同様に ContainerView は単独では実行できない、という点です。ContainerView を実行するには、他のコンポーネントの子に指定して、ページ内に配置する必要があります。

もう 1 つの違いは、ContainerView および他の種類のページレットは、JSP フラグメントを作成しないで、その代わりにすべての生成をその親コンポーネントに委任することも可能だ、という点です。これは、ページレットとその子を表すカスタムタグが、スタンドアロンの JSP フラグメント内ではなく、親の JSP (または、親が自身の JSP を持っていない場合はその親の JSP) 内に配置されることを意味します。

一次モデル

多くの種類の ContainerView では、一次モデル、つまりその生成を推進するモデルという考え方が採用されます。たとえば、TiledView の実装は、タイルからタイルに進み、また繰り返しを中止するタイミングを知るために、通常 DatasetModel の種類の一次モデルを必要とします。同様に、TreeView の実装でも、そのデータの階層ビューを表示するために、`TreeModel` の種類の一次モデルを必要とします。

要求の処理

Web アプリケーションフレームワークアプリケーションは、クライアント要求を処理して応答するために作成されます。この節では、クライアント要求を処理して応答するために Web アプリケーションフレームワークが提供するさまざまな機能について概説します。

要求ライフサイクル

一般的に、各要求は送付フェーズと表示フェーズの 2 つの部分から構成されます。

Web アプリケーションフレームワークのページにユーザーがアクセスすると、アプリケーションが実行されているサーバーに対する HTTP 要求が生成されます。このクライアント要求は、最初は、特定アプリケーションモジュールのフロントコントローラとして動作するアプリケーションのモジュールサーブレットによって処理されます。モジュールサーブレットは、さまざまな要求ライフサイクルイベント (セッションタイムアウト検出など) を起動してから、クライアントから送付された情報に基づいて、呼び出す ViewBean クラスを判断します。サーブレットが呼び出す ViewBean クラスは、通常、ユーザーの前回の応答の生成を担当したものと同じクラスになります。このようにして、アプリケーション開発者の指示に従って、サーバー上でのみ Web アプリケーションフレームワークでページ (ViewBean) 間の切り替えが行われます。

要求に応答するようにページが指示されると、ページの ViewBean の `invokeRequestHandler()` メソッドがモジュールサーブレットによって呼び出されます。これは、正しくアプリケーションに入る最初のポイントです。これによって送付フェーズが開始します。送付フェーズ時、送付された値はビューコンポーネントとモデルにマップされ、それからイベントハンドラが呼び出されて要求を処理し、結果を準備します。送付フェーズは通常、アプリケーション (またはフレームワーク) がイベントハンドラ内から同じまたは別の ViewBean 上で `forwardTo()` メソッドを呼び出したときに終了し、これによって表示フェーズが開始します。

表示フェーズ時には、多数のことが実行されます。これらの中で最も重要なのは、ViewBean に関連付けられた JSP が生成されることです。これによって、関連付けられた ViewBean の `beginDisplay()` メソッドへのコールバックが実行されます。その後 JSP の処理にともない、ViewBean とそのすべての子が生成を行い、最後に ViewBean 上で `endDisplay()` メソッドが呼び出されて完了します。しかし、`forwardTo()` メソッドが呼び出し元コードに戻った時点では、要求はまだ完了していません。このコードが完了すると、呼び出しスタックが展開されて、応答がクライアントに送信されます。

呼び出しスタックが解体される直前に、モジュールサーブレットはさらなる要求ライフサイクルイベントを起動し、セッションへの最終情報の追加といったことを実行できるように登録されているすべての RequestCompletionListener が呼び出されます (ただし、応答にはまったく影響を与えません)。これによって要求が完了します。

フロントコントローライベント

各要求は個々の要求ハンドラ (ViewBean またはその子の 1 つ) に送信される前に、フロントコントローラによって処理されます。具体的には、フロントコントローラとは、モジュールサーブレットです。モジュールサーブレットは通常アプリケーションのアプリケーションサーブレットから派生し、アプリケーションのアプリケーションサーブレットは通常 `com.iplanet.jato.ApplicationServletBase` から派生します (このスキーマが変動する場合もありますが、最終的には、すべての Web アプリケーションフレームワークサーブレットは `ApplicationServletBase` から派生する必要があります)。ApplicationServletBase は、要求処理時に多数のイベントを起動します。開発者は、アプリケーションまたはモジュールサーブレットのイベントコールバックメソッドをオーバーライドすることで、これらのイベントに応答できます。これらのイベントによって、一般的な要求処理ロジックを 1 カ所にまとめることができます。

モジュールサーブレットが起動するイベントを、以下の表に示します (詳細は、`com.iplanet.jato.ApplicationServletBase` の JavaDoc を参照してください)。

イベントメソッド	説明
`onAfterRequest`	現在の要求の完了後に起動されます。要求固有のリソースをクリーンアップしたりセッションのデータを確定するために使用できます。
`onBeforeRequest`	新しい要求が受信されたときに起動されます。要求の一部 (ユーザー主体パラメータなど) の確認、あるいは特定の条件での要求のリダイレクトに使用できます。
`onInitializeHandler`	要求ハンドラインスタンス (通常は ViewBean) にアプリケーション固有の初期化を行うために起動されます。要求属性に基づいて ViewBean の一般的な初期化を行うために使用できます。

イベントメソッド	説明
onNewSession	セッションなしの要求が受信されて、クライアントのために新しいセッションが作成されるときに起動されます。ユーザーセッションにデータを事前設定するために使用できます。
onRequestHandlerNotFound	要求された要求ハンドラ (ViewBean) が見つからない場合に起動されます。通常は応答するために使用されます。
onRequestHandlerNotSpecified	要求に要求ハンドラ (ViewBean) が指定されていない場合に起動されます。
onSessionTimeout	セッション ID の有効期限が切れた要求が受信されたときに起動されます。通常、クライアントのセッションを再初期化したり、ログインページにリダイレクトするために使用されます。
onUncaughtException	要求処理時に予期しない例外が発生したときに起動されます。通常、エラーページをクライアントに表示するために使用されます。

アプリケーション開発者は、単にこれらのイベントメソッドをオーバーライドし、必要なロジックを実行するだけで、これらのイベントを使用できます。最も一般的な動作は、クライアントが本来要求したページ以外のページに要求をリダイレクトする動作です。これは、エラーまたはセッションタイムアウト状態でよく使用されます。ただし、これらのイベントは要求処理が発生すると起動され、通常の処理を停止すべき旨をメイン要求処理ロジックに伝達する特定のメカニズムを提供しません。

したがって、こうした状況では、以下のようなテクニックがよく使用されます。Web アプリケーションフレームワークは、`com.iplanet.jato.CompleteRequestException` という特殊な例外クラスを提供します。これは、クライアントに実際にエラー状態を発生させることなく、要求処理時に任意の場所でスローできます。この例外は、要求の処理を他の ViewBean にリダイレクトした後に、またはサーブレット API を使ってクライアントに結果を直接送信した後に、`CompleteRequestException` をスローして、元の要求がその途中で停止するようにする目的でよく使用されます。この例外は基本的に、開発者がすでに応答を処理したため、さらなる処理は不要であるということを、Web アプリケーションフレームワークの要求処理インフラストラクチャに伝えます。以下に、セッションタイムアウトが発生した場合に、このテクニックを使って要求を別の ViewBean にリダイレクトする例を示します。

```
public void onSessionTimeout(RequestContext requestContext)
{
    // アプリケーションの ViewBean を取得
    ViewBean viewBean=
requestContext.getViewBeanManager().getViewBean(SessionTimeoutViewBean.class);
    // 要求を ViewBean に転送
    viewBean.forwardTo(requestContext);

    // 通常の要求処理を中止
    throw new CompleteRequestException();
}
```

すでに応答が処理されている場合に CompleteRequestException のスローを行わないと、通常は J2EE エラーまたはアプリケーションエラーが発生するか、あるいは最初の応答がコンテナによって完全に無視されて、イベント処理に失敗します。

アプリケーション内でコードを配置する他の場所とは異なり、モジュールまたはアプリケーションサーブレット内のコードはスレッドに対して安全でなければなりません。したがって、メンバー変数 (静的その他) を使って要求スコープのデータをサーブレットに格納しないようにしてください。

アプリケーションイベント

フロントコントローラが要求のためにさまざまなイベントを起動し、要求を送信すべき要求ハンドラ (例 : ViewBean) を決定した後で、ViewBean が呼び出されて要求の処理が完了します。具体的に言えば、ViewBean が行うのは、そのサブコンポーネントの中からどれが要求の処理を担当するのかを判断し、実際にイベントを処理するためにそのコンポーネントを呼び出すことです。このプロセスは要求の処置と呼ばれます。

ここで、コントローラの論理的な役割が、少なくとも一部は ViewBean によって実行されている、という意見も出るでしょう。これまで ViewBean もまたビューコンポーネントであると説明してきたため、これは混乱を招くかも知れません。これが、ViewBean を特殊な存在にしている 1 つの理由です。ViewBean は、ビューと要求の部分的コントローラとしての二重性を持っているのです。実際にこの状況はかなり直観的なものです。クライアント側アプリケーションのウィンドウやダイアログオブジェクトが、その中に含まれるどの GUI ウィジェットがマウスクリックイベントを受信すべきかを判断するのと同様に、ViewBean はその中のどのビューコンポーネントが要求イベントに応答すべきかの判断を行います。

このアプローチは、Web アプリケーションフレームワークがコンポーネントモデルを使ってコンポーネントをアプリケーションにまとめるという事実を考えると、自然なものです。Swing と同様に、可視コンポーネントは、その子コンポーネントのどれがクリックされたかを判断するためのロジックを備えているため、Web アプリケー

ションフレームワークも同じようなロジックを備えています。この機能がないと、ビュー階層を形成するためにアセンブルして結び付けることが可能なビューコンポーネントを作成することはできないでしょう。

要求処置時には、ViewBean とそれに含まれる各 ContainerView が順々に `javax.servlet.http.HttpServletRequest` オブジェクトを調べて、それがこの要求の処理を担当するコンポーネントであるかどうかを判断します。担当のコンポーネントが見つかると、要求はこれに送信され、通常はこれによってビジネスロジックイベントが起動します。

このような抽象的な言い方をすると、担当のコンポーネントとは何だろうと疑問に思われるでしょう。担当のコンポーネントとは、要求処理時に呼び出された `acceptRequest()` メソッドから null 以外の結果を返すために `com.iplanet.jato.RequestHandler` を実装する最初のビューコンポーネントです。コンポーネントは通常、要求パラメータを調べ、このコンポーネントが処理すべきイベントに対応する名前と値のペアがこれらに含まれているかどうかを判断することで、このメソッドからオブジェクトを返すかどうかを決定します。このプロセスの詳細はこのマニュアルでは説明しませんが、理解すべき最も重要なことは、CommandField がこのプロセスの主要意思決定者であるということです。CommandField については、次の項で説明します。

CommandField

`CommandField` は、サーバーへの要求を開始するためにクライアントが起動できるページ上のコンポーネントに対応する特殊なビューです。理論的にはどの種類のビューも `CommandField` インタフェースを実装できますが、ビューによってはこれを実装してもあまり意味がないものがあります。このインタフェースは、ボタン表示フィールドなどの特定の種類のビューでのみ実装すべきです。たとえば、HTML では、ボタン (*<input type="submit">*) とリンク (*<a href="...">*) は、ユーザーがクリック (起動) することで別の要求がサーバーに送信されるコンポーネントであるため、Web アプリケーションフレームワークアプリケーションでは CommandField で表されます。

CommandField は、他のビューコンポーネントと同じように、ViewBean または他のコンテナに追加することができ、Web アプリケーションフレームワークコンポーネントライブラリにはボタンおよび HREF/リンク CommandField コンポーネントの両方が含まれています。他のコンポーネントと異なることは、フレームワークはこれらの種類のコンポーネントが特別なものであることを理解しているため、これらは特別に扱われる、という点です。具体的には、これらは要求処置時に、現在の要求のソースであるかどうかを判断するためにチェックされます。

このプロセスの最終結果として、たとえば、ユーザーが HTML アプリケーションのフォーム上のボタンをクリックした場合は、その要求の処理方法を決定するためにアプリケーションのボタンコンポーネントが呼び出されます。ボタンコンポーネントは、その要求のビジネスロジックを呼び出すために何をすべきかを親コンポーネントに伝える特殊なプロパティを備えています。具体的に言うと、起動されたボタンなどの CommandField は、現在の要求のために呼び出す必要がある、開発者が定義した

コマンドオブジェクトを指定します。または、CommandField で (代替オブジェクトを指定しないことで)、現在の要求を処理するためにデフォルトコマンドオブジェクトの呼び出しを指定することもできます。

コマンド記述子プロパティ

開発者は、フィールドの「要求ハンドラ」プロパティを介してフィールドが起動したときに呼び出すコマンドオブジェクトを指定します。CommandDescriptor は、実行時にコマンドクラスの特定のインスタンスを作成してそれを起動するために必要とされる情報を符号化している Web アプリケーションフレームワークオブジェクトです。CommandDescriptor では最低限でも、実行時にインスタンス化するコマンドクラスを指定します。(コマンドクラスは、対応する CommandField が起動した場合のみ作成されます。)

開発者はフィールドの「要求ハンドラ」プロパティの値を「デフォルト (handleRequest イベントをトリガー)」することで、要求を処理するためにデフォルトコマンドオブジェクトを使用するようにフレームワークに指示できます。

このコマンドオブジェクトの選択により、Web アプリケーションフレームワークビジネスロジックイベントは、CommandField の親の要求イベントハンドラメソッドを介して (デフォルトコマンドが使用される場合)、または開発者が定義したコマンドオブジェクトを介して、という 2 通りの方法で処理されます。以下の項で、これら両方の方法について説明します。

要求イベントメソッドハンドラ

要求イベントを処理するための最も一般的なアプローチは、要求イベントメソッドハンドラを使用する方法です。開発者は、対応する CommandField が起動したときに自動的に呼び出される、特定の命名規則と署名を持つメソッドを実装します。これは、クライアント側アプリケーションにイベントハンドラメソッドを実装するのとよく似ています。

このアプローチの利点は、理解と維持が簡単で、関連オブジェクトに合わせてローカライズしたコードを保持し、開発者がシンプルな手続き型ロジックを作成できることにあります。このアプローチの欠点は、このイベントロジックは異なるフィールドやページ間で再利用することができない、ということです。ただし、もちろんこれらのメソッドは、このように再利用可能な共通のメソッドを呼び出すことができます。

Web アプリケーションフレームワークでは、このイベントハンドラメソッドは、担当の CommandField の「親」に実装されます。たとえば、新しい ViewBean を作成し、ボタンコンポーネントを子としてそれに追加する場合は、ボタンコンポーネントの親である ViewBean に要求イベントハンドラメソッドを実装します。たとえば、新しい ViewBean を作成し、ボタンコンポーネントを子としてそれに追加する場合は、ボタンコンポーネントの親である ViewBean に要求イベントハンドラメソッドを実装します。

このメソッドは、他のどの場所にも実装できません。なぜなら、ここではコンテナ内にボタンコンポーネントの「インスタンス」を作成しますが、インスタンスには新しいメソッドを追加できないからです。この事実は、ContainerView という「種類」とその子コンポーネント「インスタンス」の間には密接な関係があり、コンテナとその子が連携して機能的なコンポーネントを形成することをはっきりと示しています。

要求イベント処理メソッドの署名は、以下のようになります。<childName> は、子コンポーネントのローカル (非限定) 名になります。

```
public void handle<childName>Request(RequestInvocationEvent
event)
```

たとえば、前述の例では、「submit」という名前のボタンコンポーネントを ViewBean に追加した場合は、ボタンからのイベントを処理するために ViewBean に以下のメソッドを実装する必要があります。

```
public void handleSubmitRequest(RequestInvocationEvent event)
```

(規則に従ったメソッド名を作成するために、子コンポーネントの名前は必要に応じて自動的に大文字に変換されます。)

このメソッドは手動で ContainerView クラスに追加することもできますが、Sun Java Studio の Web アプリケーションフレームワークモジュールを使用すれば、追加したコンポーネントを右クリックし、コンテキストメニューの「イベント」サブメニューからイベントを選択するだけで済みます。これによって、適切なメソッドスタブがクラスに追加されます。

重要：このイベント処理様式は、開発者が CommandField の「要求ハンドラ」プロパティに値を指定しない場合のみ使用されます。開発者が CommandField の記述子を指定した場合は、記述子に指定されているコマンドオブジェクトが要求を完全に処理するとフレームワークは想定します。IDE でビューにイベントハンドラスタブを追加できても、記述子が指定されている場合は起動されません。

イベントハンドラメソッドにビジネスロジックを実際に実装する方法については、以下の項を参照してください。

コマンドイベントハンドラ

要求イベントハンドラメソッドの代わりに、コマンドオブジェクトを直接実装し、フィールドの「要求ハンドラ」プロパティに値を指定することで、これを CommandField と直接関連付けることができます。

