BEA WebLogic Server™

BEA WebLogic Express™

Web アプリケーションの アセンブルとコンフィグレーション

BEA WebLogic Server バージョン 6.1
マニュアルの日付：2002 年 6 月 24 日

著作権

限定的権利条項

商標または登録商標

Web アプリケーションのアセンブルとコンフィグレーション

パート番号	マニュアルの日付	ソフトウェアのバージョン
なし	2002 年 6 月 24 日	BEA WebLogic Server バージョン 6.1 SP3

目次

このマニュアルの内容

1. Web アプリケーションの基本事項

2. Web アプリケーションのデプロイメント

4. Web アプリケーションにおけるセッションとセッション永続性の使用

5. Web アプリケーションでのセキュリティのコンフィグレーション

6. アプリケーション イベントとリスナ

7. フィルタ

8. Web アプリケーションのデプロイメント記述子の記述

A. web.xml デプロイメント記述子の要素

B. weblogic.xml デプロイメント記述子の要素

索引

このマニュアルの内容

このマニュアルでは、J2EE Web アプリケーションをアセンブルおよびコンフィグレーションする方法について説明します。

このマニュアルの構成は次のとおりです。

- 第 1 章「Web アプリケーションの基本事項」では、WebLogic Server における Web アプリケーションの使い方について概説します。
- 第 2 章「Web アプリケーションのデプロイメント」では、Web アプリケーションを WebLogic Server にデプロイする方法について説明します。
- 第 3 章「Web アプリケーション コンポーネントのコンフィグレーション」では、Web アプリケーション コンポーネントをコンフィグレーションする方法について説明します。
- 第 4 章「Web アプリケーションにおけるセッションとセッション永続性の使用」では、Web アプリケーションで HTTP セッションとセッション永続性を使用する方法について説明します。
- 第 5 章「Web アプリケーションでのセキュリティのコンフィグレーション」では、Web アプリケーションで認証と認可をコンフィグレーションする方法について説明します。
- 第 6 章「アプリケーション イベントとリスナ」では、Web アプリケーションで J2EE イベント リスナを使用する方法について説明します。
- 第 7 章「フィルタ」では、Web アプリケーションでフィルタを使用する方法について説明します。
- 第 8 章「Web アプリケーションのデプロイメント記述子の記述」では、Web アプリケーション デプロイメント記述子を手動で記述する方法について説明します。
- 付録 A「web.xml デプロイメント記述子の要素」では、`web.xml` デプロイメント記述子用のデプロイメント記述子要素のリファレンスを提供します。

- 付録 B「weblogic.xml デプロイメント記述子の要素」では、`weblogic.xml` デプロイメント記述子用のデプロイメント記述子要素のリファレンスを提供します。

対象読者

このマニュアルは、Sun Microsystems の Java 2 Platform, Enterprise Edition（J2EE）を使った e- コマース アプリケーションを構築するアプリケーション開発者を対象としています。Web テクノロジ、オブジェクト指向プログラミング手法、および Java プログラミング言語に読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA 製品のドキュメントは、BEA の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックします。

このマニュアルの印刷方法

Web ブラウザの [ファイル｜印刷] オプションを使用すると、Web ブラウザからこのマニュアルのメイン トピックを一度に 1 つずつ印刷できます。

このマニュアルの PDF 版は、Web サイトで入手できます。PDF を Adobe Acrobat Reader で開くと、マニュアルの全体（または一部分）を書籍の形式で印刷できます。PDF を表示するには、WebLogic Server ドキュメントのホーム ページを開き、[ドキュメントのダウンロード] をクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader は、Adobe の Web サイト（http://www.adobe.co.jp）から無料で入手できます。

関連情報

BEA の Web サイトでは、WebLogic Server の全マニュアルを提供しています。以下の WebLogic Server ドキュメントには、WebLogic Server アプリケーションのコンポーネントの作成に関連する情報が含まれています。

- 『WebLogic HTTP サーブレット プログラマーズ ガイド』
- 『WebLogic JSP プログラマーズ ガイド』
- 『WebLogic Server Web サービス プログラマーズ ガイド』
- 『WebLogic Server アプリケーションの開発』

Java アプリケーションの開発に関する一般情報については、Sun Microsystems, Inc. の Java 2, Enterprise Edition Web サイト（http://java.sun.com/products/j2ee/）を参照してください。

サポート情報

BEA のドキュメントに関するユーザからのフィードバックは弊社にとって非常に重要です。質問や意見などがあれば、電子メールで docsupport-jp@bea.com までお送りください。寄せられた意見については、ドキュメントを作成および改訂する BEA の専門の担当者が直に目を通します。

電子メールのメッセージには、ご使用のソフトウェア名とバージョン名、およびマニュアルのタイトルと作成日付をお書き添えください。本バージョンの BEA WebLogic Server について不明な点がある場合、または BEA WebLogic Server のインストールおよび動作に問題がある場合は、BEA WebSUPPORT（http://www.bea.com）を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

カスタマ サポートでは以下の情報をお尋ねしますので、お問い合わせの際はあらかじめご用意ください。

- お名前、電子メール アドレス、電話番号、ファクス番号

- 会社の名前と住所
- お使いの機種とコード番号
- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+〔Tab〕	同時に押すキーを示す。
斜体	強調または本のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、Java クラス、データ型、ディレクトリ、およびファイル名とその拡張子を示す。等幅テキストはキーボードから入力するテキストも示す。
	例:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

斜体の等幅テキスト	コード内の変数を示す。 例: `String` *`CustomerName`*`;`

表記法	適用
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 例 : LPT1 BEA_HOME OR
{ }	構文内の複数の選択肢を示す。
[ ]	構文内の任意指定の項目を示す。 例 : `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
|	構文の中で相互に排他的な選択肢を区切る。 例 : `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
. . .	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる。 ■ 任意指定の引数が省略されている。 ■ パラメータや値などの情報を追加入力できる。
. . .	コード サンプルまたは構文で項目が省略されていることを示す。

1　Web アプリケーションの基本事項

この章では、Web アプリケーションをコンフィグレーションおよびデプロイする方法について説明します。

Web アプリケーションの概要

Web アプリケーションには、サーブレット、JavaServer Pages（JSP）、JSP タグライブラリなどのアプリケーションのリソースと、HTML ページや画像ファイルなどの静的リソースが組み込まれています。また、Web アプリケーションは、エンタープライズ JavaBean（EJB）など、アプリケーションの外部にあるリソースへのリンクも定義できます。WebLogic Server にデプロイされる Web アプリケーションは、標準の J2EE デプロイメント記述子と WebLogic 固有の省略可能なデプロイメント記述子を一緒に使用して、リソースとそれらの操作パラメータを定義します。

JSP ページと HTTP サーブレットは、WebLogic Server で使用可能なすべてのサービスと API にアクセスできます。これらのサービスには、EJB、Java Database Connectivity（JDBC）を介したデータベース接続、JavaMessaging Service（JMS）、XML などがあります。

Web アプリケーションは J2EE 仕様で定義されている標準ディレクトリ構造を採用しており、このディレクトリ構造を使用するファイルの集合としてデプロイされるか（この種のデプロイメントを展開ディレクトリ形式と呼ぶ)、または `.war` ファイルというアーカイブ ファイルとしてデプロイされます。展開ディレ

クトリ形式による Web アプリケーションのデプロイは、主にアプリケーションの開発時に行います。.war ファイルによる Web アプリケーションのデプロイは、主にプロダクション環境で行います。

WAR ファイルは、単独でデプロイすることも、他のアプリケーション コンポーネントと共にエンタープライズ アーカイブ（EAR ファイル）にパッケージ化することもできます。単独でデプロイする場合は、アーカイブ名の最後は .war 拡張子でなければなりません。EAR ファイルでデプロイする場合は、アーカイブ名の最後は .ear 拡張子でなければなりません。（注意 : ディレクトリ全体をデプロイする場合は、ディレクトリに .ear、.war、.jar などの名前を付けることはできません。）

Web アプリケーション作成の主な手順

次に示すのは、Web アプリケーション作成の手順をまとめたものです。Web アプリケーションの作成とコンフィグレーションに、BEA が提供する開発者向けツールを使用することもできます。詳細については、1-6 ページの「Web アプリケーション開発者向けツール」を参照してください。

Web アプリケーションを作成するには、次の手順に従います。

1. リソース（サーブレット、JSP、静的ファイル、およびデプロイメント記述子）を指定のディレクトリ形式に従って配置します。詳細については、1-4 ページの「ディレクトリ構造」を参照してください。
2. Web アプリケーション デプロイメント記述子（web.xml）を作成します。この手順では、サーブレットの登録、サーブレット初期化パラメータの定義、JSP タグ ライブラリの登録、セキュリティ制約の定義、およびその他の Web アプリケーション パラメータの定義を行います（Web アプリケーションのさまざまなコンポーネントに関する情報は、このドキュメント全体にわたって記載されています）。デプロイメント記述子を Web アプリケーションの WEB-INF ディレクトリに配置します。

   手順の詳細については、8-3 ページの「web.xml デプロイメント記述子の記述」を参照してください。

   Web アプリケーション デプロイメント記述子の編集には、Administration Console を使用することもできます。詳細については、「Web アプリケーション デプロイメント記述子エディタ ヘルプ」を参照してください。

3. WebLogic 固有のデプロイメント記述子（weblogic.xml）を作成します。この手順では、JSP プロパティ、JNDI マッピング、セキュリティ ロール マッピング、および HTTP セッション パラメータを定義します。このファイルに属性を定義する必要がない場合は、このファイルを作成する必要はありません。

   WebLogic 固有のデプロイメント記述子の詳しい作成手順については、8-23 ページの「WebLogic 固有のデプロイメント記述子（weblogic.xml）の記述」を参照してください。

   Web アプリケーション デプロイメント記述子の編集には、Administration Console を使用することもできます。詳細については、「Web アプリケーション デプロイメント記述子エディタ ヘルプ」を参照してください。

4. 上記のディレクトリ構造を .war ファイルにアーカイブします。アーカイブ化は、Web アプリケーションをプロダクション環境にデプロイする準備が整ったときにだけ行います（開発中には、展開ディレクトリ形式でアプリケーションを開発して、Web アプリケーションの個別コンポーネントを更新するほうが便利な場合があります）。.war アーカイブを作成するには、Web アプリケーションが格納されているルート ディレクトリから次のコマンドを使用します。

   ```
   jar cv0f myWebApp.war.
   ```

   このコマンドを実行すると、myWebApp.war という Web アプリケーション アーカイブ ファイルが作成されます。

5. （省略可能）。Web アプリケーションをエンタープライズ アプリケーションにアーカイブします。詳細については、2-19 ページの「エンタープライズ アプリケーションの一部としての Web アプリケーションのデプロイ」を参照してください。

6. Web アプリケーションを WebLogic Server にデプロイします。この最後の手順で、Web アプリケーションは WebLogic Server 上のリクエストにサービスを提供するようにコンフィグレーションされます。手順の詳細については、2-1 ページの「Web アプリケーションのデプロイメント」を参照してください。

ディレクトリ構造

Web アプリケーションは、指定されたディレクトリ構造の中で開発します。これにより、Web アプリケーションをアーカイブして、WebLogic Server または別の J2EE 対応サーバにデプロイできるようになります。Web アプリケーションに属するすべてのサーブレット、クラス、静的ファイル、およびその他のリソースは、ディレクトリ階層に基づいて配置されます。この階層のルートは、Web アプリケーションのドキュメント ルートを定義します。このルート ディレクトリの下に置かれたファイルは、WEB-INF という特別なディレクトリの下にあるファイルを除き、すべてクライアントに提供されます。Web アプリケーションの名前は、Web アプリケーションのコンポーネントに対するリクエストを解決するために使われます。

非公開ファイルは、ルート ディレクトリの下の WEB-INF ディレクトリに配置します。WEB-INF の下のすべてのファイルは公開されず、クライアントには提供されません。

`WebApplicationName\`

このディレクトリ（またはサブディレクトリ）には、HTML ファイルなどの静的ファイルと JSP ファイルを配置します。このディレクトリは、Web アプリケーションのドキュメント ルートです。

`\WEB-INF\web.xml`

Web アプリケーションをコンフィグレーションするデプロイメント記述子です。

`\WEB-INF\weblogic.xml`

WebLogic 固有のデプロイメント記述子ファイルです。このファイルには、web.xml ファイルに記述されたリソースを WebLogic Server 内の別の場所に存在するリソースにマップする方法が定義されます。またこのファイルは、JSP および HTTP セッション属性を定義するために使用されます。

`\WEB-INF\classes`

HTTP サーブレットやユーティリティ クラスなどのサーバサイド クラスが格納されます。

`\WEB-INF\lib`

JSP タグ ライブラリなど、Web アプリケーションによって使用される .jar ファイルが格納されます。

URL と Web アプリケーション

クライアントから Web アプリケーションにアクセスするために使用する URL は、次のパターンで作成します。

`http://`*`hoststring/ContextPath/servletPath/pathInfo`*

各要素の説明は次のとおりです。

`hoststring`
: 仮想ホストにマップされるホスト名または `hostname:portNumber`。

`ContextPath`
: Web アプリケーションの名前。

`servletPath`
: `servletPath` にマップされるサーブレット。

`pathInfo`
: URL の残りの部分（通常はファイル名）。

仮想ホスティングを使用している場合、URL の *`hoststring`* の部分を仮想ホスト名に置き換えることができます。

詳細については、「WebLogic Server による HTTP リクエストの解決方法」を参照してください。

Web アプリケーション開発者向けツール

BEA では、Web アプリケーションの作成とコンフィグレーションを支援するツールを提供しています。この節では、これらのツールについて説明します。

スケルトン デプロイメント記述子を作成する ANT タスク

スケルトン デプロイメント記述子を作成するときに、WebLogic ANT ユーティリティを利用できます。ANT ユーティリティは WebLogic Server 配布キットと共に出荷されている Java クラスです。ANT タスクによって、Web アプリケーションを含むディレクトリが調べられ、その Web アプリケーションで検出されたファイルを基にデプロイメント記述子が作成されます。ANT ユーティリティは、個別の Web アプリケーションに必要なコンフィグレーションやマッピングに関する情報をすべて備えているわけではないので、ANT ユーティリティによって作成されるスケルトン デプロイメント記述子は不完全なものです。ANT ユーティリティがスケルトン デプロイメント記述子を作成した後で、テキスト エディタ、XML エディタ、または Administration Console を使ってデプロイメント記述子を編集し、Web アプリケーションのコンフィグレーションを完全なものにしてください。