この方法の利点は、アプリケーション固有の動作やライブラリが提供する動作を再利用できること、ポイント & クリックのアプリケーションアセンブリ、そしてイベントハンドラメソッドを使用しないことでビューコンポーネントをよりすっきりしたものに保てることです。このアプローチの欠点は、コマンドクラスが増殖し (各 CommandField インスタンスにつき 1 つ作成される場合もある)、イベントを生成するオブジェクトからイベント処理ロジックが切り離され (コンポーネント動作のカプ

セル化の度合いの低下)、事前設計がより多く必要になる他、要求イベントメソッドハンドラと同じタスクを実行するのにより多くのコードが必要になる場合があることです。

このアプローチは、com.iplanet.jato.command.Command インタフェースを実装する Java クラスを作成するだけで使用できます (これは手動または Web アプリケーションフレームワークのウィザードを使って作成できます)。この作業が完了したら、CommandField の「要求ハンドラ」プロパティの「コマンドクラス名」プロパティにコマンドを指定できます。

コマンドオブジェクトにビジネスロジックを実際に実装する方法については、以下の項を参照してください。

イベント処理アプローチの選択

どちらの要求イベント処理アプローチを使用するかを決定する際は、アプリケーション要件を解決する上で役立つ方法を選ぶことが推奨されます。実際には、ほとんどのアプリケーションで両方の方法が使用されます。

一般的に、コンポーネント内にイベント処理ロジックをカプセル化したい場合、再利用するためにコンポーネントを作成する場合、またはイベント処理の各詳細の抽象化を意識しないで機能オブジェクトをスピーディーに作成したい場合は、要求イベントハンドラメソッドのほうが適しているでしょう。

反対に、イベント処理の詳細を一般的なクラスに抽象化したい場合、複雑なロジックを再利用する必要がある場合、またはアプリケーションに容易にアセンブルできる事前に作成された要求イベント処理ロジックを提供したい場合は、コマンドオブジェクトを使用します。

通常もう 1 つ考慮すべき点は、コマンドオブジェクトは作成するのに事前により多くの作業が必要ですが、再利用する際に役立つということです。要求イベントハンドラメソッドは迅速、簡単で、Swing などのクライアント側アプリケーション開発テクノロジを使ったことのある多くのアプリケーション開発者が馴れ親しんだ手段ですが、重要なロジックを再利用可能にするにはより多くの注意を払う必要があるでしょう。

結局は決定的なルールはなく、同じオブジェクト内でもどちらか 1 つの方法だけに決める必要はありません。Web アプリケーションフレームワークでは、フィールドごとに最適なアプローチを柔軟に使用できるため、当面の作業に最善のアプローチを自由に選択することが推奨されます。

イベント処理ロジックの記述

要求イベントの処理では、イベントハンドラメソッド、コマンドオブジェクトのどちらを使用するかに関わらず、アプリケーションに有用な動作を実行させるためのロジックを記述する必要があります。一般的に、イベント処理ロジックは以下のルーチンパターンに従います。

1. ビジネスロジックの実行
   a. ユーザーが送付した値の取得
   b. これらの値の処理
   c. 結果を使って表示するためのオブジェクトの準備
2. クライアントに対する応答の生成

通常、イベント処理ロジックは本質的に手続き型になります。各要求は主として単一のメソッド本体によって処理されますが、(他の Java コードと同様に) このロジックは必要なオブジェクトやメソッドへの参照を行う場合があります。

前方参照

ViewBean のような Web アプリケーションフレームワークオブジェクトに対しては、これから実行される表示サイクルに備えてオブジェクトを構成するために、イベント処理ロジック内で前方参照することが可能です。たとえば、現在のページのフィールドから値を取得して、イベント処理ロジック内の別のページ上の別のフィールドにこの値を設定できます。GUI ウィジェットがステートフルでアプリケーションのほとんどどこからでも随時参照できるクライアント側アプリケーション開発テクノロジを使用したことのある開発者にとっては、このテクニックは馴染みあるものです。

ビジネスロジック

Web アプリケーションフレームワーク API を使用すること以外は、ほとんどの開発者はすでにアプリケーションビジネスロジックの記述に関する一般的概念に精通していることでしょう。Web アプリケーションフレームワーク API の使い方は、39 ページの「プログラミングガイド」で説明されているため、ここではビジネスロジックの実行後にクライアント要求に応答する方法について説明します。

開発者は Web アプリケーションフレームワーク API を使用できることに加え、J2EE コンテナが提供する J2EE API をイベント処理ロジック内から自由に使用できます。たとえば、サーブレット、JDBC、JNDI、EJB、JavaMail、または他の J2EE テクノロジや API を、Web アプリケーションフレームワークイベントハンドラのスコープ内

でまったく問題なく使用できます。ただし、これらの API の使用を容易にする機能を Web アプリケーションフレームワークが提供している場合もあります。こうした機能の一部は、39 ページの「プログラミングガイド」で説明されています。

応答の生成

要求に応答してビジネスロジックを実行したら、開発者はクライアントに送信する応答を決定し、必要なテクニックに応じてこの応答を実際に生成する必要があります。クライアント要求に対しては、別のページやエラーメッセージを含めたある種の応答を常に送信します。ほとんどの場合、応答は、実行された要求から論理的に得られる別のページになります。たとえば、ショッピングカートに商品を追加するボタンをユーザーがクリックした場合、この応答はユーザーのショッピングカートの明細を示すページの表示になることでしょう。

あらゆる要求には、重点がそこから離れて、要求への応答の送信に重点が移り始める重要な地点があります。Web アプリケーションフレームワークでは、要求ライフサイクルにおけるこの地点は、一般的に送付サイクルから表示サイクルへの切り替えと表現されます。送付サイクル時、送付された値はビューコンポーネントとモデルにマップされ、イベントハンドラが呼び出されて要求を処理し、結果を準備します。表示サイクル時には、クライアントに対して結果が生成 (表示) されます。

ページフロー

よく行われている thin クライアント方式でのページの連鎖を、一般的にページフローと呼びます。ユーザーは最初のページでアクションを起動することで、あるページから別のページにフローします。このアクションによって、サーバーに要求が送信され、この結果、応答ページがクライアントに送信されます。

ページフローの扱い方には、さまざまなアプローチがあります。フレームワークによっては、ページフローを、フレームワークが直接扱うべき最上位の要件と考えているものがあります。たとえば、これらのフレームワークでは、応答の生成に使用する JSP URL をビジネスロジックが返すことを仕様にしている場合があります。こうしたアプローチで多く発生する問題は、応答生成メカニズムが脆弱になることで、つまりこれ以外のメカニズムを使って応答を送信するのは困難になります。このメカニズムで、応答を生成するために JSP か URL が起動されることが前提とされると、このアプローチと衝突する特定のアプリケーション要件を満たすことが難しくなります。たとえば、アプリケーション要求の結果として、バイナリ応答 (PDF ファイルなど) が送信される場合もあります。アプリケーションがクライアントに直接応答できないことがフレームワークで前提とされる場合は、こうしたシナリオを解決するのは困難です。

これとは対照的に、Web アプリケーションフレームワークでは簡単なページフローメカニズムを使用できますが、応答の生成方法は開発者の判断に委ねられています。具体的に言えば、Web アプリケーションフレームワーク開発者は、任意の ViewBean

への転送や任意の ViewBean の応答への包含、任意の URL への転送や包含、または標準サーブレット API を使ったクライアントへの結果の直接送信といったことを簡単に実行できます。

ViewBean の表示

結果を生成するための最も一般的なアプローチは、別の ViewBean (この用語は、応答 URL への要求の転送を可能にする、基礎となる J2EE サーブレット機能に基づいています) に要求を転送する方法です。Web アプリケーションフレームワークでは、これはよく ViewBean の表示と表現されます。ViewBean は通常、これらに関連付けられた JSP のようなピアリソース用の URL を備えているため、フレームワークは、どの ViewBean を使って応答を表示するかだけが明らかであれば、ViewBean の適切な JSP を実際に起動するための詳細を処理できます。

したがって、所定の要求のビジネスロジックを実行した後は、`ViewBeanManager` クラスを使って必要なターゲット ViewBean への参照を取得し、ViewBean の `forwardTo()` メソッドを呼び出すだけで、クライアントへの応答としてその ViewBean を生成できます。

```
public void handleSubmitRequest(RequestInvocationEvent event)
{
    // ここでビジネスロジックを実行 （これはまだ送付サイクル）
    ...

    // 結果として ViewBean を表示
    ViewBean targetViewBean=
        event.getRequestContext().getViewBeanManager().getViewBean(
            ResponseViewBean.class);
    targetViewBean.forwardTo(event.getRequestContext()); // 表示サイクルを開始

    // 仕上げ （これは表示サイクル時に起こる）
    ...
}
```

示されているように、送付サイクルと表示サイクルの切り替えは、開発者が `forwardTo()` を呼び出すことでトリガーされます。開発者は、イベントハンドラ時の任意の時点にこの呼び出しをトリガーできます。さらに、`forwardTo()` メソッドが最終的に復帰した後で、開発者は表示サイクル時に追加作業を実行できます (ただし、これはまれなことです)。

ここでは送付および表示フェーズに焦点をあてていますが、これらのフェーズは事実上論理的なものです。つまりこれらは、アプリケーションに対する 1 つの要求の局面に過ぎません。

URL と並列コンテンツの表示

ViewBean はどのようにして、表示に使用する JSP を知るのでしょうか。ViewBean では、2 つのメソッドを使って、`forwardTo()` が呼び出されたときに生成のために使用する URL を特定できます。

1 番目のメソッドは `ViewBean.getDisplayURL()` です。このメソッドは、サーブレット API RequestDispatcher 機構を使って、転送先の URL を含む文字列を返します。(この URL は JSP を呼び出す必要はありません。その代わりに、データを動的に表示する必要がない場合は静的 HTML ページまたは非 JSP の動的生成メカニズムを使って書き込まれたページを呼び出すことができます。ただし実際には、大部分の表示 URL は、簡単に利用できる J2EE メカニズムである JSP を参照します。)

表示 URL の値は、ViewBean のみが決定できます。たとえば ViewBean は、クライアントのロケールといった要求内の情報を使って、スペイン語とフランス語という 2 種類の URL のどちらかに決定します。コアの `BasicViewBean` クラスの実装では、開発者が `getDisplayURL()` メソッドの代替ロジックを提供しないと、メソッドは単に `getDefaultDisplayURL()` メソッドの値を返します。この値は、開発者がコードで設定することも、または ViewBean ノードの下の JSP を選択して右クリックし、「デフォルトとして設定」を選択することで、IDE で自動的に設定することもできます。

一連の URL から特定の URL を決定する ViewBean の能力は、Web アプリケーションフレームワークの並列コンテンツ機能です。同じ動的データを何種類かの方法で生成したい状況はよくありますが、並列コンテンツはこのような場合に使用できます。単一の JSP を使用した場合、多くの条件付けをしたり、JSP から 静的コンテンツを取得したりしなくては、これは実行できません。これらの方法を使用すると、通常は生産性や保守の問題が起こります。このために Web アプリケーションフレームワークでは、複数 JSP (または他の URL) を同じ ViewBean に関連付ける機能を提供しています。これによって開発者は、基本的には同じ動的データを根本的に別の方法で表現する「並列」ページを提供できます。

この機能の 1 つの使い方は、静的コンテンツを含むページを異なる言語で作成することです。たとえば、英語と日本語の両方で ViewBean データを表すことが可能な単一の JSP を作成するのは困難でしょう。文字セットや言語が異なるだけでなく、日本語ページのレイアウトは英語ページのレイアウトと相当異なります。この場合は、1 つの非常に複雑な JSP を記述する代わりに、2 つのはるかに簡単な JSP を記述できます。それぞれの JSP は同じ ViewBean を参照しますが、異なる方法でデータを表します。ViewBean は、ロケールやユーザーの言語設定を調べることで自身の表示に使用する JSP を決定し、その `getDisplayURL()` メソッドから適切な URL を返します。

並列コンテンツの別の使い方は、さまざまな種類のデバイスをターゲットとすることです。たとえば、ViewBean を HTML ドキュメントまたは WML ドキュメントとして生成できます。これらのマークアップ言語には非互換の相違点があるため、両方を生成する共通の JSP は作成できません。しかし開発者は、代わりにそれぞれのマークアップ言語で JSP を作成し、どちらを使用するかを ViewBean に決定させることができます。ここでも ViewBean は、ある情報を使って、その `getDisplayURL()` メソッドから返す URL を決定します。

この意思決定機構は、コア BasicViewBean クラスに構築済みでいつでも使用できる、というわけではありません。この理由は、他の URL ではなく特定の URL を表示する決定を行う要因となるアプリケーション固有の変数の範囲を Web アプリケーションフレームワークが把握することは不可能だからです さらに、さまざまな JSP バージョン (英語と日本語など) を区別するための URL スキーマは、アプリケーション開発者が決定すべき事項だからです。これらを始めとする理由から、この機構は規定されていません。

したがって並列コンテンツが必要な場合は、開発者は getDispayURL() メソッドをオーバーライドして必要な意思決定ロジックを指定する必要があります。また、アプリケーション開発者とコンポーネントプロバイダは、並列 JSP を決定して配置するための特定セットのヒューリスティックスとスキーマを組み込んだ ViewBean コンポーネントを提供する必要があります。こうしたコンポーネントのユーザーは、その決定を完全に把握して特定のメカニズムを受け入れ、必要に応じて異なるメカニズムを備えた異なるコンポーネントを使用できます。

他のオブジェクトが ViewBean の setDefaultDisplayURL() メソッドを呼び出して、その要求時に ViewBean が使用する URL を変更することも可能ですが、この方法は、このメソッドのセマンティクスに違反し、維持が困難なオブジェクト間の依存を生じるため、推奨されません。

J2EE の制限事項

サーブレット仕様には、すでに要求がクライアントにコミットされている場合は、所定要求内で RequestDispatcher.forward() を 2 回以上呼び出すのは規則違反である、と示されています。これは ViewBean.forwardTo() メソッドを意味しています。

この制限事項は通常、1 つの要求内で ViewBean.forwardTo() を何回も呼び出すことはできない、ということを意味します。唯一の例外は、現在の応答がまだクライアントにコミットされていない場合です。応答バッファがいっぱいになり、他の応答データ用のスペースを作るためにフラッシュする必要が出たときに、応答はクライアントの出力ストリームにコミットされます。クライアントへの最初のフラッシュの後で応答はコミットされたとみなされ、再度転送しようとするとサーブレットコンテナによって IllegalStateException がスローされます。JSP を使って応答を生成する場合は、ページの属性を設定することで応答バッファを構成できるため、開発者はこの制限事項をある程度制御できます。

ただし、所定の要求内で 2 回以上転送することは滅多になく、開発者は 1 度だけ応答を選択します。このため、この制限事項は、ほとんどのアプリケーションには影響を与えません。例外として考えられるのは、転送後の表示サイクル時にアプリケーションでエラーが発生し、エラーページを表示する必要がある場合です。こうした状況が発生した場合は、このような事態に対応できるだけの十分な大きさに応答バッファサイズを設定してください。

第3章

プログラミングガイド

この章では、一般的なプログラミングシナリオを示し、Web アプリケーションフレームワークのいくつかの基本的なオブジェクトの使い方を説明します。

概して、ここで提供されている情報は、Web アプリケーションフレームワーク JavaDoc を補足するものです。API の使い方の詳細は、JavaDoc を参照してください。

RequestContext の使用

RequestContext は、要求時に Web アプリケーションフレームワーク関連のサービスを提供する一次オブジェクトです。これは事実上 Web アプリケーションフレームワーク要求内で随時どこからでも取得でき、J2EE および Web アプリケーションフレームワーク機能の両方にアクセスするために使用されます。

RequestContext の取得

現在の RequestContext インスタンスは、主として 2 通りの方法で取得できます。第一に、コードが書かれている ViewBean やモデルといった現在のオブジェクトが `com.iplanet.jato.RequestParticipant` インタフェースを実装している場合、ほぼそのオブジェクトはコードに制御が渡される前にすでに RequestContext のインスタンスを持っています。ViewBeanManager および ModelManager はともに、`RequestParticipant` インタフェースを実装している ViewBean とモデルを呼び出し元に返す前に、それらのオブジェクトに自動的に RequestContext を設定します。具体的には BasicViewBean、BasicContainerView、BasicTiledView、および BasicTreeView はどれも、RequestContext を取得するためにこれらのオブジェクト内の事実上どの時点でも使用できる `getRequestContext()` というメソッドを持っています。ただし、RequestContext は完全に構築されていないオブジェクトには設定できないため、これらのオブジェクトの 1 つを構築している間はこのメソッドは使用できないことに注意してください。

RequestContext (およびその一次サブオブジェクトの一部) を取得するための 2 番目の主なテクニックは、com.iplanet.jato.RequestManager クラスの静的メソッドを使用する方法です。

以下の表に、これらのメソッドとその説明を示します。

メソッド	説明
getRequestContext()	現在の要求の RequestContext オブジェクトを返します。このメソッドは、要求のスコープ内でのみ使用できます。
getRequest()	現在の要求の javax.servlet.http.HttpServletRequest オブジェクトを返します。このメソッドは、要求のスコープ内でのみ使用できます。
getResponse()	現在の要求の javax.servlet.http.HttpServletResponse オブジェクトを返します。このメソッドは、要求のスコープ内でのみ使用できます。
getSession()	現在の要求の javax.servlet.http.HttpSession オブジェクトを返します。このメソッドは、要求のスコープ内でのみ使用できます。