ANT ユーティリティを使ってデプロイメント記述子を作成する方法の詳細については、「Web アプリケーションのパッケージ化」を参照してください。

Web アプリケーション デプロイメント記述子エディタ

WebLogic Server の Administration Console には、統合されたデプロイメント記述子エディタがあります。この統合エディタを使用する前に、少なくともスケルトン `web.xml` デプロイメント記述子を作成しておく必要があります。

詳細については、「Web アプリケーション デプロイメント記述子エディタ ヘルプ」を参照してください。

XML エディタ

BEA では、XML ファイルの作成と編集のために簡単で使いやすい Ensemble のツールを用意しました。このツールを使うと、指定した DTD または XML スキーマに従って XML コードの有効性を検証できます。この XML エディタは、Windows または Solaris のマシンで使用でき、BEA dev2dev からダウンロードできます。

2 Web アプリケーションのデプロイメント

この章では、Web アプリケーションをデプロイする方法について説明します。

Web アプリケーションのデプロイメントの概要

Web アプリケーションのデプロイメントは最後の手順です。これによって WebLogic Server は Web アプリケーションのコンポーネントをクライアントに提供します。Web アプリケーションは、ユーザの環境と Web アプリケーションが

プロダクション環境に置かれるかどうかに基づいて、複数の手順のうちの 1 つでデプロイできます。WebLogic Server Administration Console や `weblogic.deploy` ユーティリティを使うことも、自動デプロイメントを使うことも可能です。

Web アプリケーションをデプロイする手順では、正しいディレクトリ構造を使用し、`web.xml` デプロイメント記述子を含み、必要であれば `weblogic.xml` デプロイメント記述子も含む、正常に作動する Web アプリケーションを作成していることを前提としています。Web アプリケーションの作成に必要な手順の概要については、「Web アプリケーション作成の主な手順」を参照してください。

WebLogic Server を実行するには、開発モードとプロダクション モードの 2 つのモードが利用できます。開発モードを有効にするには、`-Dweblogic.ProductionModeEnabled=false` フラグを指定して WebLogic Server を起動します（これはデフォルトの動作です）。プロダクション モードを有効にするには、`-Dweblogic.ProductionModeEnabled=true` フラグを指定して WebLogic Server を起動します。これらのモードの設定の詳細については、「WebLogic Server の起動と停止」を参照してください。

開発モードを指定すると、自動デプロイ機能を使用できます。自動デプロイ機能では、管理サーバの `applications` ディレクトリに Web アプリケーションをコピーすることによって、Web アプリケーションを短時間でデプロイできます。このディレクトリは、WebLogic Server インストール環境の `config\`*`mydomain`* ディレクトリ（*`mydomain`* は WebLogic Server ドメインの名前）にあります。WebLogic Server は `applications` ディレクトリにコピーされるすべてのアプリケーションを自動的にデプロイします。`.war` 形式の Web アプリケーションが変更された場合、WebLogic Server はその Web アプリケーションを自動的に再デプロイします。Web アプリケーションが展開形式の場合は、2-6 ページの「展開ディレクトリ形式でデプロイされた Web アプリケーションの再デプロイ」を参照してください。アプリケーションを `config/`*`mydomain`*`/applications` ディレクトリに格納しておくことはお勧めできません。このディレクトリにアプリケーションが既に存在していると、プロダクション モードであっても自動的にデプロイされます。

Web アプリケーションは、ファイル システムから、展開ディレクトリ形式、またはアーカイブされた `.war` 形式にデプロイできます。展開ディレクトリ形式とは、Web アプリケーション コンポーネントが Web アプリケーション用の標準ディレクトリ構造を使ってファイル システム上に整理されていることを意味し

ています。.war アーカイブは、Web アプリケーションの内容を展開ディレクトリ形式でアーカイブするために、Java jar ユーティリティを使って作成したものです。

管理対象の WebLogic Server のクラスタを使用している場合、1 つまたは複数の「対象となった」管理対象サーバに Web アプリケーションをデプロイするように指定することができます。管理サーバは、Web アプリケーションの対象になっているドメインにある管理対象サーバに Web アプリケーションを自動的にコピーします。

自動デプロイメントを使用した Web アプリケーションのデプロイメント

自動デプロイメントは Web アプリケーションの開発のために設計された便利な機能であり、プロダクション アプリケーションでの使用には適しません。アプリケーションをプロダクション環境に移動するには、2-8 ページの「プロダクション モードを有効にした Web アプリケーションのデプロイ」で説明している手動のデプロイメント手順を使用してください。管理サーバに Web アプリケーションをデプロイする方法は自動デプロイメントだけです。自動デプロイされた Web アプリケーションが管理対象サーバを対象としている場合、Web アプリケーションも管理対象サーバにデプロイされます。

自動デプロイメントを使用する前に、正しいディレクトリ構造を使用し、web.xml デプロイメント記述子を含み、必要であれば weblogic.xml デプロイメント記述子も含む、正常に作動する Web アプリケーションを作成する必要があります。Web アプリケーションは、アーカイブされた .war ファイルとして、または展開ディレクトリ形式でデプロイできます（1-4 ページの「ディレクトリ構造」を参照）。

自動デプロイメントを使用した Web アプリケーションのデプロイ方法

自動デプロイメントを使用して Web アプリケーションをデプロイするには、次の手順に従います。

1. `-DProductionModeEnabled=false` フラグを指定して WebLogic 管理サーバを起動します。詳細については、「WebLogic Server の起動と停止」を参照してください。
2. `.war` アーカイブ ファイルまたは Web アプリケーションを含むディレクトリを、ドメインの `applications` ディレクトリにコピーします。applications ディレクトリは WebLogic Server インストール環境の `config\mydomain` ディレクトリ（*`mydomain`* は WebLogic Server ドメイン）にあります。Web アプリケーションは管理サーバによって自動的にデプロイされます。
3. Administration Console を使用して Web アプリケーションをコンフィグレーションします。
   a. Web ブラウザで Administration Console を起動します。
   b. 左ペインの [デプロイメント] ノードを展開します。
   c. 左ペインの [Web アプリケーション] ノードを展開します。右ペインに Web アプリケーションのリストが表示され、その中には `applications` ディレクトリにコピーしたばかりの Web アプリケーションも入っています。
   d. Web アプリケーションの名前をクリックします。
   e. [対象] タブを選択し、適切なタブから対象サーバ、クラスタ、または仮想ホストを選択し、対象を [選択可] カラムから [選択済み] カラムに移動します。Web アプリケーションをサーバまたはクラスタの対象にしたときに、アプリケーションは任意の管理対象サーバにデプロイされます。Web アプリケーションを仮想ホストの対象にすると、Web アプリケーションが仮想ホスト経由でクライアントから利用できるようになります。
   f. [適用] をクリックします。

g. [ファイル] タブと [その他] タブで、別の属性をコンフィグレーションします。詳細については、「Web アプリケーション」の [コンフィグレーション] を参照してください。

h. [適用] をクリックします。

4. Web ブラウザからリソースをアクセスして、Web アプリケーションをテストします。次のように構築した URL からリソースをアクセスします。

   `http://myServer:myPort/myWebApp/resource`

   各要素の説明は次のとおりです。

   - *myServer* は、WebLogic Server のホスト マシンの名前。
   - *myPort* は、WebLogic Server がリクエストをリスンしているポート番号。
   - *myWebApp* は、Web アプリケーションのアーカイブ ファイルの名前（myWebApp.war など）または Web アプリケーションを含むディレクトリの名前。
   - *resource* は、JSP、HTTP サーブレット、HTML ページなどのリソースの名前。

Web アプリケーションの applications ディレクトリへのアップロード

Administration Console は、Web アプリケーションの applications ディレクトリへのアップロードにも使用できます。この手順は、WebLogic Server がリモート マシンにある場合に便利です。

Web アプリケーションをアップロードするには、次の手順に従います。

1. -DProductionModeEnabled=false フラグを指定して WebLogic 管理サーバを起動します。詳細については、「WebLogic Server の起動と停止」を参照してください。
2. 左ペインの [Web アプリケーション] ノードを選択します。
3. [新しい Web アプリケーションをインストール] をクリックします。
4. ファイル システム内の .war ファイルの格納場所に移動します。

5. [Upload] をクリックします。Web アプリケーションは管理サーバの applications ディレクトリにコピーされ、デプロイされます。

自動デプロイメントを使用した Web アプリケーションの再デプロイ

自動デプロイメントを使用している場合は、applications ディレクトリにデプロイされている Web アプリケーションの静的コンポーネント（JSP や HTML ページなど）を変更したなら、変更を反映するために、Web アプリケーションを再デプロイする必要があります。その手順は、.war アーカイブ ファイルでデプロイされた Web アプリケーションと、展開ディレクトリ形式でデプロイされた Web アプリケーションとでは異なります。

.war アーカイブでの Web アプリケーションの再デプロイ

アーカイブ ファイルを変更すると、Web アプリケーションの再デプロイメントが自動的に開始されます。自動デプロイされた Web アプリケーションが管理対象サーバを対象としている場合、Web アプリケーションもその管理対象サーバに再デプロイされます。

展開ディレクトリ形式でデプロイされた Web アプリケーションの再デプロイ

自動デプロイメントを使用している場合、展開ディレクトリ形式でデプロイした Web アプリケーションの再デプロイは、REDEPLOY と呼ばれる特殊ファイルを変更して、または Administration Console を使用して行うことができます。または、新しいバージョンのクラス ファイルを WEB-INF/classes ディレクトリにある古いファイルの上にコピーすることで、部分的な再デプロイを行うこともできます。

REDEPLOY ファイルの変更

REDEPLOY ファイルを変更して Web アプリケーションを再デプロイするには、次の手順に従います。

1. REDEPLOY という空のファイルを作成して、Web アプリケーションの WEB-INF ディレクトリに格納します（ディレクトリが存在しない場合は作成します）。
2. REDEPLOY ファイルを変更します。まずファイルを開き、内容を変更して（スペースを挿入するのが最も簡単な方法）、ファイルを保存します。また UNIX マシンでは、REDEPLOY ファイルに対して touch コマンドを使用することもできます。次に例を示します。

   ```
   touch
   user_domains/mydomain/applications/DefaultWebApp/WEB-INF/REDEPL
   OY
   ```

   REDEPLOY ファイルが変更されると、Web アプリケーションはすぐに再デプロイされます。

Administration Console を使った再デプロイ

Administration Console を使用して Web アプリケーションを再デプロイするには、次の手順に従います。

1. 左ペインの [デプロイメント] ノードを展開します。
2. [Web アプリケーション] ノードを選択します。
3. 再デプロイする Web アプリケーション ノードを選択します。
4. 右ペインの [デプロイ] ボックスの選択を解除します。
5. [適用] をクリックします。
6. 右ペインの [デプロイ] ボックスをチェックします。
7. [適用] をクリックします。

ホット デプロイメント

WEB-INF/classes ディレクトリにあるファイルの再デプロイは、次の方法で行います。クラスが WEB-INF/classes にデプロイされている場合は、古いファイルより後のタイム スタンプを持つ新しいバージョンのファイルをコピーするだけで、Web アプリケーションは、WEB-INF/classes フォルダ内のすべてを新しいクラスローダで再ロードします。

WLS がファイル システムを調べる頻度はコンソールで制御します。[デプロイメント | Web アプリケーション] で Web アプリケーションを選択します。[コンフィグレーション] タブの [ファイル] サブタブに移動し、[再ロード間隔（秒）] に秒数を入力します。

プロダクション モードを有効にした Web アプリケーションのデプロイ

プロダクション モードを有効にして Web アプリケーションをデプロイするには、`-DProductionModeEnabled=true` フラグを指定して WebLogic 管理サーバを起動する必要があります。この機能は WebLogic 管理サーバでは使用できますが、管理対象サーバでは使用できません。詳細については、「WebLogic Server の起動と停止」を参照してください。

Web アプリケーションは、Administration Console または `weblogic.deploy` ユーティリティを使用してデプロイできます。Web アプリケーションは、デプロイすると、ドメイン内で対象とした管理対象サーバすべてに自動的にデプロイされます（管理対象サーバの管理の詳細については、「WebLogic Server とクラスタのコンフィグレーション」を参照してください）。

`config/`*`mydomain`*`/applications` ディレクトリにはアプリケーションを格納しておかないことをお勧めします。プロダクション モードでは、WebLogic Server は `config/`*`mydomain`*`/applications` ディレクトリに新しく置かれたアプリケーションは検出しませんが、プロダクション モードに切り替わる前にこのディレクトリで検出していたアプリケーションは、引き続き自動的にデプロイします。

これらのデプロイメント手順の実行には、正しいディレクトリ構造を使用し、`web.xml` デプロイメント記述子を含み、必要であれば `weblogic.xml` デプロイメント記述子も含む、正常に作動する Web アプリケーションを作成していることを前提としています。Web アプリケーションは、アーカイブされた `.war` ファイルとして、または展開ディレクトリ形式でデプロイできます。詳細については、1-4 ページの「ディレクトリ構造」を参照してください。

Administration Console を使用した Web アプリケーションのデプロイ

Administration Console を使用して Web アプリケーションをデプロイするには、次の手順に従います。

1. WebLogic Server の管理サーバを起動します。
2. WebLogic Server Administration Console を起動します。詳細については、「WebLogic Server 管理の概要」を参照してください。
3. 左ペインの [デプロイメント] ノードを展開します。
4. [Web アプリケーション] ノードを右クリックします。
5. [新しい WebAppComponent のコンフィグレーション] を選択します。
6. [Path URI] フィールドに、展開ディレクトリへの絶対パスまたは `.war` ファイルへのパスを入力します。次に例を示します。