これらの静的メソッドを使用して、要求のコンテキスト内にある任意のオブジェクトやメソッドの内部から、Web アプリケーションフレームワーク RequestContext を取得できます。これらのメソッドは、オブジェクトのメソッド呼び出しに RequestContext パラメータを追加する必要がなくなるという点で最も有用です。これらのメソッドを使用しても、特にスレッド同期化を起こすことはなく、アプリケーションのパフォーマンスに影響を与えることはありません。

これらのメソッドのいずれも、要求のスコープ外では使用できません。たとえば、これらを使ってオブジェクトの静的初期化子内、またはサーブレットの init() メソッド内のセッションオブジェクトを取得することはできません。

サーブレットイベントメソッドはパラメータとして RequestContext を指定し、他のアプリケーションイベントの起動時に使用されるイベントオブジェクトの多くは RequestContext メンバーを持っています。前述の他のテクニックの 1 つを使用する代わりに、このように提供されるインスタンスを使用できます。

サーブレット要求および応答オブジェクトの取得

Web アプリケーションフレームワークアプリケーションは最終的にはサーブレットベースのアプリケーションであるため、これは現在の要求の javax.servet.http.HttpServletRequest および javax.servet.http.HttpServletResponse オブジェクトにアクセスします。要求および応答オブジェクトは、RequestContext の getRequest() および getResponse() メソッドを使って取得できます。また、これらのオブジェクトは、前述の RequestManager メソッドを介して便利に取得することもできます。

要求および応答オブジェクトは、2、3 の点だけに気を付ければ (主に応答オブジェクトに関するもので、他の箇所で説明しています)、任意のサーブレットまたは JSP ベースの J2EE アプリケーション内で使用する場合と同じように使用できます。

セッションオブジェクトの取得

`javax.servlet.http.HttpSession` オブジェクトは、現在の要求の HttpServletRequest オブジェクトがあれば取得できます (前述を参照)。また、前述の RequestManager メソッド `getSession()` を介して直接取得することもできます。

利用可能な他のオブジェクト

前述のオブジェクトと他の 2、3 の便利なオブジェクトに加え、RequestContext は Web アプリケーションフレームワーク固有の 3 つの主なマネージャオブジェクトへのアクセスも提供します。

以下の表に、これら 3 つの主要オブジェクトを示します。

オブジェクト	説明
`com.iplanet.jato.ViewBeanManager`	getViewBeanManager() への呼び出しにより取得します。 ViewBeanManager は、現在の要求内の ViewBean インスタンスの管理を支援します。すべての ViewBean は、このオブジェクトを介して取得する必要があります。
`com.iplanet.jato.ModelManager`	getModelManager() への呼び出しにより取得します。 ModelManager は、現在の要求内のモデルインスタンスの管理を支援します。すべてのモデルは、このオブジェクト (または ModelReference) を介して取得する必要があります。
`com.iplanet.jato.SQLConnectionManager`	getSQLConnectionManager() への呼び出しにより取得します。 JDBC を使って RDBMS バックエンドに通信するアプリケーションでは、SQLConnectionManager が J2EE コンテナのデータベース接続サポートの上に薄いユーティリティレイヤーを提供します。SQLConnectionManager の使用は必須ではありませんが、推奨されます。

RequestCompletionListener インタフェース

現在の要求の完了通知に関係するオブジェクトは、RequestCompletionListener として自身を RequestContext に登録できます。登録されたリスナーは、現在の要求の結果にかかわらず (たとえば、要求が正常に完了したかエラー終了したかに関係なく)、現在の要求の処理終了時に通知を受け取ります。

オブジェクトはこのメカニズムを使って、セッションへの状態の保存、開いている要求スコープのリソースのクローズなど、タスクが出力ストリームに影響を与えない限り、事実上あらゆる最終タスクを実行できます。オブジェクトが関係する場合は、各要求で RequestCompletionListener として自己を登録する必要があります。

メッセージライターの使用

RequestContext は、メッセージライターという有用なツールも提供します。メッセージライターは `java.io.PrintWriter` で、要求経過時にその内容が蓄積され、クライアントにページが送信される直前に生成されるページに追加されます。このメカニズムは非サーバーベースのアプリケーションにおけるコンソールのように動作し、デバッグに非常に役立ちます。関係するオブジェクトは、RequestContext の `getMessageWriter()` メソッドによってメッセージライターに書き込むことができます。ViewBean はまた、`appMessage()` という便利なメソッドも備えています。これを使用すると、1 つのステップでメッセージライターを取得して使用できます。

メッセージライターの出力は必然的に HTML 固有のものになるため、異なるマークアップ言語で生成されるページと非互換になる場合があります。さらに、メッセージライターは、ViewBean JSP タグによりページ末尾に自動的に追加されるため、他の非 JSP メカニズムを使ってページを生成する場合は表示されません。

メッセージライターに書き込まれたメッセージの表示は、`web.xml` ファイルの構成パラメータを使ってオンやオフにできます。このため、開発時にはメッセージをオンにして、配備時にはオフにすることができます。詳細は、75 ページの「アプリケーションの配備」を参照してください。

ViewBeanManager の使用

アプリケーションの 2 つの異なる部分が 1 つの ViewBean インスタンスを使って動作しようとする場合、その 2 つの部分は常に同じインスタンスを利用しようとします。1 つの要求内で ViewBean は単独であり、このことは、あらゆる要求で ViewBean は 1 種類につき 1 つのインスタンスしか存在しないことを意味します。単独インスタンスを保証するために、ViewBean は `com.iplanet.jato.ViewBeanManager` のインスタンスである ViewBeanManager によって管理されます。

ViewBeanManager は、getViewBeanManager() メソッドを呼び出すことにより、RequestContext から利用できます。それぞれの要求は、新しい ViewBeanManager インスタンスを受け取ります。ViewBeanManager は RequestContext を介して利用でき、RequestContext は前述したいくつかのテクニックの 1 つを使って広く利用できるため、ViewBean インスタンスは Web アプリケーションフレームワーク固有のオブジェクトやメソッド内だけでなく、アプリケーションコードのどこからでも広く取得できます。

ViewBeanManager を取得したら、必要な ViewBean のクラスを指定して getViewBean() メソッドを呼び出すことで ViewBean インスタンスを簡単に取得できます。たとえば、以下のコードはログイン ViewBean の単独インスタンスを返します。

```
requestContext.getViewBeanManager().getViewBean(Login.class);
```

ViewBeanManager はまた、クラス名で ViewBean を返すメソッドも備えていますが、このメソッドは前述のメソッドが持つコンパイル時の安全性を提供しません。

アプリケーション開発者は通常 getLocalViewBean() メソッドを使用しませんが、これは少し説明しておく必要があります。このメソッドは、指定の論理名 (この名前は ViewBean のローカル名とも呼ばれます) に対応する ViewBean インスタンスを取得します。ローカル ViewBean 名は、「ViewBean」接尾辞のない ViewBean クラスの非限定名です。たとえば、モジュールに ViewBean クラス com.foo.Page1 がある場合は、ローカル ViewBean 名は単純に Page1 になります。ローカル ViewBean 名は現在のモジュールの基本パッケージ名によって自動的に限定されるため、ローカル名は、そのサーブレットが最初に現在の要求を処理したモジュール内の ViewBean への参照を取得するためにのみ使用できます。

ModelManager の使用

アプリケーションの 2 つの異なる部分が 1 つの ViewBean インスタンスを使って動作しようとする場合、その 2 つの部分は常に同じインスタンスを利用しようとします。要求内のモデルインスタンスの共有を可能にするために、RequestContext には ModelManager オブジェクトが含まれています。このオブジェクトは、ほとんどすべてのモデルインスタンスの取得に使用します (考えられる例外を以下の項に示します)。

ModelManager は、その種類とオプションのインスタンス名に基づいてモデルインスタンスを検索するための便利なメカニズムを提供します。同じ要求内の検索では、いつでも、種類とインスタンス名で指定される同じモデルインスタンスが返されます。特にセッションに格納されるモデル (以下の項を参照) を除いて、ModelManager によって取得されるモデルは、各要求でインスタンス化されて共有されます。セッションへのオブジェクトの格納はリソースを消費する操作であるため、要求間でモデルデータを格納する必要がなく、必要なときにモデルデータを再度簡単に取得できれば、これは通常アプリケーションにとってより効率的です。

ModelManager の主要なメソッドは getModel() およびその変形です。一般に、getModel() のパラメータとして、モデルクラス名を指定するのではなく、モデルクラスを指定することが推奨されます。これによって、コンパイル時にメソッド呼び出しをチェックすることが可能になります。

ModelManager からモデルを取り出す際には、インスタンス名を指定することも可能ですが、これはほとんど行われません。インスタンス名を指定しない getModel() メソッドの変形の 1 つを使用すると、ModelManager はモデルのクラス名に基づいてデフォルトインスタンス名を使用します。一般的に、ほとんどのアプリケーションでは、これで十分です。ただし、同じ要求内で同じモデルの種類の 2 つの異なるインスタンスを使用する場合、この機能が有用な場合があります。

セッションにおけるモデルの取得と保存

HTTP セッションの要求間でモデルデータを格納する必要がある場合があります。ModelManager を使用すると、セッションで検索しながら、モデルを簡単に取得および保存できます。

一般にアプリケーションは、要求間のモデルデータを格納しないように作成します。なぜなら、セッションにおけるデータの格納は一般的に比較的リソースを消費し、過度に使用するとアプリケーションのスケーラビリティが制限されるからです。

1 つか 2 つの付加的なセッション関連の boolean 型のパラメータを指定できる getModel() の変形がいくつかあります。1 番目は lookInSession パラメータです。これが True の場合は、ModelManager はまずそのローカルモデルキャッシュをチェックし、次に HTTP セッションをチェックしてから、新しいモデルインスタンスをインスタンス化します。2 番目のパラメータは storeInSession です。これが True の場合、ModelManager は、セッションで設定するために返されたモデルインスタンスをスケジュールします。J2EE コンテナのセッション実装にかかわらず、現在の要求におけるオブジェクトへの変更がすべてセッションに書き込まれるように、現在の要求の終了までは、モデルインスタンスは実際にはセッションに設定されません。

モデルインスタンスを手動でセッションに追加したりセッションから削除する場合は、ModelManager の addToSession() または removeFromSession() メソッドをモデルインスタンスを指定して呼び出します。もちろん、セッション API を使い手動でセッションにモデルを設定することもできますが、addToSession() メソッドを使用すると、セッションに追加されたオブジェクトに後で lookInSession パラメータを使ってアクセスすることが可能になります。

ModelTypeMap

この機能は最近の Web アプリケーションフレームワークバージョンではほとんど使用されず、IDE ではまったく使用されませんが、少し説明しておきます。この機能をアプリケーションで使用するかどうかは開発者の判断ですが、具体的な効果がある場合のみ使用すべきだということを念頭に置いてください。

Web アプリケーションフレームワーク設計サイクルの早期には、モデルインタフェースだけを指定することでモデルインスタンス (実装) を検索できる機能が有用だと考えられました。アプリケーション開発者はこの機能を使用すると、アプリケーションコードにおけるモデルの実装詳細をもとに作業するのではなく、実装に中立なモデルインタフェースを作成し、アプリケーション全体を通してこれらだけを使って作業を行うことができます。これはビュー層とモデル層間に最大限の疎結合を提供し、開発者はこれを使ってビュー層にまったく影響を与えずに異なるモデル実装をプラグインできました。

このため、ModelManager 内でインタフェース型を実装型にマッピングするために、`com.iplanet.jato.ModelTypeMap` クラスが追加されました。開発者が ModelManager の `getModel()` メソッドを呼び出すと、ModelTypeMap を使って指定のモデルの種類が実装型に変換されます。マッピングを指定しないと、元の種類が変更なしで返されます。これは本来フォールバック機構を提供し、開発者はこれを通して実装型を指定できると同時に有効なモデルインスタンスを取得できました。

実際には、多くの開発者にとって ModelTypeMap 機能は制限があって使いにくかったため、今はほとんど使用されていません。しかし、これは今でも ModelManger アーキテクチャに前提とされる部分であり、アプリケーションサーブレットは今も ModelTypeMap の作成時に空の ModelTypeMap のインスタンスを ModelManager に提供します。ModelTypeMap がアプリケーションに有用と考えられるまれな場合を除いて、これは基本的に単なるフレームワークインフラストラクチャとして無視できます。

ModelTypeMap を使用する場合は、静的初期化子をアプリケーションサーブレットクラスに追加し、必要な各マッピングについて 1 つのエントリを ModelTypeMap に追加します。次に例を示します。

```
ModeTypeMapBase.addModelInterfaceMapping(
     apppkg.modulepkg.CustomerModel.class,
     apppkg.modulepkg.CustomerModelImpl.class);
```

次に、アプリケーションコード内で、`ModelManager.getModel(CustomerModel.class)` を呼び出してモデルを取得しますが、`CustomerModelImpl` のインスタンスを再度取得します。

ModelManager 使用の例外

Web アプリケーションフレームワークで提供されている大部分のモデルの種類を有用なものにするには、何らかの初期化を行う必要があります。これらのモデルを構成するための通常のアプローチは、モデルがインスタンス化されたときにそれを適切に構成するサブクラスを作成することです。ただし、(特に IDE は、現在一部のモデルの種類を拡張可能コンポーネントとしてサポートしていないため) 一部のモデルの種類は、構成済みインスタンスとして使用するほうが簡単な場合もあります。これらのモデルは、`com.iplanet.jato.model.BeanAdapterModel`、`com.iplanet.jato.model.SessionModel`、`com.iplanet.jato.model.MultipleModelAdapter`、および `com.iplanet.jato.model.ResourceBundleModel` です。

これらのモデルはコード内でインスタンス化することが有用な場合もありますが、その場合は `ModelManager.addToSession()` メソッドを使ってこれらをセッションに追加してください。これは、1 度だけ初期化する必要があるが長期にわたって存続するモデルに最適で、これにより以降の要求でこれらのモデルを ModelManager からシームレスに取得できるようになるため、DisplayField または他のビューにこれらを簡単にバインドできます。

SQLConnectionManager の使用

Web アプリケーションフレームワークアプリケーションは、多くの場合、少なくともアプリケーションデータの格納に RDBMS を使用します。J2EE では、J2EE アプリケーション内からスケーラブルにデータベース接続を取得できるように、JDBC 標準拡張 API が提供されています。ただし、実際に現実のアプリケーションを開発する際は、注意事項がいくつかあります。

このため、Web アプリケーションフレームワークの SQLConnectionManager には、開発時のアプリケーションの存続期間中、より簡単にデータベース接続を使って作業するのに役立つ付加的な機能が追加されています。特に、開発時のアプリケーションは、データベース接続プールをすぐにはサポートしないコンテナで実行されることがよくあります。さらに、接続プールを使用するには、アプリケーションの配備とは別に、コンテナでそのプールを追加構成する必要があります。このため、単にユーティリティまたはデモアプリケーションを配備するのが難しくなります。

SQLConnectionManager は、JDBC 接続を取得するタスクに thin 抽象化オブジェクトを追加します。これにより、データソース名を他の値にマッピングして、JNDI と `java.sql.DriverManager` 接続検索テクニックを簡単に切り替える機能が有用な場合に、これらを行うことが可能になります。

SQLConnectionManager から接続を取得するときは、`jdbc/mydb` など、JDBC 標準拡張規則に従って JDNI のようなデータソース名を指定します。
SQLConnectionManager が現在 JNDI 検索を使って JDBC 接続を取得している場合

は、これはこの名前を検索のキーとして使用します。一方、SQLConnectionManager が DriverManager を使って JDBC 接続を取得している場合は、これはこの名前を使ってそのデータソースマッピング表でローカル検索を実行して、接続の取得に使用できる JDBC URL を取得します。

現在の接続検索モードは、SQLConnectionManager クラスの `setUsingJNDI()` メソッドを使って設定します。通常このメソッドは、アプリケーションパッケージの `SQLConnectionManagerImpl` クラスの静的初期化子から呼び出されます。このパラメータに True 値を指定すると、SQLConnectionManager は標準 JNDI/JDBC 接続検索テクニックを使って JDBC 接続を検索します。これは、データベース接続プールがすでに構成されて J2EE コンテナに登録済みであると想定します。

`setUsingJNDI()` への呼び出しで False 値を指定すると、SQLConnectionManager は自身のローカルデータソースマッピングセットを使って、`java.sql.DriverManager` に送信できる JDBC URL を取得し、接続を取得しようとします。このテクニックは接続プールを使用するより少なくとも 1 桁は効率が悪いものですが、開発時、またはコンテナの追加構成なしでスタンドアロンで動作する必要がある、いくつかのクラスのアプリケーションでは一般的に容認できるものです。