   （展開ディレクトリ形式）`c:\myApps\myWebApp`

   （アーカイブ形式）`c:\myApps\myWebApp.war`
7. [作成] をクリックします。
8. 左ペインの [Web アプリケーション] ノードから新しい Web アプリケーションを選択します。
9. 右ペインの [対象] タブを選択します。
10. [サーバ] タブを選択し、適切なサーバを [選択済み] ボックスに移動します。
11. WebLogic Server のクラスタを使用している場合、[クラスタ] タブを選択し、適切なクラスタを [選択済み] ボックスに移動します。
12. 仮想ホスティングを使用している場合、[仮想ホスト] タブを選択し、この Web アプリケーションに適用するすべての仮想ホストを対象にします。詳細については、「仮想ホスティングのコンフィグレーション」を参照してください。

13. [コンフィグレーション] タブを選択して、[一般]、[ファイル]、および [その他] タブで適用する属性をコンフィグレーションします。これらの属性の詳細情報を見るには、疑問符のアイコンをクリックします。
14. Web ブラウザからリソースをアクセスして、Web アプリケーションをテストします。次のように構築した URL からリソースをアクセスします。

    `http://myServer:myPort/myWebApp/resource`

    各要素の説明は次のとおりです。

    - *myServer* は、WebLogic Server のホスト マシンの名前。
    - *myPort* は、WebLogic Server がリクエストをリスンしているポート番号。
    - *myWebApp* は、Web アプリケーションのアーカイブ ファイルの名前（myWebApp.war など）、または Web アプリケーションを含むディレクトリの名前。
    - *resource* は、JSP、HTTP サーブレット、HTML ページなどのリソースの名前。

weblogic.deploy ユーティリティを使った Web アプリケーションのデプロイ

weblogic.deploy ユーティリティを使って Web アプリケーションをデプロイするには、次の手順に従います。

1. WebLogic Server クラスがシステムの CLASSPATH に入り、JDK が利用できるようにローカル環境を設定します。CLASSPATH の設定には、config*mydomain* ディレクトリにある setEnv スクリプトを使用できます。
2. 次のコマンドを入力します。

    ```
    % java weblogic.deploy -port port_number -host host_name
       -component application:target deploy password application
    source
    ```

    各要素の説明は次のとおりです。

    - *host_name* は、WebLogic Server のホスト マシンの名前。
    - *port_number* は、WebLogic Server がリクエストをリスンしているポート番号。

- *application* は、この Web アプリケーションに割り当てる名前。
- *target* は、この Web アプリケーションの対象となるサーバ、クラスタ、仮想ホストのいずれかの名前。対象はカンマで区切ると複数指定できます。
- *password* はシステム管理用パスワード。
- *source* はデプロイする .war ファイルのフル パス名、または展開ディレクトリ形式の Web アプリケーションを含むディレクトリのフル パス名。

次に例を示します。

```
java weblogic.deploy -port 7001 -host myhost -component
 myWebApp:myserver deploy pswd1234 myWebApp d:\myWebApp.war
```

プロダクション モードを有効にした Web アプリケーションの再デプロイ

プロダクション モードを有効にして Web アプリケーションを再デプロイするには、-DProductionModeEnabled=true フラグを指定して WebLogic 管理サーバを起動する必要があります。この機能は WebLogic 管理サーバでは使用できますが、管理対象サーバでは使用できません。詳細については、「WebLogic Server の起動と停止」を参照してください。

管理サーバ上の Web アプリケーションのコンポーネント（サーブレット、JSP、HTML ページなど）を変更するときは、対象になっている管理対象サーバに変更したコンポーネントもデプロイされるように、変更したコンポーネントを更新する追加の手順を実行する必要があります。コンポーネントを更新する 1 つの方法は、Web アプリケーション全体を再デプロイすることです。Web アプリケーションを再デプロイすることは、Web アプリケーションの対象になっている管理対象サーバすべてに、変更したコンポーネントだけではなく Web アプリケーション全体をネットワーク経由で再送信することを意味します。

Web アプリケーションの再デプロイメントでは、次の点に注意してください。

- 環境によっては、Web アプリケーションが管理対象サーバに送信されるときにネットワーク トラフィックが増加するために、パフォーマンスに影響が出る可能性があります。

- Web アプリケーションが現在プロダクション環境にあり、使用中である場合、この Web アプリケーションを再デプロイすることによって、WebLogic Server は、Web アプリケーションを使用中のユーザに対するすべてのアクティブな HTTP セッションを失うことになります。
- Java クラス ファイルを更新した場合、クラスを更新するために Web アプリケーション全体を再デプロイする必要があります。
- デプロイメント記述子を変更する場合、Web アプリケーションを再デプロイする必要があります。

JSP、HTML ファイル、画像ファイルなどの静的ファイルを更新する必要がある場合、`weblogic.deploy` ユーティリティの更新オプションを使って静的ファイルを更新できます。詳細については、2-14 ページの「アプリケーションの再デプロイを利用しない静的コンポーネントの更新」を参照してください。

Administration Console を使用した Web アプリケーションの再デプロイ

Administration Console で Web アプリケーションを再デプロイするには、次の手順に従います。

1. WebLogic Server Administration Console を起動します。
2. 左ペインの [デプロイメント] ノードを展開します。
3. [Web アプリケーション] ノードを展開します。
4. 再デプロイする Web アプリケーション ノードを選択します。
5. 右ペインの [デプロイ] ボックスの選択を解除します。
6. [適用] をクリックします。
7. 右ペインの [デプロイ] ボックスを選択します。
8. [適用] をクリックします。

注意 : Web アプリケーションを再デプロイすると、アクティブな HTTP セッションは失われます。

weblogic.deploy ユーティリティを使った Web アプリケーションの再デプロイ

次のコマンドを入力します。

```
% java weblogic.deploy -port port_number -host host_name
   update password application source
```

各要素の説明は次のとおりです。

- *host_name* は、WebLogic Server のホスト マシンの名前。
- *port_number* は、WebLogic Server がリクエストをリスンしているポート番号。
- *password* はシステム管理用パスワード。
- *application* は、Web アプリケーションの名前。
- *source* は再デプロイする .war ファイルのフル パス名、または展開ディレクトリ形式の Web アプリケーションを含むディレクトリのフル パス名。

注意： Web アプリケーションを再デプロイすると、アクティブな HTTP セッションは失われます。

アプリケーションの対象になっているサーバ インスタンスの 1 つでそのアプリケーションを更新すると、対象になっているすべてのサーバでアプリケーションが更新されます。たとえば、アプリケーションの対象がクラスタの場合、クラスタ化されたサーバ インスタンスの 1 つでアプリケーションを更新すると、アプリケーションはクラスタの全メンバで更新されます。同様に、クラスタとスタンドアロン サーバ インスタンスがアプリケーションの対象になっている場合は、スタンドアロン サーバのインスタンスでアプリケーションを更新すると、クラスタでもアプリケーションが更新されます。また、逆の場合も同様の処理が行われます。

部分的な再デプロイメント

アプリケーションの再デプロイを利用しない静的コンポーネントの更新

アプリケーション全体を再デプロイしないでアプリケーションの静的コンポーネントを更新するには、`weblogic.refresh` ツールを使用します。

管理サーバ上で Web アプリケーションの静的ファイルを更新する場合、ファイルは管理対象サーバ上の Web アプリケーションに自動的にはコピーされません。しかし、`weblogic.refresh` を使えば、次のような静的ファイルの更新、追加、または削除を行うことができます。

- JSP
- XML ファイル
- HTML ファイル
- `.gif` や `.jpg` などの画像ファイル
- テキスト ファイル

各 JSP はそれ自体のクラスローダによってロードされるので、JSP を更新するために Web アプリケーションを再デプロイする必要はありません。Web アプリケーションのクラスローダは、JSP 個別のクラスローダの親です。

このユーティリティには以下の制限があることに注意してください。

- このユーティリティは Java クラス ファイルの更新には使用できません。
- このユーティリティを使用するには、展開ディレクトリ形式で Web アプリケーションをデプロイする必要があります。また、Web アプリケーションが `.ear` ファイルに収められている場合は、`.ear` ファイルも展開形式でなければなりません。このユーティリティは、`.war` ファイルでアーカイブされたコンポーネントには機能しません。

静的ファイルを更新するには、次の手順に従います。

1. WebLogic Server クラスがシステムの CLASSPATH に入り、JDK が利用できるように開発環境を設定します。環境の設定には、config*mydomain* ディレクトリにある setEnv スクリプトを使用できます。

2. 次の形式でコマンドを入力します。

   ```
   % java weblogic.refresh -url -username -password -application
   -component -files -delete -root
   ```

   各要素の説明は次のとおりです。

   - *url* は、WebLogic 管理サーバの URL。
   - *username* は、システム管理用ユーザ名。
   - *password* はシステム管理用パスワード。
   - *application* は、更新する Web アプリケーションを含むエンタープライズ アプリケーションの名前。Web アプリケーションがエンタープライズ アプリケーションの一部ではない場合は、Web アプリケーションの名前を入力します。
   - *component* は、更新する Web アプリケーションの名前。
   - *files* は、更新されるファイルのカンマで区切ったリスト。*.jsp や *.gif などのワイルドカードを使って、ファイルを指定できます。ただし、複数のファイルを指定する場合、ワイルドカードとカンマ区切りの複数ファイル指定の両方を一緒に使うことはできないことに注意してください。ファイル名は、Web アプリケーションのルートを基準とする相対ファイル名でなければなりません。したがって、ファイルが ball.gif で、Web アプリケーションのルート ディレクトリの \foo サブディレクトリにある場合は、ファイル名は foo\ball.gif になります。サーバ上にファイルがない場合は、指定したサブディレクトリと共にファイルが作成されます。
   - *delete* を設定すると、指定したファイルが削除されます。
   - *root* は、更新用に使用するファイルが格納されているディレクトリ。デフォルトはカレント ディレクトリです。

   たとえば、次のコマンドは myWebApp Web アプリケーションの HelloWorld.jsp ファイルと foo\ball.gif ファイルを更新します。

   ```
   java weblogic.refresh -url t3://localhost:7001 -username
    myUsername -password myPassword  -application myApplication
    -component myWebApp -root c:\stagedir\myWebApp
    HelloWorld.jsp,foo\ball.gif
   ```

Web アプリケーションのアンデプロイ

Web アプリケーションをアンデプロイすると、コンフィグレーションには何ら影響はありませんが、Web アプリケーションはクライアントからのリクエストに応答できなくなります。Web アプリケーションは、Administration Console または `weblogic.deploy` ユーティリティを使用してアンデプロイできます。

Administration Console を使用した Web アプリケーションのアンデプロイ

Administration Console を使用して Web アプリケーションをアンデプロイするには、次の手順に従います。

1. Administration Console を起動します。
2. 左ペインの [デプロイメント] ノードを展開します。
3. [Web アプリケーション] ノードを展開します。
4. アンデプロイする [Web アプリケーション] ノードを選択します。
5. 右ペインの [コンフィグレーション] タブを選択します。
6. [デプロイ] ボックスの選択を解除します。
7. [適用] をクリックします。

weblogic.deploy ユーティリティを使った Web アプリケーションのアンデプロイ

`weblogic.deploy` ユーティリティを使って Web アプリケーションをアンデプロイするには、次の手順に従います。

1. WebLogic Server クラスがシステムの CLASSPATH に入り、JDK が利用できるように開発環境を設定します。CLASSPATH の設定には、config*mydomain* ディレクトリにある setEnv スクリプトを使用できます。
2. 次のコマンドを入力します。

   ```
   % java weblogic.deploy -port port_number -host host_name
      undeploy password application source
   ```

   各要素の説明は次のとおりです。

   - *host_name* は、WebLogic Server のホスト マシンの名前。
   - *port_number* は、WebLogic Server がリクエストをリスンしているポート番号。
   - *password* はシステム管理用パスワード。
   - *application* は、Web アプリケーションの名前。
   - *source* は再デプロイする .war ファイルのフル パス名、または展開ディレクトリ形式の Web アプリケーションを含むディレクトリのフル パス名。

Web アプリケーションの削除

Web アプリケーションの削除とは、ドメイン コンフィグレーションから Web アプリケーションのコンフィグレーション情報すべてを削除し、Web アプリケーションをクライアントからのリクエストに応答できなくすることです。Web アプリケーションは、Administration Console または weblogic.deploy ユーティリティを使用して削除できます。

Web アプリケーションを削除しても、ファイル システムから Web アプリケーションが削除されるわけではありません。

Web アプリケーションの applications ディレクトリからの削除

（WebLogic Server を `-DProductionModeEnabled=false` フラグで起動した）開発モードで WebLogic Server を稼働している場合、`applications` ディレクトリの Web アプリケーションを削除するには、`applications` ディレクトリから Web アプリケーションを含むディレクトリまたはアーカイブを物理的に削除する必要があります。

Administration Console を使用した Web アプリケーションの削除

Administration Console を使用して Web アプリケーションを削除するには、次の手順に従います。

1. WebLogic Server Administration Console を起動します。詳細については、「WebLogic Server 管理の概要」を参照してください。
2. 左ペインの [デプロイメント] ノードを展開します。
3. [Web アプリケーション] ノードを選択します。
4. 右ペインで、削除する Web アプリケーションの名前の隣にあるごみ箱アイコンをクリックします。本当に削除するかどうかを確認するプロンプトが表示されます。
5. [はい] をクリックします。

weblogic.deploy ユーティリティを使った Web アプリケーションの削除

`weblogic.deploy` ユーティリティを使って Web アプリケーションを削除するには、次の手順に従います。

1. WebLogic Server クラスがシステムの CLASSPATH に入り、JDK が利用できるように開発環境を設定します。CLASSPATH の設定には、config*mydomain* ディレクトリにある setEnv スクリプトを使用できます。
2. 次のコマンドを入力します。

   ```
   % java weblogic.deploy -port port_number -host host_name
      delete password application
   ```

   各要素の説明は次のとおりです。

   - *host_name* は、WebLogic Server のホスト マシンの名前。
   - *port_number* は、WebLogic Server がリクエストをリスンしているポート番号。
   - *password* はシステム管理用パスワード。
   - *application* は、Web アプリケーションの名前。