DriverManager 接続検索を使用するときは、`SQLConnectionManagerBase.addDataSourceMapping()` メソッドを使ってデータソースマッピングを追加できます。このマッピングで、物理 JDBC URL にマッピングする論理データソース名を指定します。以下に、PointBase URL を使った例を示します。

```
SQLConnectionManagerBase.addDataSourceMapping("jdbc/sample",
    "jdbc:PointBase://localhost:9092/sample");
```

`addDataSourceMapping()` メソッドは通常アプリケーションパッケージのアプリケーションサーブレットクラスの静的初期化子から呼び出されるため、コンテナによってアプリケーションが読み込まれたときにマッピングが初期化されます (理論的には、マッピングは後で追加できますが、アプリケーションの存続期間中は一貫している必要があるため、後で追加する理由はほとんどありません)。JNDI 接続検索を使用するときは、データソースマッピングを追加する必要はありません。ただし、何らかの理由で、あるデータソース名を別のデータソース名にマッピングする場合は、マッピングを追加できます。

本稼働のアプリケーションでは、どのような場合も、決して DriverManager を使って JDBC 接続を取得しないでください。このような使い方をすると、接続を使用するたびに新しいデータベース接続が開かれ、アプリケーションのパフォーマンスとスケーラビリティに深刻な問題が発生します。アプリケーションを本稼働用に配備するときは、必ずコンテナのデータベース接続プールとともに SQLConnectionManager の JNDI 検索メカニズムを使用してください。

アプリケーションでは、その `getConnection()` または `obtainConnection()` メソッドを使って、SQLConnectionManager から JDBC 接続を取得できます。static obtainConnection() メソッドは、アプリケーション初期化時など、要求スコープ外の

接続を取得するために使用します。また、必要であれば SQLConnectionManager をバイパスして JNDI 検索または DriverManager を直接使うこともできますが、com.iplanet.jato.model.sql.QueryModelBase といった JDBC を使用する Web アプリケーションフレームワーククラスでは、常に SQLConnectionManager を使ってデータベース接続を取得します。

RequestManager の使用

RequestManager は、要求スコープの主要オブジェクトを取得するために使用できる少数の static メソッドを提供します。

以下の表に、これらのメソッドを示します。

メソッド	説明
getRequestContext()	現在の要求の RequestContext オブジェクトを返します。このメソッドは要求のスコープ内でのみ使用できます。
getRequest()	現在の要求の javax.servlet.http.HttpServletRequest オブジェクトを返します。このメソッドは、要求のスコープ内でのみ使用できます。
getResponse()	現在の要求の javax.servlet.http.HttpServletResponse オブジェクトを返します。このメソッドは、要求のスコープ内でのみ使用できます。
getSession()	現在の要求の javax.servlet.http.HttpSession オブジェクトを返します。このメソッドは、要求のスコープ内でのみ使用できます。
getHandlingServlet()	最初に現在の要求を処理した javax.servlet.Servlet インスタンスを返します。近い将来、このサーブレットは com.iplanet.jato.ApplicationServletBase のサブクラスになる予定です。このメソッドは、要求のスコープ内でのみ使用できます。

注 – 「RequestManager」という名前は、Web アプリケーションフレームワークの将来バージョンに追加される予定の多数の機能のプレースホルダとなるクラスに使用されています。単なる static ユーティリティメソッドのコレクションとしての現在の状態は、一時的なものです。

ログ

Web アプリケーションフレームワークのログサポートは、それ自体ほとんど最低限の機能しか持たない標準 ServletContext ベースのログ機能の上に構築されています。したがって Web アプリケーションフレームワークのログ機能は、この既存の基礎メカニズムに少しの機能を追加することだけを目的としており、Log4J や JDK 1.4 のログパッケージが提供するような全機能を完備したログソリューションを提供することは意図していません。J2EE コンテナのネイティブログ機能を望む開発者向けに軽量で手頃に使えることを意図しています。

Web アプリケーションフレームワークのログには、`com.iplanet.jato.Log` クラスの static メソッドを使ってアクセスできます。このクラスはこれらのメソッドに加えて、ログレベルに基づいてメッセージをフィルタし、標準出力にログをエコーする機能も提供します。

メッセージのログ

メッセージをログに記録するには、オプションとしてログレベルパラメータを指定して、`com.iplanet.jato.Log` クラスの各種 static `log()` メソッドの 1 つを呼び出します。ログレベルを指定した場合、`Log` クラスは現在有効なログレベルに基づいてそのメッセージをログに記録すべきかどうかを判断します。ログレベルが指定されていない場合は、現在有効なログレベルは考慮されず、メッセージは常にログとして記録されます。

通過を許可されたメッセージは ServletContext の各種 `log()` メソッドを介してコンテナに送信され、通常はコンテナのログとコンソールの両方または一方に表示されます。

ログレベル

基礎の ServletContext ログメカニズムに追加されている主な付加価値は、レベルを使ってメッセージをログとして記録し、これらのレベルによって現在のログ出力をフィルタする機能です。ログレベルは、いくつかの大きなカテゴリに分類されます。

以下の表に、これらログレベルのカテゴリを示します。

ログレベルカテゴリ	説明
エラーレベル	アプリケーションのエラーや警告を指定するレベル。WARNING、ERROR、および CRITICAL レベルが含まれます。このカテゴリの全レベルは ANY_ERROR レベルを通して有効にできます。
デバッグレベル	情報通知またはデバッグのために開発者が使用するレベル。STANDARD、TERSE_DEBUG (より少ない情報)、および VERBOSE_DEBUG (より多くの情報) レベルが含まれます。このカテゴリの全レベルは ANY_DEBUG レベルを通して有効にできます。
トレースレベル	要求の実行をトレースするために開発者や Web アプリケーションフレームワークによって使用されるレベル。JATO_TRACE、JATO_QOS_TRACE、および APP_TRACE レベルが含まれます。このカテゴリの全レベルは ANY_TRACE レベルを通して有効にできます。JATO_TRACE および JATO_QOS_TRACE レベルは開発者が使用するためのものではなく、フレームワークそのものが記録したトレース情報を表示するために使用されるだけです。
ユーザーレベル	開発者が必要に応じて任意に定義して使用するレベル。これらの各レベルの意味は、アプリケーションと開発者が定義します。USER_LEVEL_1、USER_LEVEL_2、および USER_LEVEL_3 レベルが含まれます。このカテゴリの全レベルは ANY_USER_LEVEL レベルを通して有効にできます。

JATO_TRACE および JATO_QOS_TRACE ログレベルを除き、すべてのログレベルは主として開発者が使用するためのものです。つまり、コア実行時は通常、これらのログレベルのいずれでもログを記録しません (一部のエラーレベルを除きます)。ただし、他の作成者が作成した Web アプリケーションフレームワークコンポーネントはこれらのレベルでログに記録する可能性があるため、常にこれらを完全に制御できるとは限りません。この 1 つの例外は、いくつかの USER_LEVEL_* ログレベルです。これらは現在のアプリケーション専用に予約される決まりになっています。公開後、配布可能コンポーネントはこれらのレベルを使用すべきではありません。

Log.setEnabledLevels() への呼び出し内で複数レベルの論理和をとることにより、一度に複数のログレベルを有効にできます。たとえば、以下のコードを使用すると、複数のログレベルを一度に有効にできます。

```
Log.setEnabledLevels( STANDARD | ERROR | CRITICAL | JATO_TRACE );
```

あるレベルが現在有効かどうかをチェックするには、チェックするレベルを指定して isLevelEnabled() メソッドを呼び出します。さらに、各種 ANY_* ログレベルや DEFAULT_LOG_LEVELS レベルのように、自動的に他のレベルをいくつか含む便利なログレベルも数個あります。

ログレベルは、アプリケーションコードやサーブレットの init() メソッド、静的初期化子、あるいは基本的には、実行時に Log クラスに静的にアクセスできる任意の場所からいつでも変更できます。ただしログ設定は、配備されたアプリケーションに

つき 1 セットだけ指定することができ、現在の設定は現在実行中のすべての要求スレッドが共有します。したがって、要求ごとにログレベルを変更しても通常は意味がありません。

Web アプリケーションフレームワークのログ機能は基礎の J2EE ログ機能のみを使って最低限の要件を満たすことを目的としているため、この機能を大幅に改良する計画は今のところありません。アプリケーションでより充実したログ機能が必要な場合は、Log4J または JDK 1.4 の組み込みログパッケージを使用してください。

標準出力へのログ

J2EE コンテナのログファイルには簡単にアクセスできないこともあるため、`Log` クラスは ServletContext ログに送信された内容をコンソールの `System.out` ストリームにエコーする機能を備えています。現在のコンテナのコンソールは開発者のマシンですぐに表示できるため、この機能は開発時に役に立つことがよくあります。

ログメッセージの強調

Web アプリケーションフレームワークログは J2EE コンテナのログに出力され、J2EE コンテナのログにはおそらく他のログメッセージも出力されるため、`Log` クラスでは `setMessagePrefix()` メソッドによってメッセージ接頭辞を設定する機能を提供しています。この接頭辞は、`Log` クラスを介してログとして記録されたすべてのメッセージに追加されます。適切な接頭辞を選択することで、Web アプリケーションフレームワークログメッセージを同じログ内の他のメッセージより強調させることができます。デフォルトの接頭辞は、3 つのダッシュ (---) の後に空白文字が続きます。

値の操作

以下の項では、Web アプリケーションフレームワークアプリケーションでの値の操作方法をいくつか説明します。

DisplayField 値の操作

値を取得するための 1 番目の最も一般的なアプローチは、DisplayField から直接取得する方法です。たとえば、テキストフィールドへの参照を取得してからその `getValue()` メソッドを呼び出すことで、テキストフィールドにその値を問い合わ

せることができます。このアプローチでは、ビュー開発者は構築しているビュー階層と直接対話することができ、通常はデータを取得するために必要な全情報が 1 つのファイルに含まれているため、これはビュー開発者にとって便利な方法です。

DisplayField または他の任意のビュー (ViewBean を除く) への参照は、その子ビューコンポーネントの非限定ローカル名を指定してその親の `getChild()` メソッドを呼び出すことで取得できます。(このメソッドは `ContainerView` インタフェースで定義されるため、ViewBean を含むすべての種類のコンテナビューに存在します。) `getChild()` は単なる表示フィールドではなくビューのインスタンスを返すため、使用可能な型に結果をキャストしてからこれを使用する必要があります。

以下に、ViewBean 内から子の BasicDisplayField を取得してその値を取得する例を示します。

```
View childView=getChild("textField1");
Object value=((BasicDisplayField)childView).getValue();
```

このテクニックでは大量のキャストが必要になる可能性があるため、コア ContainerView 実装 (BasicContainerView、BasicTiledView など) 内では、以下のような便利なメソッドを使用できます。

```
DisplayField field=getDisplayField("textField1");
Object value=field.getValue();
```

返される値は DisplayField インスタンスであるため、特定のコンポーネントの型にそれをキャストしなくても、それで直接 `getValue()` を呼び出すことができます。

さらに、以下のように別の便利なメソッド `getDisplayFieldValue()` を呼び出すと、1 つのステップで値を取得できます。

```
Object value=getDisplayFieldValue("textField1");
```

また、`getDisplayFieldStringValue()`、`getDisplayFieldBooleanValue()`、および `getDisplayFieldIntValue()` といったこのメソッドの変形は、特定の型のオブジェクトを返します。

IDE は、パターン `CHILD_<childName>` を使って、ViewBean または ContainerView クラスに子ビューコンポーネント名定数を自動的に生成します。したがって、前述のように `getChild()` またはその変形への呼び出しで文字列表現を使用する必要はありません。さらに IDE は、キャストしないでビューコンポーネントへの参照を取得できるように、`get<childName>Child()` のような子ビューコンポーネント補助メソッドを生成します。

DisplayField 値の設定は、その取得と同じように行います。DisplayField インスタンスを取得したら、その `setValue()` または `setValues()` メソッドを呼び出してその値を変更できます。また、コア ContainerView 実装はいくつかの変形を備えた便利なメソッド `setDisplayFieldValue()` を提供しており、開発者はこれを使って簡単に値を設定できます。

以下に、DisplayField 値を取得し、これらを処理してから、その結果を別の DisplayField に設定する例を示します。

```
int value1=getDisplayFieldIntValue("intField1");
int value2=getDisplayFieldIntValue("intField2");
int result=value1+value2;
setDisplayFieldValue("message",value1+"+"+value2+"="+result);
```

TiledView といった ContainerView の変形には、データを適切に取得および設定するために付加的なメソッドを使用する必要があるものもあります。この詳細については、このマニュアルの他の部分で説明します。

モデル値の操作

アプリケーション値を取得するための 2 番目のテクニックは、モデルを直接使用する方法です。通常 DisplayField は、その値の格納場所を提供するモデルにバインドされています。このバインドは、ModelReference オブジェクトとフィールド名を介して実行されます。一般に DisplayField は、モデルフィールドにバインドされます。

このため、モデルへの参照を持っており、取得するフィールドが分かっている場合は、`Mode.getValue()` または `Model.getValues()` メソッドを使ってモデルに直接データを要求できます。IDE で作成されたビューコンポーネント内でモデルへの参照を最も簡単に取得する方法は、ModelReference オブジェクトを使用することです。ModelReference オブジェクトは、開発者が DisplayField とモデル間のバインドを定義すると自動的に作成され、クラスのメソッドからも使用できます (これにより、全員が同じモデル参照を共有できます)。

ModelReference インスタンスを取得したら、その `getModel()` メソッドを呼び出すだけでモデル参照を取得できます。その後で、以下のようにモデルから値を取得できます。

```
Model model=modelReference1.getModel();
Object value1=model.getValue("field1");
Object value2=model.getValue("field2");
```

IDE によって、IDE 内に作成されたモデルクラスにフィールド名定数が自動的に生成されます。文字列表現の代わりにこれらの定数を使用することで、モデル内のフィールドをより簡単に参照できます。

また、`Model.setValue()` および `Model.setValues()` メソッドを呼び出して、モデルの値を設定できます。この呼び出しでは、単に新しい値とともにフィールド名を指定します。

DatasetModel といった特殊なモデルには、データを適切に取得および設定するために付加的なメソッドを使用する必要があるものもあります。この詳細については、このマニュアルの他の部分で説明します。

J2EE API による値の取得

最後に開発者は、下位レベルの J2EE/ サーブレット API を使ってアプリケーションデータを取得できます。サーブレット API は、要求で直接送付された値を取得できる `getParameter()` および `getParameterValues()` といったメソッドを持つ、`javax.servlet.http.HttpServletRequest` クラスを定義します。適切なパラメータの名前さえ分かっていれば、これらの値を取得できます。

Web アプリケーションフレームワークビューコンポーネントは、命名スキーマを使って DisplayField コンポーネントの限定名を自動的に生成します。この限定名には、接頭辞としてその親の名前がルートから順に付けられた DisplayField の名前が使用されます。デフォルトでは、これらの名前はドット (.) 文字によって限定されます (別の値の使い方については、配備の節を参照してください)。

たとえば、「Foo」という ViewBean に「bar」という ContainerView が含まれ、この ContainerView に「bat」というテキスト DisplayField が含まれる場合は、フィールドの限定名は `Foo.bar.bat` になります。このフィールドが HTML フォーム内にある場合は、開発者は要求内で「Foo.bar.bat」というパラメータを探すことでこのフィールドの値を取得できます。

TiledView のような ContainerView の変形では、要求の複数の値をデコードできるように、付加的な情報が限定名に追加されます。TiledView では、表示される行を区別するために、フィールド名に添字が追加されます。要求で送付されたタイルの数に応じて、添字によって区別される同じようなパラメータ名がいくつか使用されます。たとえば前述の例で「bar」が TiledView の場合、要求内のパラメータは以下のようになります。

```
Foo.bar[0].bat
Foo.bar[1].bat
Foo.bar[2].bat
```

コンポーネントに含まれる子の命名方法については、各コンポーネントのマニュアルを参照してください。

コンポーネントに生成された限定名によって、フレームワークは送付サイクル時に要求内のパラメータを表示フィールドに自動的にマッピングして、さまざまな作成者が作成したコンポーネントを互いに識別できます。フィールド名で均一の名前空間が使用されていたら、アセンブルしてアプリケーションを作成するためにすぐに使用できるコンポーネントは作成できないでしょう。

コア DisplayField 実装 (BasicDisplayField など) には、要求パラメータのこれらのフィールドに対応する値を取得するために使用できる `getRequestValue()` および `getRequestValues()` というメソッドが装備されています。フィールドに送付された値を取得する方法としては、フィールド名を使って要求パラメータを直接使用するより通常はこちらのほうが便利です。これらのメソッドを使用すると、現在のフィールド値が変更されている場合でも、要求で当初送付された値を参照できます。

要求内の値は読み取り専用であるため、設定することはできません。

表示イベントの使用

ページの生成の際に、ビューの生成方法を変更したり、または表示を完全に省略した方がよい場合があります。JSP 主体型の他のフレームワークでは、JSP で複雑なコントロールや他のロジックを使用しない限りこれは困難または不可能で、これは本質的にデバッグと保守が難しいものです。JSP 標準タグライブラリといったソリューションでも JSP に複雑なコードのような構造が生じてしまい、基本的に JSP タグでのプログラミングのために Java のプログラミングを諦めることになります。