エンタープライズ アプリケーションの一部としての Web アプリケーションのデプロイ

エンタープライズ アプリケーションは、Web アプリケーション、EJB、およびリソース アダプタを単一のデプロイ可能なユニットにバンドルする J2EE デプロイメント ユニットです。詳細については、「WebLogic Server J2EE アプリケーションのパッケージ化」を参照してください。エンタープライズ アプリケーションの一部として Web アプリケーションをデプロイすると、WebLogic Server が Web アプリケーション用のリクエストを解決するときに、Web アプリケーションの実際の名前の代わりに使う文字列を指定することができます。エンタープライズ アプリケーション用の application.xml デプロイメント記述子の <context-root> 要素に新しい名前を指定します。詳細については、「application.xml デプロイメント記述子の要素」を参照してください。

たとえば、oranges という Web アプリケーションでは、多くの場合、次のような URL で oranges Web アプリケーションのリソースを要求します。

```
http://host:port/oranges/catalog.jsp
```

oranges Web アプリケーションがエンタープライズ アプリケーションにパッケージ化された場合、次の例で示す <context-root> に値を指定できます。

```
<module>
   <web>
      <web-uri>oranges.war</web-uri>
      <context-root>fruit</context-root>
   </web>
</module>
```

その結果、次の URL を使用して、oranges Web アプリケーションの同じリソースにアクセスできます。

```
http://host:port/fruit/catalog.jsp
```

注意： 1 つのエンタープライズ アプリケーションでは、複数の名前で 1 つの Web アプリケーションをデプロイすることはできません。ただし、それぞれの Web アプリケーションが異なるエンタープライズ アプリケーションにパッケージ化されている場合には、複数の名前で同一の Web アプリケーションをデプロイできます。

スタンドアロン Web アプリケーションのデプロイ

Web アプリケーションがスタンドアロンである場合は、weblogic.xml ファイルの context-root 要素を使って、Web アプリケーションのコンテキスト ルートを指定できます。Web アプリケーションが EAR の一部である場合は、EAR の application.xml ファイルでコンテキスト ルートを指定します。application.xml ファイルにおける context-root の設定の方が、weblogic.xml での context-root の設定より優先されます。context-root 要素を参照してください。

3 Web アプリケーション コンポーネントのコンフィグレーション

次の節では、Web アプリケーションをコンフィグレーションする方法について説明します。

サーブレットのコンフィグレーション

サーブレットは、Web アプリケーションの一部として登録およびコンフィグレーションされます。サーブレットを登録するには、Web アプリケーション デプロイメント記述子にいくつかのエントリを追加します。<servlet> 要素の下の最初のエントリには、サーブレットの名前が定義され、そのサーブレットを実行するコンパイル済みクラスが指定されます（あるいは、サーブレット クラスを指定する代わりに、JSP ページを指定することもできます）。この要素には、サーブレットの初期化パラメータとセキュリティ ロール用の定義も含まれています。<servlet-mapping> 要素の下の 2 番目のエントリには、このサーブレットを呼び出す URL パターンが定義されています。

Web アプリケーション デプロイメント記述子 の詳しい編集手順については、以下のトピックを参照してください。

- 8-10 ページの「手順 9 : サーブレットのデプロイ」
- 8-13 ページの「手順 10:URL へのサーブレットのマッピング」

サーブレット マッピング

サーブレット マッピングとは、サーブレットへのアクセス方法を制御することです。次の例は、Web アプリケーションでサーブレット マッピングを使用する方法を示しています。この例には、（web.xml デプロイメント記述子の）サーブレット コンフィグレーションとマッピングがあり、その後にこれらのサーブレットの起動に使う URL を示す表（3-3 ページの「url-pattern と呼び出されるサーブレット」を参照）があります。

コード リスト 3-1　サーブレット マッピングの例

```
<servlet>
  <servlet-name>watermelon</servlet-name>
  <servlet-class>myservlets.watermelon</servlet-class>
</servlet>

<servlet>
  <servlet-name>garden</servlet-name>
```

```
  <servlet-class>myservlets.garden</servlet-class>
</servlet>

<servlet>
  <servlet-name>list</servlet-name>
  <servlet-class>myservlets.list</servlet-class>
</servlet>

<servlet>
  <servlet-name>kiwi</servlet-name>
  <servlet-class>myservlets.kiwi</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>watermelon</servlet-name>
  <url-pattern>/fruit/summer/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>garden</servlet-name>
  <url-pattern>/seeds/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>list</servlet-name>
  <url-pattern>/seedlist</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>kiwi</servlet-name>
  <url-pattern>*.abc</url-pattern>
</servlet-mapping>
```

表 3-1 url-pattern と呼び出されるサーブレット

URL	呼び出されるサーブレット
http://host:port/mywebapp/fruit/summer/index.html	watermelon
http://host:port/mywebapp/fruit/summer/index.abc	watermelon
http://host:port/mywebapp/seedlist	list

表 3-1 url-pattern と呼び出されるサーブレット（続き）

URL	呼び出されるサーブレット
http://host:port/mywebapp/seedlist/index.html	デフォルト サーバ（コンフィグレーションされている場合）、または HTTP 404 エラー メッセージ（「File not found」）。 list サーブレットが /seedlist* にマップされていた場合、list サーブレットが呼び出される。
http://host:port/mywebapp/seedlist/pear.abc	kiwi list サーブレットが /seedlist* にマップされていた場合、list サーブレットが呼び出される。
http://host:port/mywebapp/seeds	garden
http://host:port/mywebapp/seeds/index.html	garden
http://host:port/mywebapp/index.abc	kiwi

サーブレット初期化パラメータ

サーブレットの初期化パラメータは、Web アプリケーション デプロイメント記述子の中の <servlet> 要素の <init-param> 要素に、<param-name> タグと <param-value> タグを使って定義します。次に例を示します。

コード リスト 3-2　サーブレット初期化パラメータのコンフィグレーション例

```
<servlet>
  <servlet-name>HelloWorld2</servlet-name>
  <servlet-class>examples.servlets.HelloWorld2</servlet-class>

  <init-param>
    <param-name>greeting</param-name>
    <param-value>Welcome</param-value>
  </init-param>

  <init-param>
    <param-name>person</param-name>
    <param-value>WebLogic Developer</param-value>
  </init-param>
</servlet>
```

Web アプリケーション デプロイメント記述子の編集の詳細については、8-1 ページの「Web アプリケーションのデプロイメント記述子の記述」を参照してください。

JSP のコンフィグレーション

JavaServer Pages（JSP）ファイルは、Web アプリケーションのルート（またはルートの下のサブディレクトリ）に格納することでデプロイされます。追加の JSP コンフィグレーション パラメータは WebLogic 固有のデプロイメント記述子である weblogic.xml の <jsp-descriptor> 要素に定義されます。これらのパラメータは次の機能を定義します。

- JSP パラメータのオプション
- デバッグ

- 再コンパイルが必要になる更新した JSP を WebLogic Server がチェックする頻度
- 文字エンコーディング

これらのパラメータの詳細については、B-12 ページの「JSP パラメータの名前と値」を参照してください。

`weblogic.xml` ファイルの編集手順については、8-24 ページの「weblogic.xml ファイル作成の主な手順」を参照してください。

`<servlet>` タグを使うと JSP をサーブレットとして登録することもできます。次の例で、`/main` を含む URL は `myJSPfile.jsp` を起動します。

```
<servlet>
     <servlet-name>myFoo</servlet-name>
     <jsp-file>myJSPfile.jsp</jsp-file>
</servlet>

<servlet-mapping>
     <servlet-name>myFoo</servlet-name>
     <url-pattern>/main</url-pattern>
</servlet-mapping>
```