これに対し、Web アプリケーションフレームワークは表示イベントというものを提供しています。これによって開発者は、複雑な表示指向のロジックを JSP 外部ではなく、Java コードが意味をなすビュークラス内で実現できます。表示関連イベントには主として、コンポーネントの表示開始時にトリガーされるものと、コンポーネントの表示終了時にトリガーされるものの 2 つがあります。これらのイベントは、コンポーネントが ContainerView 自体であるか、または子ビューであるかによって異なります。

コンテナ表示イベント

JSP の生成時に ContainerView が表示されるときは、対応する JSP タグが表示を開始および終了するときに、その `beginDisplay()` および `endDisplay()` メソッドがそれぞれ自動的に呼び出されます。開発者はサブクラスでこれらのメソッドをオーバーライドできますが、多くのスーパークラスは表示通知時に有用な必須タスクを実行するためにこれらのメソッドを実装しています。このため、これらのメソッドを

オーバーライドした場合、開発者は常にメソッドのスーパーバージョンを呼び出すことが必要になり、その結果アプリケーションを組む際にこれらを使用するのが困難になります。

したがって、アプリケーション開発者が使用できるように予約されているイベントメソッド (少なくとも com.iplanet.jato.view.ContainerViewBase から派生したコンポーネント内のもの) は、beginComponentDisplay() および endComponentDisplay() メソッドになります。これらのメソッドはスーパークラスにより必要に応じてトリガーされ、スーパークラスに何も実装していないこと (より正確に言えば、アプリケーション開発者が完全に無視できない実装がないこと) がある程度保証されています。これらのメソッドは、他の ContainerView に子として追加された ContainerView の「イベント」コンテキストメニューにイベントとして示されます。

コンポーネント開発者は、これらのイベントに応答する beginDisplay() および endDisplay() メソッドを直接オーバーライドして、アプリケーションアセンブラ用のコンポーネントバージョンを残します。

子表示イベント

前の節で説明したコンテナ関連の表示イベントによって、ContainerView コンポーネント自体が表示通知に応答することが可能になりますが、多くの場合コンテナは自身の子の生成に応答することが必要とされます。このため、ContainerView では、その子の 1 つが生成間近であることをコンテナに伝えるために生成時に使用される beginChildDisplay() および endChildDisplay() メソッドを定義しています。beginChildDisplay() メソッドは boolean 型の結果を返し、これが True の場合は子を生成すべきであることを、False の場合は子をスキップすべきであることを示します。

コンポーネント開発者はこれらのメソッドを直接オーバーライドできますが、アプリケーション開発者には、きめの細かい子表示に応答するための簡単な仕組みがあります。Web アプリケーションフレームワークコンポーネントライブラリに含まれる各種拡張可能ページレットコンポーネントは、開始または終了の子表示通知に応答するときに特定の署名のメソッドを探します。これらのイベントが存在する場合は、対応する子が生成されるたびに呼び出されます。これらのメソッドのシグニチャーは、以下のようになります。

```
public boolean begin<child name>Display(ChildDisplayEvent event)
    throws ModelControlException;
public String end<child name>Display(ChildContentDisplayEvent
event)
    throws ModelControlException;
```

<*child name*> は、大文字表記の子ビュー名です。たとえば、ページに「foo」という子がある場合は、アプリケーション開発者は単に前述のシグニチャーを使って `beginFooDisplay()` メソッドを実装することで、その生成に応答できます。

`begin<child>Display()` メソッドの場合の `boolean` 戻り値により、開発者はオプションとして子の生成をスキップできます。`end<child>Display()` メソッドの場合の `String` 戻り値は、その子に対して生成される実際のマークアップです。イベントオブジェクトには JSP 生成時に子に対して計算されたマークアップが含まれますが、開発者は終了表示イベントでそのマークアップを調整または完全にオーバーライドできます。

子表示イベントの使用は動的ページを作成するための強力なテクニックで、可能な限り JSP がコードなしですむようにします。表示イベントの一般的な使い方としては、ユーザーロールまたは他のユーザー情報に基づいた子の表示のスキップ、(生成されるたびの) 生成直前の子の値の計算、それが返した行数を判断するモデルの実行、生成された HTML コントロールへの付加的なマークアップの追加、表の行の色の動的な変更、TiledView 生成時の定期的なヘッダー情報の挿入など、多くがあげられます。開発者にとって表示イベントは、エンタープライズアプリケーションを作成する際に不可欠なツールで、表示関連コードが ViewBean や ContainerView の 1 カ所にまとめられるため、そのコンポーネントを使うすべての JSP で同じロジックを使用できます。

コンテンツタグ

多くの場合、含まれるコンテンツが動的コンテンツ、静的コンテンツ、または両方の混在であるかどうかに関係なく、Web アプリケーションフレームワーク表示イベントを JSP の任意のセクションと関連付けることができれば有用です。<*jato:content*> タグは、まさにこの機能を提供します。このタグの名前属性を使って、取り囲んでいる ContainerView に対する表示イベントコールバックを判断します。

コンテンツタグを使用するときは、あらゆる子ビューに対して行うのとまったく同じ方法で、表示イベントを実装します。違う点は、子ビューが存在せず、生成されるコンテンツが、コンテンツタグが取り囲んでいるあらゆるもの (または、そのタグの endDisplay イベントが返すあらゆるもの) から得られることです。

コンテンツタグは一般的に、ページの一部の生成を条件付けたり、特定ポイントで任意のマークアップをページに挿入するために使用されます。たとえば、コンテンツタグを使用すると、生成される TIledView の各タイルの色を簡単に変更できます。

ViewBean の使用

ViewBean は特殊な種類の ContainerView に過ぎず、その動作のほとんどをスーパークラスから継承します。60 ページの「ContainerView の使用」では、ViewBean を使用する際に必要なほとんどの情報が提供されています。以下の節では、ViewBean に関する補足的な情報を詳しく説明していきます。

forwardTo() メソッド

ページである ViewBean は、開発者がアプリケーション内に作成する基本の作品です。このため ViewBean は、要求を制御するための主要メソッドを備えています。開発者は `RequestDispatcher` といったサーブレット機能を直接使ってクライアントへの応答を生成できますが、実際には ViewBean で `forwardTo()` メソッドを使ってページの生成を開始したほうがより簡単です。その主な理由は、各 ViewBean はそれが関連付けられている JSP を知っているため、サーブレット要求を転送するための詳細を処理できるからです。さらにこの方法を使うと、アプリケーション開発者は下位の J2EE プリミティブではなく、常に「ページ」を念頭に置いて作業することができます。

`forwardTo()` メソッドは、表示フェーズが始まる前に ViewBean がロジックを実行する機会を提供します。最も注目すべきテクニックは、実際に要求を転送する前に生成したい JSP を ViewBean が動的に選択する方法です。たとえば ViewBean は、現在の要求が WAP デバイスからのものであると判断できるため、HTML JSP の代わりに応答として WML JSP を生成します。複数の関連付けられた JSP を使って同じ ViewBean を生成するこの機能は、並列コンテンツと呼ばれます。この詳細は、37 ページの「URL と並列コンテンツの表示」を参照してください。

ページセッション

要求間の一部の情報を保持することが必要な場合がよくありますが、この情報をサーバー側 HTTP セッションに置くと、ユーザーがブラウザの「戻る」ボタンを使ったときにアプリケーションの同期がずれる可能性があります。この問題を解決するために、ViewBean はページセッションというものを備えています。ページセッションは、HTTP セッションのように動作します。ただし、ページセッション属性はクライアントに生成された応答に格納されて、次回の要求で再送付されます。ユーザーが「戻る」ボタンを使用しても、ページセッションの同期は自動的に維持されます。なぜなら、特別に生成されたページのページセッションの各バージョンが、クライアント上で各ページにキャッシュされるからです。

ページセッションは、従来の Web アプリケーションで HTML 非表示フィールドが使用されるように、アプリケーションでユーザーのコンテキストを追跡するために最もよく使用されます。ただし、非表示フィールドよりページセッションが優れている点は、ページセッションはすべてのリンクやフォームに自動的に追加され、文字列だけではなく複雑なオブジェクトを含む場合があることです。さらに、HTTP GET 要求時にブラウザに送信される URL の長さには制限があるため、ページセッションメカニズムはページセッションが、あるサイズに達したらこれを圧縮することが可能です (圧縮率は通常、実効率約 40 ～ 50% です)。

ある一連のページセッション属性は、単一の ViewBean に固有のものです。ページセッション属性は、複数の ViewBean で共有されません。次回要求用に属性を保持する必要がある場合は、そのアプリケーションは処理ページから次の表示ページにページセッション属性をコピーする必要があります。

ページセッション API の詳細は、`com.iplanet.jato.view.ViewBean` JavaDoc を参照してください。

重要：ページセッションは、HTML 非表示フィールドに格納された値のようにクライアントに送信される他の暗号化されていない値と同様、セキュリティ保護されていません。これはクライアントに送信されるときに、暗号化ではなく単に符号化されるだけであるため、悪意のあるユーザーが、改ざんされたまたは偽のページセッションを含む要求を巧妙に作ることが可能です。アプリケーションで何をページセッションに格納するかを決める際は、必ずこのことを念頭に置いてください。

クライアントセッション

クライアントセッションの概念はページセッションに似ていますが、これはアプリケーションの全ページ (および他のオブジェクト) によって共有されます。ページセッションと同じように、クライアントセッションはすべての Web アプリケーションフレームワーク URL に自動的に追加され、大きくなると自動的に圧縮される場合があります。

クライアントセッションは RequestContext から使用できます。API の詳細は、`com.iplanet.jato.ClientSession` JavaDoc を参照してください。

重要：クライアントセッションは、HTML 非表示フィールドに格納された値のようにクライアントに送信される他の暗号化されていない値と同様、セキュリティ保護されていません。これはクライアントに送信されるときに、暗号化ではなく単に符号化されるだけであるため、悪意のあるユーザーが、改ざんされたまたは偽のクライアントセッションを含む要求を巧妙に作ることが可能です。アプリケーションで何をクライアントセッションに格納するかを決める際は、必ずこのことを念頭に置いてください。

ContainerView の使用

ContainerView は、他のビジュアル開発環境におけるパネルコンポーネントに類似しています。これらは、含まれている一連のコンポーネントをグループとして操作できるようにグループ化する方法を提供します。また ContainerView は、非常に複雑なコンポーネント (配布可能コンポーネントと非配布可能コンポーネントの両方) の基礎を形成します。

含まれているコンポーネントの繰り返し生成能力といった補助的な動作が追加された、標準的な個別化の施された ContainerView があります。詳細は、64 ページの「TiledView の使用」を参照してください。

他のコンポーネントを含める基本機能を超えた、ContainerView に補助的な動作を追加する機能により、これらは非常にパワフルで用途が広くなっています。

ContainerView の IDE サポート

Web アプリケーションフレームワーク IDE を使用すると、一覧から ContainerView を選択し、ContainerView サブクラスをアプリケーションに追加するだけで、簡単に ContainerView を構築できます。ただし、`com.iplanet.jato.view.ContainerView` インタフェースを学習した場合は、子として作成すべきコンポーネントの自動ツールによる作成を可能にするメソッドがこれには含まれていないことがわかります。このインタフェースは実行時のフレームワーク要件の見地から作成されており、実行時に子コンポーネントが ContainerView によって実際にどのように作成および管理されるかをフレームワークは意識しないためです。

このため、ContainerView ベースのコンポーネント作成のための IDE サポートは、規則に依拠することになります。IDE で操作可能な新しい拡張可能 ContainerView コンポーネントを作成するには、コンポーネントが以下のメソッドを提供する必要があります。ただし、コンポーネントが最終的に `com.iplanet.jato.view.ContainerViewBase` から派生する場合は、これらの規則はすでに満たされているため、これらを意識する必要はありません。この規則は、ViewBean や TiledView を含むすべての特殊な ContainerView にも適用されます。

以下の表に、IDE で操作可能な新しい拡張可能 ContainerView コンポーネントを作成するためにコンポーネントが提供するメソッドを示します。

メソッド	説明
`void registerChild(String name, Class type)`	このメソッドは、コンポーネントの初期化時に、そのすべての子コンポーネントの名前と型を登録するために呼び出されます。この情報を使って、子コンポーネントのインスタンス化を行うことなく、特定 ContainerView インタフェースと実装メソッドを満たすことができます。
`View createChildReserved(String name)`	このファクトリメソッドは、必要に応じて名前付き子コンポーネントを作成するために呼び出されます。開発者がコンテナに子コンポーネントを追加したりコンテナの子コンポーネントを操作したりすると、このメソッドの実装が IDE によって自動的に生成されます。
`View createChild(String name)*`	この代替ファクトリメソッドは、必要に応じて名前付き子コンポーネントを作成するために呼び出されます。このメソッドの実装は IDE によって自動的に生成されませんが、従来どおり開発者がコードを使ってコンテナにコンポーネントを手動で追加できるようになっています。 * このリストの他の 2 つのメソッドとは異なり、このメソッドは IDE では必要とされませんが、Web アプリケーションフレームワーク開発者は通常この存在に慣れているため、新しいコンポーネントの種類として含めることを強く推奨します。

将来的にはこれらのメソッドは、IDE がサポートする ContainerView コンポーネントによって実装されるように、追加インタフェースに含まれる可能性があります。ただし現時点では、このインタフェースは必要条件ではありません。

ContainerView API

以下の説明では、前の項「ContainerView の IDE サポート」で説明されている規則を実装している、Web アプリケーションフレームワークコンポーネントライブラリの一部として提供されているデフォルト ContainerView 実装 (`ContainerViewBase`) に焦点をあてます。ただし、`ContainerView` インタフェースは、ContainerView コンポーネントの最低要件で、実装は異なる場合があります。ContainerView コンポーネントを実装するための最低要件については、`com.iplanet.jato.view.ContainerView` クラスの JavaDoc を参照してください。

基礎の ContainerView API では、他の ContainerView を含む他のビューコンポーネントを含む機能に焦点をあてています。この機能に付随しているのが、親および子コンポーネントという概念です。親コンポーネントには、1 つ以上の子コンポーネントが含まれます。子コンポーネントもそれ自身の子の親となることが可能なため、他の

コンポーネントの子としてコンポーネントを追加するだけで、任意に入り組んだコンテナ階層を構築できます。ContainerView.getChildNames()、ContainerView.getChildType()、および ContainerView.getChild() といったメソッドは、ContainerView の使用先 (開発者とフレームワークの両方) がその子コンポーネントを使って作業できるようにするコアメソッドです。

ContainerView の子としてコンポーネントを定義するには、通常 2 つの作業を行います。IDE を使用している場合は、ViewBean または他の可視コンポーネントに子コンポーネントを視覚的に追加すると、これらの作業は (コード生成によって) 自動的に処理されます。

最初の作業では、名前と型のマッピングを登録するために、親コンポーネントで registerChild() メソッドを呼び出します。この作業は、ContainerView がその子コンポーネントを実際にインスタンス化しないで、これらについての判断を下せるようにするために必要です。子コンポーネントオブジェクトが要求時に使用されるかどうかに関係なく、フレームワークではすべての要求で子コンポーネントオブジェクトを作成することを避けたいため、この機能は効率を上げるために重要です。登録によって収集された情報は、すべての子コンポーネントをインスタンス化しないで ContainerView.getChildType() および ContainerView.getNumChildren() メソッドへの呼び出しを満たすために使用されます。

2 番目の作業では、所定の子を名前によってインスタンス化する方法を ContainerView に指示します。各 ContainerView はこのために、createChild() および createChildReserved() という 2 つのメソッドを備えています。これらのメソッドは、動作はまったく同じですが、後者は自動ツールで使用されるために予約されています。これらのメソッドは、名前パラメータによって指定された子をインスタンス化し、構成して、返すファクトリメソッドです。コンポーネントの実装はこのメソッドで返されたコンポーネントをキャッシュするため、このメソッドは要求時に子について多くても 1 度だけ呼び出されます。各要求で、または要求時に同時に、すべての子コンポーネントがインスタンス化されるわけではありません。このメソッドは、子コンポーネントをインスタンス化するために必要に応じて呼び出されます。たとえば、getChild() メソッドを呼び出して子への参照を取得したときに、これがその要求内でまだインスタンス化されていない場合は、子コンポーネントのインスタンスを取得するために createChild() (または createChildReserved()) が呼び出されます。コンポーネントはすでにインスタンス化されていて、コンテナによってキャッシュされているため、その要求内で getChild() を再度呼び出しても、createChild() は呼び出されません。

createChild() または createChildReserved() は、子コンポーネントインスタンスを取得するために決して直接呼び出してはいけません。その代わりに、getChild()、getDisplayField()、または IDE で生成された補助メソッド (例 : getFooChild()) を呼び出して、コンポーネントへの参照を取得してください。また一般には、ContainerView のコンストラクタ内からは、これらのメソッドのどれも確実に呼び出したり、子コンポーネントの参照を取得したりすることはできません。この制限事項に対する唯一の注意点は、子コンポーネントを取得するための呼び出しの前に ContainerView で RequestContext が設定されている場合 (またはすべての子コンポーネントが RequestContext に静的アクセスできる場合) です。その場

合、ContainerView とそのすべての子は、適切にインスタンス化および構成されることが必要なすべての要求リソース (モデルなど) に確実にアクセスできるようにするのに役立ちます。

アプリケーションでの ContainerView の使用

ContainerView は多くの点でページ (または他の ContainerView) に追加する他の子コンポーネントと似ており、設定可能なプロパティを備え、表示イベントが関連付けられています。

デフォルトモデル

すべての ContainerView は、デフォルトモデルを備えています。これは、モデルの格納場所が他で指定されていない子コンポーネントに対して、「デフォルト」の格納場所として機能するモデルです。特に指定されていない限り、デフォルトモデルは、基本的なメモリー記憶機構を実装している `com.iplanet.jato.model.DefaultModel` のインスタンスの 1 つです。

デフォルトモデルを使用する利点は、複雑なモデルバインドを必要とせず、必要とされる任意の値をフィールドに格納できることです。さらに、DisplayField 値は送付されて、そのままデフォルトモデルに格納されるため、バックエンドシステムや他のモデルに値を送信する前に、開発者は非常に柔軟に値を検査して操作できます。

現在の Web アプリケーションフレームワークにおける各種 BasicDisplayField 型の実装では、ModelReference がコンポーネントで明示的に設定されていない場合は、その親のデフォルトモデルを使用します。

子ビューパス

`ContainerView.getChild(String)` への呼び出し内の子の名前には、区切り記号としてスラッシュ (/) を使った限定ビューパスを使用できます。最後のものを除くパス内の、全構成要素は、ContainerView または ContainerView の派生物 (TiledView など) を参照する必要があります。相対パスと絶対パスの両方とも使用できます。名前パスがスラッシュで始まる場合は、名前はルート (ViewBean) への相対パスと想定されます。パスの先頭文字がスラッシュ以外の場合、名前は現在のコンテナに関係のある子を参照すると想定されます。現在のコンテナの親であるコンテナは、2 つのドット (..) を使って参照できます。

TiledView の使用

TiledView は、自分の子を繰り返し生成する特殊な ContainerView です。それぞれの繰り返しは、タイルと呼ばれます。タイルは、表の行を生成するために最もよく使用されますが、一連のタブ、ブレッドクラブコントロール、または反復生成を必要とする他の任意の構造を生成するためにも使用できます。TiledView は、子を管理する標準機能に加え、一連のタイルの繰り返しを制御するための `next()`、`first()`、および `last()` といったメソッドを提供します。

それぞれの TiledView は、種類が `DatasetModel` の一次モデルと関連付ける必要があります。DatasetModel は表形式でデータを提供し、TiledView の生成時に、基本的に関連付けられた一次モデルのデータの反復子のように機能します。モデルの各行で、子ビューが生成されます。これらの子 (特に子の DisplayField) が一次モデルにバインドされている場合は、それらの値はタイルの繰り返しごとに変更されます。

TiledView の `nextTile()` メソッドは、生成時に必要に応じて各タイルに移動するために使用されます。このメソッドは TiledView が新しいタイルに移動したかどうかを示す論理値を返し、デフォルトの実装では一次モデルの `next()` メソッドを使って生成するデータがまだあるかどうかを判断します。一般には、`nextTile()` メソッドをオーバーライドして追加チェックを実行するか、タイルの生成に必要なデータを初期化する方法が使用されます。

送付フェーズでは、まず TiledView を所定のタイルに位置付けてから、そのタイルの DisplayField 値を読み取る必要があります。CommandField コンポーネントを TiledView 内部に配置し、その起動によって開始された要求を処理するときにその値を読み取りたい場合は、以下のコード断片を使って CommandField の値を取得する前に適切な行に TiledView を位置付ける必要があります。

```
getPrimaryModel().setLocation(
    ((TiledViewRequestInvocationEvent)event).getTileNumber());
```

`event` は、要求イベントハンドラに指定された RequestInvocationEvent パラメータになります。

TiledView の入れ子 (TiledView 内への TiledView の配置) は、読み取り専用データの生成には機能しますが、内部 TiledView のデータがアプリケーションに送付される場合は、一般に期待どおりには機能しません。送付されたデータは要求パラメータ内に存在しますが、ターゲットモデルに適切にマッピングできません。この理由は、このマニュアルでは説明しません。この代わりにアプリケーションで機能するテクニックは、SimpleCustomModel (またはそれ自身のデフォルトモデル) をその一次モデルとして使用するように 内部 TiledView を設定する方法です。そしてモデルにプッシュされたデータを無視して、代わりに各子の `BasicDisplayField` で `getRequestValue()` メソッドを呼び出しながら入れ子の TiledView を繰り返しま

す。これにより、文字列の構文解析を大量に行って手動でパラメータ名を構築しなくても、TiledView 位置の組み合わせに送付されたサーブレットパラメータ値が返されます。

TreeView の使用

TreeView は、階層型に子を生成する特殊な ContainerView です。これらは子を管理する標準機能に加えて、データ階層の生成時に現在のノード位置を判断するためのサポートを提供し、また展開または縮小するノードを判断する状態データが関連付けられています。

それぞれの TreeView は、種類が `TreeModel` の一次モデルと関連付ける必要があります。TreeModel は階層型でデータを提供し、TreeView の生成時に、基本的に関連付けられた一次モデルのデータの反復子のように機能します。ツリーの各ノードで、関連付けられた JSP タグで開発者が宣言する規則に基づいて、子ビューを表示、非表示、またはカスタマイズできます。このため、他の種類の ContainerView と比べて TreeView は比較的複雑になります (ただし、同じ作業を手動で実行するよりかなり簡単に使用できます)。

TreeView は Web アプリケーションフレームワークタグライブラリと密接に連携して動作し、その特殊な要件のためだけに定義されたタグをいくつか備えています。これらのタグの詳細は、『Sun Java Studio Web アプリケーションフレームワークタグライブラリリファレンス』を参照してください。

実行モデルの使用

ほとんどのモデルは、データを含めることに加えて、ビジネス委託として機能し、バックエンドシステムのデータを取得したり変更するためのオペレーションをサポートする必要があります。このため Web アプリケーションフレームワークでは、自身でオペレーションを起動することも可能なモデルとして、実行モデルという概念を定義しています。

Web アプリケーションフレームワークの実行モデルには、いくつかの種類があります。最も基本的な種類は、`com.iplanet.jato.model.ExecutingModel` インタフェースによって表されます。このインタフェースは、モデル上で任意の名前のオペレーションを起動するために使用できる `execute()` という 1 つのメソッドを提示します。開発者が適切なオペレーション名を使って `execute()` メソッドを呼び出すことができるように、このインタフェースを実装する各モデルはサポートするオペレーションを記録する必要があります。

ただし、ほとんどのモデルに共通のオペレーションがいくつかあります。これらのオペレーションは、検出、挿入、更新、削除という 4 種類に分類されます (これらの種類は SQL オペレーションと似ていますが、正式な関係はありません。ただし、JDBC ベースのモデルの場合は、これらは SQL オペレーションにマッピングされる場合があります)。これらのオペレーションを処理する機能は、`RetrievingModel`、`InsertingModel`、`UpdatingModel`、および `DeletingModel` インタフェースによって表されます。これらのインタフェースを実装するモデルは、この種の実行対話をサポートすることを宣言するため、呼び出し元でオペレーション名を指定する必要はありません。

最も一般的な種類の実行モデルインタフェースである `ExecutingModel` を使用すると、モデルでほとんどのオペレーションを起動できますが、フレームワークをはじめとする特定のインタフェースの型を使用すると、よく理解されている方法でモデルと連携することが可能になります。実際に、こうした特定の種類のインタフェースは WebAction 機能の基盤となります。一方これらは、構築したり対話するモデルのセマンティクスを明確に定義するという点で、Web アプリケーションフレームワーク開発者にとって非常に有用なものでもあります。

一般に、実行モデル (全種類) は、アプリケーションの特定ポイントで開発者によって実行されます。たとえば、要求イベントハンドラが現在の要求からデータを取得した場合は、`UpdatingModel` を実装するモデルにそのデータを挿入し、そのモデルで `update()` を呼び出して、そのモデルによって表されるバックエンドシステムにそのデータをプッシュします。ただし、WebAction (以下で説明) もアプリケーションの特定ポイントで特定の種類のモデルを自動的に実行することができ、これによって開発者はデータをモデルから出し入れするためのコードを書く必要がなくなります。

BeanAdapterModel の使用

BeanAdapterModel を使用すると、開発者は 1 つ以上の JavaBean をモデルのバッキングデータストアとして使用できます。これによって DisplayField を JavaBean プロパティにバインドすることが可能になります。これは、アプリケーションオブジェクトモデルがあり、ビューに対するこれらのオブジェクトの自動バインドを利用するときに便利です。

モデルで使用する JavaBeans は、`setBean()` または `setBeans()` メソッドによって開発者が明示的に設定することも、標準 J2EE 要求、セッション、またはアプリケーションスコープの検索メカニズムを使って取得することもできます。この機能によって、モデルを他の J2EE コンポーネントと簡単に相互運用できます。その Bean をモデルに自動的に検索させるには、「Bean スコープ」プロパティまたは `setBeanScope()` メソッドを使ってスコープを指定します。

BeanAdapterModel は `DatasetModel` インタフェースをサポートしており、1 行あたり 1 要素の JavaBean の並びをモデルで使用できます。これはまた WebAction を介してページネーションもサポートしています。BeanAdapterModel のサブクラスを

作成する開発者は代替実行モデルメソッドを実装し、これらを使って EJB、オブジェクトリレーショナルマッピングレイヤー、または JavaBean としてデータを提供する他のシステムから Bean を取得することもできます。これによって BeanAdapterModel はさまざまな状況で使用することが可能となっており、また BeanAdapterModel を使って非 Web アプリケーションフレームワークオブジェクトを柔軟かつ簡単にアプリケーションに統合することが可能となっています。

ObjectAdapterModel の使用

使い方については、`com.iplanet.jato.model.object.ObjectAdapterModel` の JavaDoc を参照してください。

WebAction の使用

WebAction は Web アプリケーションフレームワークビューコンポーネントに組み込まれた特殊な動作で、これによってビューコンポーネントは自動モデル実行とページネーション機能を提供することが可能になります。特に、BasicViewBean、BasicContainerView、および BasicTiledView コンポーネントは、`com.iplanet.jato.view.WebActionHandler` インタフェースの実装として WebAction をサポートしています。

WebAction ビューコンポーネントと連携するのが、WebAction モデルです。これらは新種のモデルではなく、むしろモデルと WebAction ビューコンポーネントの関連付けと言えます。モデルは、「自動削除モデル」、「自動実行モデル」、「自動挿入モデル」、「自動検出モデル」、および「自動更新モデル」プロパティによって、WebAction ビューコンポーネントと関連付けられます。これらのプロパティは実行モデルの標準カテゴリを反映しています。これは、1 つ以上の標準 `ExecutingModel` インタフェースを実装するどのモデルも、WebAction モデルとして使用できることを意味します。

WebAction コンポーネントと関連付けられたモデルは、その処理メソッド `handleWebAction(int)` を呼び出すことで、WebAction のコンテキストで実行できます。開発者はこのメソッドをイベントハンドラから直接呼び出すことも、WebAction コマンドコンポーネントなどによって自動的に呼び出すこともできます。要求イベントハンドラメソッド内での一般的な WebAction の使用例を以下に示します。

```
public void handleButton1Request(RequestInvocationEvent event)
     throws Exception
{
     handleWebAction(WebActions.ACTION_NEXT);
     getParentViewBean().forwardTo(getRequestContext());
}
```

一般的に WebAction の実行後に適切な動作は、現在のページを再度読み込むことです。ただし、ACTION_UPDATE、ACTION_INSERT、および ACTION_DELETE といった特定の WebAction では、別のページに転送するのが適切な場合もあります。

WebAction の種類

handleWebAction(int) メソッドを呼び出すときは、実行すべき WebAction の種類を指定する必要があります。

以下の表に、利用可能な WebAction の種類を概説します。

WebAction の種類	説明
ACTION_FIRST	すべての自動検出データセットモデルを実行して、データセットの先頭行に移動します。
ACTION_NEXT	すべての自動検出データセットモデルを実行して、(前のアクションで定義された)次の行グループに移動します。
ACTION_PREVIOUS	すべての自動検出データセットモデルを実行して、(前のアクションで定義された)前の行グループに移動します。
ACTION_LAST	すべての自動検出データセットモデルを実行して、最終行グループに移動します。
ACTION_UPDATE	すべての更新 Web アクションモデルを実行します。
ACTION_INSERT	すべての挿入 Web アクションモデルを実行します。
ACTION_DELETE	すべての削除 Web アクションモデルを実行します。
ACTION_CLEAR	どのモデルも実行せず、結果として空白ページを生成します。
ACTION_EXECUTE	すべての実行 Web アクションモデルを実行します。
ACTION_REFRESH	現在の自動検出データセットモデルを再実行します。

WebAction イベント

WebAction は、実行したりエラー状態に応答するためにモデルを準備できないコンテキストで起こる可能性もあるため (try/catch ブロック内など)、開発者が実装できる多数の WebAction イベントが用意されています。

以下の表に、これらの WebAction イベントを示します。

WebAction イベント	説明
`beforeWebActionModelExecutes()`	WebAction コンテキスト (例 : 検出、挿入、更新、削除) に関係なく、各 WebAction モデルが実行される前に呼び出されます。開発者はこのメソッドを使って、要求固有の値をモデルに設定するなど、実行用にモデルを準備できます。
`afterWebActionModelExecutes()`	WebAction コンテキスト (例 : 検出、挿入、更新、削除) に関係なく、各 WebAction モデルが実行された後で呼び出されます。ビューによって生成される前にモデルデータをチェック、集計、ソートするために使用できます。
`afterAllWebActionModelsExecute()`	WebAction コンテキスト (例 : 検出、挿入、更新、削除) に関係なく、`handleWebAction()` の現在の呼び出しに対してすべての WebAction モデルが実行された後で呼び出されます。WebAction に関連して、ビューが生成される前に最後のアクションを実行するために使用できます。
`onWebActionExecutionError()`	WebAction モデルの実行にエラーが発生した場合に呼び出されます。対策を講じたり要求を中止するために使用できます。

自動検出モデル

WebAction は、WebAction ビューコンポーネントの生成時に自動的にモデルを実行するために最もよく使用されます。ページやページレットを表示するためのモデルを実行するコード (`beginComponentDisplay()` イベントなど) を開発者がアプリケーションの特定ポイントに挿入する代わりに、1 つ以上のモデルを WebAction 検出モデルとして関連付けることで、表示が開始するとコンポーネントによってデータを自動的に検出できます。

さまざまな WebAction ビューコンポーネントが、生成を開始するときにすべての WebAction 検出モデルを実行しようと自動的に試みます。WebAction 検出モデルがない場合は、自動的に実行されるモデルはありません。同様に、モデルの自動検出は `setAutoRetrieveEnabled(boolean)` メソッドによって手動で制御できます。この値を False に設定すると、自動実行をスキップします。

自動検出モデルは、それに関連付けられた WebAction ビューコンポーネントの表示時に実行されるのを待つのではなく、手動で実行したい場合もあります。(たとえば、よくあるケースとして、関連付けられた自動検出モデルを使って子コンポーネン

トの生成を開始する前にモデル内のデータの行数を知っておきたい場合)。このような場合は、モデルを手動で実行してから、関連付けられたビューコンポーネントで `setAutoRetrieveEnabled(false)` を呼び出すことができます。ビューコンポーネントは通常どおり生成されます。

自動検出モデルは、関連付けられたビューコンポーネントがページ上で生成されるたびに実行されます。自動実行の恩恵を受けるには、一般にコンポーネントの子 DisplayField コンポーネントを自動実行されるモデルにバインドすることが必要です。

WebAction によるページネーション

最も有用な WebAction 機能の 1 つが、任意のモデルを使ってデータセットのページネーションを行う機能です。WebAction ページネーション (`ACTION_FIRST`、`ACTION_NEXT`、`ACTION_PREVIOUS`、および `ACTION_LAST`) では、WebAction 検出モデルを使って、1 ページで表示するには大き過ぎるデータセットをユーザーがページ付けできるようにします。

モデルが `RetrievingModel` および `DatasetModel` インタフェース (または複合 `RetrievingDatasetModel` インタフェース) を実装している限り、WebAction ページネーションとともにこれを使用できます。つまり、そのデータソースに関係なく、基本的にどの種類のモデルもこれらのアクションとともに使用できる、ということです。たとえば、JDBC、XML、Web サービス、CICS メインフレームデータ、または他のほとんどのデータソースを使うモデルも、これらが `RetrievingModel` および `DatasetModel` インタフェースを実装していれば、WebAction を使ってページ付けできます。

ページネーションのサポートを有効にするために特別な作業を行う必要はありません。モデルの種類による制限を念頭に置いて、他の WebAction と同じようにこれらの WebAction を使用してください。一般に、Web アプリケーションフレームワークコンポーネントライブラリ付属のコンポーネントの中では、さまざまなカスタムモデルコンポーネント、JDBC QueryModel、BeanAdapterModel、ObjectAdapterModel、および WebServiceModel が、`RetrievingModel` および `DatasetModel` 実装としてページネーションをサポートします。

WebAction を使うタイミング

WebAction は、標準 Web アプリケーションフレームワークコンポーネントの一部の単なる付加価値機能です。したがって、開発者の必要に応じて使用できます。一般に、自動検出および WebAction ページネーションは最も有用な機能です。自動検出によって開発者のコードが減少し、またページネーションコードの記述は一般的に難しいためです。

一般論としては、WebAction によって作業が簡単になる場合は、これらを使用してください。ただし、何らかの理由でこれらに制約される場合は、手動で作業を行ってください。WebAction が提供するすべての動作は、便利ではありませんが、既存の Web アプリケーションフレームワーク機能を使って複製できます。

Web アプリケーションフレームワークアプリケーションとの相互運用

Web アプリケーションフレームワークアプリケーションは結局は J2EE Web アプリケーションであるため、他の Web アプリケーションはこれらと綿密に定義した方法で相互運用できます。Web アプリケーションフレームワークアプリケーションとの相互運用のためのテクニックは、アプリケーションコンポーネントへのアクセス方法によって異なります。

外部アプリケーションからの相互運用

「外部アプリケーション」とは、異なるサーブレットコンテキスト、アプリケーションサーバー、Web サーバー、J2EE または非 J2EE コンテナなどのまったく別個のアプリケーションを示します。

この種の相互運用は、他の Web アプリケーションの相互運用と同じで、Web アプリケーションフレームワークアプリケーションは、標準 HTTP URL を通して起動できます。Web アプリケーションフレームワークアプリケーションにパラメータを渡す場合は、HTTP 呼び出しに照会パラメータを追加して、呼び出された Web アプリケーションフレームワークコンポーネント内からこれらにアクセスします。

ただし、この種の呼び出しを行うと、要求イベントハンドラを呼び出さない最初の要求が発生します。ほとんどの場合、これは問題ありません。しかし、場合によっては、ボタンを押したり HREF (ハイパーリンク) をクリックする動作を試したい場合もあります。この場合は、照会パラメータまたは送信される値として、名前と値のペアを最低 1 つ含める必要があります。

Web アプリケーションフレームワークボタンまたは HREF から結果として発生した各要求には、要求イベントハンドラを呼び出すことをアプリケーションの要求処理インフラストラクチャに伝えるパラメータが含まれています。このパラメータの名前は、ボタンまたは HREF の完全限定名になります。たとえば、`PgFoo` 上のボタン名が `SubmitButton` の場合は、以下のようなパラメータが生成されます。

```
PgFoo.SubmitButton=Submit
```

したがって、このボタンを押す動作をシミュレートするには、以下のような HTTP 要求を Web アプリケーションフレームワークアプリケーションに送信します。

```
http://<host>/fooapp/main/PageFoo?PageFoo.SubmitButton=
Submit&...
```

通常はこれによって `handleSubmitButtonRequest()` メソッドが `PageFoo` で呼び出されます。ほとんどの要求イベントハンドラメソッドで予想されるように、同じページに他のフィールドを生成する場合は、照会文字列または送信される内容に付加的なパラメータを追加する必要があります。

ページの子の TiledView 内で要求イベントハンドラを呼び出す場合は、以下のように行番号の添字もパラメータ名に含める必要があります。

```
PageFoo.FooTiledView[3].SubmitButton=Submit
```

これによって、4 番目のタイルでボタンが押されたかのように (タイル番号は 0 から開始します)、`FooTiledView` という名前の TiledView で要求イベントハンドラが呼び出されます。

同じアプリケーション内からの相互運用

1 つの Web アプリケーションに Web アプリケーションフレームワークコンポーネントと他の J2EE コンポーネント (非 Web アプリケーションフレームワークサーブレットや JSP など) の両方が含まれている場合もあります。同じアプリケーション内から Web アプリケーションフレームワークと相互運用する利点は、要求やサーブレットコンテキスト属性を介してアプリケーションコンポーネントがセッションや他のオブジェクトを共有できることです。これによってコンポーネント間をシームレスに移動することが簡単になります。

ほとんどの場合は、外部アプリケーションから相互運用する場合と同じ基本ガイドラインに従うだけです。ただし、標準 `javax.servlet.RequestDispatcher` メカニズムを使って要求に Web アプリケーションフレームワークコンポーネントを含めるか、転送する必要があります。このように転送したり含めたりするときは、Web アプリケーションフレームワーク JSP を直接要求するのではなく、必ず標準 Web アプリケーションフレームワーク URL を要求する必要があります (適切な Web アプリケーションフレームワークコンテキストを提供するために、要求はモジュールサーブレットを通して実行されることが必要です)。

```
String target="/fooapp/main/PageFoo?PageFoo.SubmitButton=
Submit&...";
RequestDispatcher dispatcher=
    request.getServletContext().getRequestDispatcher(target);
dispatcher.forward(request,response);
```

第4章

アプリケーションの配備

この章では、ほとんどの J2EE コンテナでの配備用に Web アプリケーションフレームワークアプリケーションを準備する方法と、Web アプリケーションフレームワークアプリケーションの配備時構成について説明します。

ここでは、Web アプリケーションフレームワーク、WAR (Web Application Archive: Web アプリケーションアーカイブ) ファイル、配備記述子、および使用する J2EE コンテナ固有の配備方法の全般に習熟していることを前提として説明します。

注 – Web アプリケーションフレームワークアプリケーションは、Servlet 2.2/JSP 1.1 準拠 (J2EE 1.2) の任意の Web コンテナで実行できます。

アプリケーションの構成

Web アプリケーションフレームワークアプリケーションは、パッケージングに加え、配備前に構成する必要があります。

Web アプリケーションフレームワーク IDE を使用する場合は、以下のすべての構成は自動的に処理されます。

この節では、Web アプリケーションフレームワークアプリケーションまたはオブジェクトを手動で作成したい開発者、またはフレームワークの動作を詳しく理解したい開発者に情報を提供することを目的としています。

モジュールサーブレットの構成

一般的な構成

ApplicationServletBase によって構築された Web アプリケーションフレームワークサーブレットインフラストラクチャには、よく似たものを使って各モジュールサーブレットの任意のプロパティを設定する機能が含まれています。アプリケーション配備記述子 (web.xml) 内のパラメータは、以下の特殊な名前書式を使って、コンテキストまたはサーブレット初期パラメータとして指定します。

```
jato:<class name expression>:<param name>
```

以下に例を示します。

```
<context-param>
    <param-name>jato:fooapp.main.MainModuleServlet:foo</param-name>
    <param-value>bar</param-value>
</context-param>
```

以下のように、複数オブジェクトにパラメータを設定するために、指定のクラス名にはアスタリスク (*) のワイルドカード文字を使用できます。

```
<context-param>
    <param-name>jato:fooapp.*:foo</param-name>
    <param-value>bar</param-value>
</context-param>
```

1 つのパラメータは、アスタリスクが 1 つだけ含まれることがあり、これは、アスタリスクの前後両方または一方の文字列とクラス名が一致する任意のクラスと一致します。

すべてのモジュールサーブレットにパラメータが適用される場合は、クラス名表現も省略できます。

```
<context-param>
    <param-name>jato:enabledLogLevels</param-name>
    <param-value>ALL</param-value>
</context-param>
```

パラメータを設定するには、モジュールサーブレットクラスは JavaBeans メソッド命名規則に準拠した設定メソッドを持つ必要があります。たとえば、「foo」という名前のパラメータを指定すると、サーブレットはパラメータ値を設定するために `setFoo()` というメソッドを呼び出します。値は、設定メソッドに適切な型に変換されます。型を変換できない場合は、例外を示すエラーメッセージがサーブレットコンテキストに書き込まれます。

現在使用できるモジュールサーブレットパラメータを、以下の表に示します。

パラメータ名	型	説明	必須
allowShortViewBeanNames	`boolean`	2.1 より前の Web アプリケーションフレームワークバージョンでは、ViewBean クラス名に *ViewBean* 接尾辞を付けることが必要とされました (例 : *FooViewBean*)。 バージョン 2.1 からは、ViewBean にこの接尾辞を付ける必要はありません。 要求のパス情報から ViewBean を検索するメカニズムは、*ViewBean* 接尾辞を強制すべきかどうかを知る必要があります。下位互換性を維持するためにデフォルトではこのパラメータは False になっていますが、新しいアプリケーションではこの機能を有効にしてこの要件を避けることができます。 **注 :** アプリケーションの古い命名規則を使用している部分と新しい命名規則を使用している部分が共存できるように、このパラメータはモジュールごとに設定できます。	×

パラメータ名	型	説明	必須
enabledLogLevels	String	有効にするログレベルのコンマ区切りのリスト。可能な値は以下のとおりです。 NONE MANDATORY STD STANDARD VERBOSE_DEBUG TERSE_DEBUG ANY_DEBUG JATO_TRACE JATO_QOS_TRACE APP_TRACE ANY_TRACE WARNING ERROR CRITICAL ANY_ERROR USER_LEVEL_1 USER_LEVEL_2 USER_LEVEL_3 ANY_USER_LEVEL ALL DEFAULT	×
echoLogToSystemOut	boolean	サーブレットコンテキストに加えて、System.out へのログの状態を設定します。	×
enforceStrictSessionTimeout	boolean	アプリケーションの開発時にアプリケーションを再配備すると、これに関連付けられたセッションは通常すべてタイムアウトし、これによって反復開発が困難になります。このパラメータを False に設定すると、現在のアプリケーションに対するユーザーのブラウザが閉じて再度開くまでの間、セッションタイムアウトにより使用できなくなるのを回避することができます。	×
generateUniqueURLs	boolean	ページ生成中の固有の URL の生成を有効または無効にします。固有の URL は、プロキシとブラウザの問題のあるキャッシュ戦略を覆すのに役立ちます。たとえば、ブラウザが動的ページを不適切にキャッシュしている場合は、この機能を有効にすることで、すべてのページが固有の URL から来るとブラウザは認識するため、通常は不適切にキャッシュされた動的ページをブラウザが表示しないようになります。この機能を有効にすると、実行時に少しリソースを消費するため、デフォルトでは無効になっています。	×

パラメータ名	型	説明	必須
logMessagePrefix	`String`	(ログメッセージをより目立たせるために) ログメッセージ接頭辞を設定します。	×
moduleURL	`String`	モジュールサーブレットの URL。詳細については、次の項を参照してください。	○
qualifiedViewNameSeparator	`char`	Web アプリケーションフレームワークの最初のバージョン以来、限定ビュー名を区切るためにピリオド (.) が使用されてきました。しかし、この区切り文字を使用すると、HTML ページで JavaScript を使いにくくなる可能性があります。このため、アプリケーション開発者はここで、下線 (_) のような代替値を設定できます。通常の子ビュー名では区切り文字を使用しないように注意してください。	×
showMessageBuffer	`boolean`	生成された HTML ページ下部のアプリケーションメッセージバッファの表示を有効または無効にします。この機能はデバッグには役立ちますが、配備時にはオフにする必要があります。	×
useTaglibTEI	`boolean`	J2EE コンテナの前バージョンには、Web アプリケーションフレームワーク TLD で宣言された TEI (Tag Extra Information) を生成する際に問題がありました。バージョン 2.1 からは、TEI はデフォルトで有効になっていますが、古いコンテナバージョンで動作するアプリケーションでは、互換性を維持するためにこの機能を無効にできます。	×

モジュール URL の構成

各モジュールサーブレットには、モジュール URL が関連付けられています。この URL は標準サーブレット URL パスマッピングで、アプリケーションの配備記述子 (または等価物) で構成する必要があります。Web アプリケーションフレームワークアプリケーションの各ページが生成されるときは、Web アプリケーションフレームワークタグライブラリが現在のモジュールの URL への参照を HTML 出力に生成するため、送付されたフォームや起動されたリンクは、元の出力を生成したモジュールと ViewBean に戻ります。このため、各モジュールのモジュール URL は、これに Web アプリケーションフレームワークインフラストラクチャがアクセスできるように標準的に構成する必要があります。さらに、アプリケーション内のモジュール間ナビゲートを可能にするには、各モジュールサーブレットのモジュール URL を他の各モジュールサーブレットが使用できることが必要です。このため、モジュール URL は、アプリケーションを実行する前に構成すべき唯一のモジュールサーブレットパラメータとなります。

モジュールサーブレットパラメータ moduleURL は他のモジュールサーブレットパラメータと同じように構成できますが、以下のような付加的な制限があります。

- moduleURL パラメータは、ただ単にパッケージ固有である必要があります (アスタリスクワイルドカードを使ってパラメータを命名する必要があります)。

moduleURL パラメータは、パッケージ名が同一の各モジュールの ViewBean によっても使用されるため、以下のように命名する必要があります。

```
<param-name>jato:[module package name].*:moduleURL</param-name>
```

- moduleURL パラメータは、サーブレット初期パラメータではなく、コンテキストパラメータとして設定する必要があります。

  他のモジュールサーブレットパラメータはコンテキストパラメータまたはサーブレット初期パラメータとして設定できますが、moduleURL パラメータはコンテキストパラメータとして設定する必要があります (例については以下を参照してください)。この制限の理由は、各モジュールサーブレットの moduleURL パラメータは他のすべてのモジュールサーブレットが使用できる必要があるからです。

- moduleURL パラメータの値は、モジュールサーブレット URL マッピングに対応していることが必要です。

  Web アプリケーションフレームワークページに生成される URL は、相対 URL です。標準 Web アプリケーションフレームワーク URL 書式では URL パス末尾にページ名を指定するため、moduleURL は以下のように設定することが必要です。

```
<context-param>
    <param-name>jato:[app package].[module package].*:moduleURL</param-name>
    <param-value>../[module package]</param-value>
</context-param>
```

同様に、モジュールサーブレット URL は、以下の URL にマッピングする必要があります。

```
<servlet-mapping>
    <servlet-name>[servlet name]</servlet-name>
    <url-pattern>/[module package]/*</url-pattern>
</servlet-mapping>
```

厳密に言えば、モジュールの URL パスはモジュールパッケージの名前をもとに命名する必要はなく、任意の名前を指定できます。これは簡素化のために推奨されている命名方法ですが、主たる制約は、moduleURL パラメータで使用される URL パスは、サーブレットマッピング URL パターンで使用されるものと同じにする、ということです。

前述の規則を踏まえて、以下のケースを考えてみましょう。

- 基本アプリケーションパッケージ : `fooapp`
- パッケージのモジュール : `fooapp.main`

基本アプリケーションは、以下の配備記述子を備えています。

```
<web-app>

    <context-param>
        <param-name>jato:fooapp.main.*:moduleURL</param-name>
        <param-value>../main</param-value>
    </context-param>

    <servlet>
        <servlet-name>MainModuleServlet</servlet-name>
        <servlet-class>fooapp.main.MainModuleServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>MainModuleServlet</servlet-name>
        <url-pattern>/main/*</url-pattern>
    </servlet-mapping>

...

</web-app>
```

Module2 という名前の 2 番目のモジュールをこのアプリケーションに追加すると、以下のようになります。

```
<web-app>

    <context-param>
        <param-name>jato:fooapp.main.*:moduleURL</param-name>
        <param-value>../main</param-value>
    </context-param>

    <context-param>
        <param-name>jato:fooapp.module2.*:moduleURL</param-name>
        <param-value>../module2</param-value>
    </context-param>

    <servlet>
        <servlet-name>MainModuleServlet</servlet-name>
        <servlet-class>fooapp.main.MainModuleServlet</servlet-class>
    </servlet>

    <servlet>
        <servlet-name>Module2Servlet</servlet-name>
        <servlet-class>fooapp.module2.Module2Servlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>MainModuleServlet</servlet-name>
        <url-pattern>/main/*</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>Module2Servlet</servlet-name>
        <url-pattern>/module2/*</url-pattern>
    </servlet-mapping>

    ...

</web-app>
```

ViewBean 表示 URL の構成

Web アプリケーションフレームワークアプリケーションの各 ViewBean はピア JSP を備えており、この JSP の URL は ViewBean の「表示 URL」と呼ばれます。それぞれの ViewBean は自己が使用するためのデフォルト表示 URL を備えており、この URL は `ViewBean.getDefaultDisplayURL()` および

ViewBean.setDefaultDisplayURL(String) メソッドを通して使用および設定できます。開発者は通常 ViewBean のコンストラクタでデフォルト表示 URL を設定します。

しかし、アプリケーション配備時に ViewBean のデフォルト表示 URL をオーバーライドすることが有用または必要な場合があります。この場合は、ViewBean のデフォルト表示 URL を実行時に設定する以下のコンテキストパラメータを配備記述子に設定できます。

パラメータ名	型	説明
defaultDisplayURL	String	指定された ViewBean の表示 URL

このパラメータは、モジュールサーブレットパラメータについて前述されている命名規則を使って設定します (ただし、これはモジュールサーブレットパラメータではありません)。

```
<context-param>
    <param-name>jato:fooapp.module1.FooViewBean:defaultDisplayURL</param-name>
    <param-value>/fooapp/module1/Foo-Alternate.jsp</param-value>
</context-param>
```

注 – このメカニズムを使って ViewBean 表示 URL をオーバーライドすると多少のオーバーヘッドが発生するため、通常は推奨されません。可能であれば、この方法の代わりに適切な表示 URL を直接指定して影響を受ける ViewBean を再コンパイルしてください。

SQLConnectionManager の構成

Web アプリケーションフレームワークには、SQLConnectionManager というマネージャオブジェクトが含まれています。このオブジェクトのインスタンスは、任意の所定の要求で RequestContext からアプリケーションで使用できます。このクラスの目的は、開発者と組み込み型 Web アプリケーションフレームワークオブジェクトが JDBC 接続オブジェクトを容易に取得できるようにすることです。

SQLConnectionManager の基本的な使い方では、呼び出し元は「データソース名」を使って JDBC 接続をこれに要求します。このデータソース名は、事前に構成されたデータベース接続の任意の論理名です。またこのクラスは、接続を取得するためのテクニックに関係なく、このタスクを標準化します。さらにクラスは、開発、テスト、配備環境を切り替えるときに有用な間接参照のレベルを提供します。

標準 Java アプリケーションでは、JDBC 接続は通常 `java.sql.DriverManager` への呼び出しによって取得します。呼び出し元は使用したいデータベースに固有の JDBC URL を指定し、`DriverManager` が利用可能な JDBC ドライバを指定の URL と照合して、ドライバからデータベース接続を取得し、これを呼び出し元に返します。そして呼び出し元は、その接続を閉じるまでの必要な期間だけ接続を使用します。

Web アプリケーションでは、`DriverManager` を使用すると、データベースアクセスを必要とする各アプリケーション要求のために新しいデータベース接続を開いて初期化することが必要とされるため、非常に非効率的です。ただし、JDBC 2.0 標準拡張 (`javax.sql` パッケージ) では、データベース接続プール用の標準 J2EE メカニズムが定義されています。これによって複数の要求で接続を再利用することが可能になり、データベースへの接続を繰り返し開くという非効率を回避できます。

Web アプリケーション開発者は、JDBC 2.0 標準拡張が提供する JNDI メカニズムを通して、プールされた JDBC 接続オブジェクトを取得できます。このメカニズムでは、JNDI コンテキストを割り当てて、名前によってこれに「データソース」を要求します。この名前 (データソース名) は、`"jdbc/"` から開始する標準書式で指定します。ここでは、アプリケーション配備者は JDBC URL をアプリケーションに直接埋め込む代わりに、ホスト、プロトコル、ユーザー名、パスワードといった必要なすべての情報を指定して JDBC データソースを事前に構成し、`"jdbc/"` という接頭辞で開始する論理データソース名で使用できるようにします。アプリケーション開発者は、この論理データソース名を参照するだけです。この論理データソース名は、アプリケーションが配備されるコンテナで適切にマップされるとみなします。

しかし、すべてのコンテナがこのデータソースメカニズムを提供するわけではなく、データソース名に特定の制限を課すコンテナもあります。たとえば、あるコンテナでは標準 `"jdbc/"` 接頭辞の後に任意の名前を付けることができますが、別のコンテナでは接頭辞の後にアプリケーションコンテキスト名を追加することが必要とされます。これは、同じコンテナで開発と配備が行われるアプリケーションでは問題になりません。しかし、アプリケーションの開発と配備を別々のコンテナで行うのは非常に一般的であるため問題が生じる可能性があります。

ここで `SQLConnectionManager` が役立ちます。`SQLConnectionManager` は自身のデータソースマッピングメカニズムを備えており、これによって開発者はアプリケーションで本当に任意のデータソース名を使用できることに加え、これらの名前をあらゆるコンテナ内で操作可能にできます。さらに、`SQLConnectionManager` では、データソース名を JNDI データソース名またはプレーンな JDBC URL のいずれかにマッピングできます。この機能によって Web アプリケーションフレームワークアプリケーションは、接続プールの利点はなくなっても、JDBC 2.0 標準拡張をサポートしていないコンテナで動作することが可能になります。

アプリケーションでは、`getConnection()` または `obtainConnection()` メソッドを使って、SQLConnectionManager から JDBC 接続を直接取得できます。静的な obtainConnection() メソッドは、アプリケーション初期化時など、要求スコープ外の接続を取得するために使用します。また、必要であれば `SQLConnectionManager` をバイパスして JNDI 検索 (または `DriverManager`) を直接使うこともできますが、

QueryModelBase といった JDBC を使用する Web アプリケーションフレームワーククラスでは、常に SQLConnectionManager を使ってデータベース接続を取得します。

JNDI または JDBC DriverManager を使用するための SQLConnectionManager の設定

前述したように、SQLConnectionManager では、任意のデータソース名を使って JDBC 接続を取得する際に、プレーン JDBC URL (JDBC DriverManager 経由) または JNDI データソース名のいずれかを使用できます。これら 2 つのモードは相互に排他的なもので、現在のモードは SQLConnectionManagerBase.setUsingJNDI(Boolean) 静的メソッドを呼び出すことで選択します。このメソッドは、1 度だけ呼び出す必要があります。アプリケーションサーブレットクラスの静的初期化子から、このメソッドを呼び出します。

true という値を指定して setUsingJNDI() メソッドを呼び出すと、データソース名の JNDI 検索が有効になります。このモードでは、すべてのデータソース名が一般書式 "jdbc/..." の JNDI データソース名にマップされていると想定されます。false という値を指定して setUsingJNDI() メソッドを呼び出すと、SQLConnectionManager はデータソース名は JDBC URL に直接マップされていると想定し、JDBC DriverManager から JDBC 接続を直接取得しようとします。

本稼働用のアプリケーションでは、どのような場合も、決して DriverManager を使って JDBC 接続を取得しないでください。この使い方をすると、接続を使用するたびに新しいデータベース接続が開かれ、アプリケーションのパフォーマンスとスケーラビリティに深刻な問題が発生します。アプリケーションを本稼働用に配備するときは、必ずコンテナのデータベース接続プールとともに SQLConnectionManager の JNDI 検索メカニズムを使用してください。

データソースマッピングの追加

前述のモードのいずれでも、指定のデータソース名のマッピングが存在するものと想定されます。これらのマッピングは、SQLConnectionManagerBase.addDataSourceMapping(String, String) 静的メソッドを呼び出すことで設定されます。このメソッドは通常、アプリケーションサーブレットクラスの静的初期化子内から呼び出します。

以下に例を示します。

```
static
{
    setUsingJNDI(true);

    // "jdbc/MyDS" という名前で接続を要求すると、
    // SQLConnectionManager が JNDI 検索を実行して、
    // 名前が "jdbc/Foo/MyDS-Test" のデータソースを検索
    addDataSourceMapping("jdbc/MyDS","jdbc/Foo/MyDS-Test");
}
```

または

```
static
{
    setUsingJNDI(false);

    // JNDI を使用しない場合は、標準的な方法で
    // JDBC ドライバを初期化する必要があることに注意
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Class.forName("oracle.jdbc.driver.OracleDriver");

    // "jdbc/MyDS" という名前で接続を要求すると、
    // SQLConnectionManager が "jdbc:odbc:northwind" という URL で
    // JDBC DriverManager から接続を取得
    addDataSourceMapping("jdbc/MyDS","jdbc:odbc:northwind");
    addDataSourceMapping("jdbc/HerDS","jdbc:oracle:thin:@10.0.0.1:1521:foo");
}
```

データソースマッピングが見つからない場合

所定データソース名のマッピングが見つからない場合、SQLConnectionManager は代替メカニズムを提供します。マッピングが見つからない場合は、接続を取得するために使用すべきデータソース名は JNDI データソース名または JDBC URL であると SQLConnectionManager は想定します。その結果、デフォルトメカニズムでは、SQLConnectionManager に渡されたリテラルなデータソース名が使用されます。つまり、開発者がマッピングを追加しない限り、SQLConnectionManager の使用は完全に透過になります。

アプリケーションのパッケージ

Web アプリケーションフレームワークアプリケーションのパッケージングは、WAR ファイル用の標準 J2EE ガイドラインに従って行います。このガイドの推奨事項に従ってアプリケーションを開発した場合は、JAR または Zip ツールを使ってアプリケーションディレクトリ構造を WAR ファイルに格納するだけで済みます。`WEB-INF` ディレクトリおよびそのピアディレクトリとファイルは、アーカイブのルートに配置します。WAR ファイルには任意の名前を指定できますが、`.war` というファイル拡張子を付ける必要があります。

アプリケーションの配備

前述のようにアプリケーションを構成したら、他の WAR ベースの J2EE Web アプリケーションと同じように Web アプリケーションフレームワークアプリケーションを配備できます。J2EE コンテナは、アプリケーションを配備する際の細かい点が異なります。詳細は、使用するコンテナの配備マニュアルを参照してください。ただし、多くの場合は、WAR ファイルをコンテナの `/webapps` ディレクトリにコピーするだけです。

以下は、配備に関する追加の注意事項です。

- 配備される各アプリケーションは、その WEB-INF/lib ディレクトリに Web アプリケーションフレームワーク実行時の自身のコピーを必要とします。サポートされている唯一の構成であることに加え、この手段によってアプリケーションは完全に自己充足型になり、どのコンテナにも直ちにインストールすることが可能になります。また、これにより、Web アプリケーションフレームワークバージョンの変更によって複数のアプリケーションが影響を受けることがなくなります。複数アプリケーションで実行時の共有コピーが使用されていると、これらすべてのアプリケーションを新しい実行時バージョンに同時にアップグレードすることが必要になります。
- アプリケーションのソースコードは、可能なときに、配備された WAR ファイル (`/WEB-INF` ディレクトリの下) にパッケージ化します。これは正式な推奨事項ではありませんが、いくつかの利点があります。まず、これは実際にアプリケーションで使用されているソースの正確なイメージを提供することで、障害追跡の助けとなります。次に、これによって開発/ソース制御のための完全な環境を必要としないで、配備されたアプリケーションに重要な修正を簡単に行うことが可能になります。
- アプリケーションで JDBC を使用する場合は、SQLConnectionManager で JNDI モードを使用していることを配備前にチェックします。

- すべての .jsp ファイルに対する外部クライアントからのアクセスを拒否するように、コンテナや Web サーバーを構成できます。これらのファイルは、Web アプリケーションフレームワークサーブレットを通してのみアクセスするように意図されています。

Web アプリケーションフレームワークアプリケーションへのアクセス

Web アプリケーションフレームワークアプリケーションを配備したら、HTTP クライアントからこれにアクセスすることが可能になります。Web アプリケーションフレームワークアプリケーションでは、以下の一般的な標準 URL 書式を使用します。

```
http://<host + container-specific path>/<servlet context name>/
<module name>/<page name>
```

- 大部分の J2EE コンテナでは、*<servlet context name>* は、配備された WAR ファイルの名前です。
- *<module name>* は、アプリケーション内のモジュールパッケージの名前です。
- *<page name>* は、そのモジュール内で表示するページの名前です。

たとえば、以下のような名前があると仮定します。

- WAR ファイル名： `HelloWorld.war`
- モジュール名： `greeting`
- ページ名：`Hello`
  - ViewBean クラス： *<app package>*`.greeting.Hello`
  - JSP 名：`Hello.jsp`

この場合、Hello ページにアクセスするための URL は、以下のようになります。

```
http://<host + container-specific path>/HelloWorld/greeting/Hello
```

基本アプリケーションパッケージ名は、URL の要素になりません。さらに、要求されるページ名は、指定のモジュール内に存在する必要があります。要求されたモジュール外部のページにアクセスしようとするのは規則違反です。詳細については、89 ページの「モジュール間のナビゲート」を参照してください。

多くの J2EE コンテナでは、サーブレットエンジンにアクセスするためにコンテナ固有のパスを使用する必要はありません。たとえば、Sun Java System Application Server、Apache Tomcat、または Caucho Resin の場合は、以下の URL を使用できます。

```
http://<host>/HelloWorld/greeting/Hello
```

ただし、一部の J2EE コンテナでは、外部アクセスに従来の Web サーバーを使った多層アーキテクチャを使用するため、URL に付加的なパス情報を指定することが必要になる場合があります。たとえば、iPlanet Application Server 6.x のサンプル URL は、以下のようになります。

```
http://<host>/NASApp/HelloWorld/greeting/Hello
```

モジュール間のナビゲート

モジュール間のナビゲートとは、あるモジュール内で要求を処理する一方で、別のモジュールのページを使って応答することを示します。このシナリオは通常、アプリケーションの論理的に関連しているある領域から別の領域に移動する場合に起こります。

大部分は Web アプリケーションフレームワーク実行環境でこの切り替えを自動的に処理しますが、基本的な機構を理解しておくことが大切です。モジュール間を移動する際の重要な仕事は、クライアントに対して最終的に生成されるページに、そのページが含まれるモジュールへの参照が含まれるようにすることにあります。そうしないと、別のモジュールのページへの要求がサーバーに送信されて、セキュリティ例外が発生します。要求がモジュール境界を越えると、ターゲットモジュールのサーブレットが、ユーザーの要求を処理する次のサーブレットになります。たとえば、ユーザーの要求が `module1` の `PageOne` のイベントをトリガーし、このイベントが `module2` の `PageTwo` に要求を転送すると、`module2` のモジュールサーブレットが (別の要求が別のモジュール境界を越えるまで) 次と以降の全ユーザー要求を処理するサーブレットになります。

アプリケーション内のモジュール境界を越える際、開発者はある点に注意する必要があります。通常、開発者は `RequestContext` から利用できる `ViewBeanManager` を使って ViewBean 参照を取得します。`ViewBeanManager` を使用すると、開発者は `getLocalViewBean()` メソッドを介して検出したい `ViewBean` の短縮名を指定できます。ただし、このメソッドでは、クラス名を取得するために、指定の名前にパッケージ名を付加する必要があります。このパッケージ名は常に、要求を処理したモジュールサーブレットのパッケージ名になります。したがって、このメソッドを使って別のモジュールから `ViewBean` を取得することは不可能です。このショートカットメソッドを通して取得できるのは、現在のモジュール内の ViewBean だけです。別のモジュール内の ViewBean への参照を取得するには、クラスまたは完全限定クラス名を想定している他の `ViewBeanManager` メソッドの 1 つを使用してください。

付録 A

障害追跡

この付録では、既知の障害追跡について概説し、それぞれの既知の問題の状態、考えられる原因、考えられる解決策、およびコメントを示します。

状態

```
javax.servlet.ServletException: Invalid request - request
handler "X" not found at
com.iplanet.jato.ApplicationServletBase.onRequestHandlerNotFound
(...)
...
```

考えられる原因

ターゲットページのスペルミス。

考えられる解決策

この例では、ターゲットページを「*login*」ではなく「*Login*」と綴ります。

大文字、小文字や複数形のミスは、よく見られる間違いです。

コメント

ページ名は常に ViewBean クラス名に関連しているため、通常は大文字から開始します。

状態

```
javax.servlet.ServletException: The request context is null -
this page must be accessed through a servlet at
org.apache.jasper.runtime.PageContextImpl.handlePageException(Pa
geContextImpl.java:457)
...
```

考えられる原因

JSP の URL を使って Web アプリケーションフレームワーク JSP に直接アクセスしようとした。

考えられる解決策

Web アプリケーションフレームワークアプリケーションへのすべてのアクセスは、モジュールサーブレット (フロントコントローラ) を通して行う必要があります。この例外は、ユーザーが不適切なリソースにアクセスしようとしたことを示しています。

不適切な URL の例を以下に示します。

```
http://localhost:8081/JatoTutorial/jatotutorial/main/Login
```

上記の代わりに、以下のような URL を使用してください。

```
http://localhost:8081/JatoTutorial/main/Login
```

コメント

実際に有効な URL は、アプリケーションの `web.xml` ファイル内で URL/サーブレットマッピングがどのように構成されているかによって変わります。`web.xml` を手動で変更することなくアプリケーションを作成した場合は、URL パターンは以下のようになります。

```
http://<server>:<port>/<ServletContext>/<module>/<targetPageName
>
```

状態

```
com.iplanet.jato.NavigationException: Exception encountered
during forward
Root cause = [java.sql.SQLException: Cannot find the user
"ADMINISTRATOR".]
...
```

考えられる原因

これは具体的に言えば、データベースアクセス SQL 例外。この場合、モデルは適切なユーザー名やパスワードを指定しないでデータベースにアクセスしようとした。

考えられる解決策

(JNDI 検索テクニックではなく) JDBC URL を使用する場合は、モデルのコンストラクタに明示的に設定することで適切なユーザー名とパスワードを指定してください (PointBase サンプルデータベースの例を以下に示します)。

```
setDefaultConnectionUser("pbpublic");
setDefaultConnectionPassword("pbpublic");
```

場合によっては、JDBC URL でユーザー名とパスワードを指定することもできます。

コメント

この問題は、JNDI データベース接続検索を使用することで回避できます。46 ページの「SQLConnectionManager の使用」を参照してください。

索引