この方法で JSP を登録することによって、サーブレットに対して行うのと同様に、ロード順、初期化パラメータ、およびセキュリティ ロールを JSP に指定できます。

JSP タグ ライブラリのコンフィグレーション

Weblogic Server には、カスタム JSP タグを作成し、使用する機能が用意されています。カスタム JSP タグは、JSP ページから呼び出すことができる Java クラスです。カスタム JSP タグを作成するには、それらをタグ ライブラリに登録して、それらの動作をタグ ライブラリ記述子（TLD）ファイルに定義します。この TLD は、JSP が組み込まれている Web アプリケーションで使用できなければなりません。そのためには、Web アプリケーション デプロイメント記述子にその TLD を定義します。TLD ファイルは、Web アプリケーションの `WEB-INF` ディレクトリに格納してください。このディレクトリは、外部には公開されません。

Web アプリケーション デプロイメント記述子には、タグ ライブラリの URI パターンを定義します。この URI パターンは、JSP ページの taglib ディレクティブの値と一致する必要があります。また、TLD の格納場所も定義します。たとえば、JSP ページに次の taglib ディレクティブがあるとします。

```
<%@ taglib uri="myTaglib" prefix="taglib" %>
```

また、TLD が Web アプリケーションの WEB-INF ディレクトリに格納されているとします。この場合、Web アプリケーション デプロイメント記述子に次のエントリを作成します。

```
<taglib>
   <taglib-uri>myTaglib</taglib-uri>
   <tablig-location>WEB-INF/myTLD.tld</taglib-location>
</taglib>
```

タグ ライブラリは、.jar ファイルとしてデプロイできます。詳細については、「JSP タグ ライブラリを JAR ファイルとしてデプロイする」を参照してください。

カスタム JSP タグ ライブラリの作成の詳細については、『JSP Tag Extensions プログラマーズ ガイド』を参照してください。

WebLogic Server には、アプリケーションで使用できるカスタム JSP タグがいくつか付属しています。これらのタグを使用すると、キャッシング、クエリ パラメータベースのフロー制御の効率化、およびオブジェクト セットに対する反復処理の効率化を行うことができます。詳細については、以下を参照してください。

- 「カスタム WebLogic JSP タグの使い方」
- 「WebLogic JSP フォーム検証タグの使い方」

ウェルカム ページのコンフィグレーション

WebLogic Server では、要求された URL がディレクトリである場合にデフォルトによって提供されるページを設定できます。ユーザが特定のファイル名を指定せずに URL を入力できるので、このウェルカム ページの機能でサイトが使いやすくなります。

ウェルカム ページは、Web アプリケーション レベルで定義します。サーバが複数の Web アプリケーションのホストになっている場合は、Web アプリケーションごとに別個のウェルカム ページを定義する必要があります。

ウェルカム ページを定義するには、Web アプリケーション デプロイメント記述子の `web.xml` を編集します。詳細については、8-15 ページの「手順 13 : ウェルカム ページの定義」を参照してください。

ウェルカム ページを定義していない場合、WebLogic Server は以下のファイルを次の順序で検索し、最初に見つけたものにサービスを提供します。

1. `index.html`
2. `index.htm`
3. `index.jsp`

詳細については、「WebLogic Server による HTTP リクエストの解決方法」を参照してください。

デフォルト サーブレットの設定

各 Web アプリケーションには、デフォルト サーブレットがあります。このデフォルト サーブレットは管理者が指定できますが、指定しない場合、WebLogic Server では `FileServlet` という内部サーブレットがデフォルト サーブレットとして使用されます。`FileServlet` の詳細については、「WebLogic Server による HTTP リクエストの解決方法」を参照してください。

どのサーブレットでも、デフォルト サーブレットとして登録できます。独自のデフォルト サーブレットを作成すれば、独自のロジックを使用して、デフォルト サーブレットに送られるリクエストの処理方法を定義できます。

設定したデフォルト サーブレットは、`FileServlet` に取って代わります。`FileServlet` はほとんどのファイル（テキスト ファイル、HTML ファイル、イメージ ファイルなど）を提供するために使用されるので、デフォルト サーブレットの設定は注意深く行う必要があります。デフォルト サーブレットでこれらのファイルを提供するには、その機能をデフォルト サーブレットに記述する必要があります。

ユーザ定義のデフォルト サーブレットを設定するには、次の手順を実行します。

1. 3-2 ページの「サーブレットのコンフィグレーション」の説明に従って、サーブレットを定義します。
2. デフォルト サーブレットを、「/」という url パターンを使ってマップします。これにより、デフォルト サーブレットは、拡張子が *.htm または *.html のファイルを除くすべてのファイルに応答します。

   デフォルト サーバが *.htm または *.html ファイルに応答するようにするには、これらの拡張子をデフォルト サーブレットにマップし、さらに「/」をマップします。サーブレットのマップの手順については、3-2 ページの「サーブレットのコンフィグレーション」を参照してください。
3. FileServlet を他の拡張子付きのファイルに提供する場合は、次の手順に従います。
   a. サーブレットを定義し、myFileServlet などの <servlet-name> を指定します。
   b. <servlet-class> を weblogic.servlet.FileServlet と定義します。
   c. <servlet-mapping> 要素を使ってファイル拡張子を myFileServlet にマップします（デフォルト サーブレット用のマッピングに追加して）。たとえば、myFileServlet で gif ファイルを提供するには、*.gif を myFileServlet にマップします。

HTTP エラー応答のカスタマイズ

WebLogic Server をコンフィグレーションすると、特定の HTTP エラーまたは Java 例外が発生したときに、標準の WebLogic Server エラー応答ページを使う代わりに、カスタム Web ページなどの HTTP リソースを使って応答させることができます。

カスタム エラー ページは、Web アプリケーション デプロイメント記述子（`web.xml`）の `<error-page>` 要素に定義します。詳細については、A-17 ページの「error-page 要素」を参照してください。

WebLogic Server での CGI の使用

WebLogic Server には、レガシー CGI（Common Gateway Interface）スクリプトのサポート機能が用意されています。しかし、新しいプロジェクトでは、HTTP サーブレットまたは JavaServer Pages を使用することをお勧めします。

WebLogic Server は、`CGIServlet` という内部 WebLogic サーブレットを介してすべての CGI スクリプトをサポートします。CGI を使用するには、この `CGIServlet` を Web アプリケーション デプロイメント記述子に登録します（3-12 ページの「CGIServlet の登録に使用する Web アプリケーション デプロイメント記述子エントリのサンプル」を参照）。詳細については、3-2 ページの「サーブレットのコンフィグレーション」を参照してください。

CGI を使用するための WebLogic Server のコンフィグレーション

WebLogic Server で CGI をコンフィグレーションするには、次の手順に従います。

1. <servlet> 要素および <servlet-mapping> 要素を使用して、Web アプリケーションで CGIServlet を宣言します。CGIServlet のクラス名は、weblogic.servlet.CGIServlet です。このクラスを Web アプリケーションにパッケージ化する必要はありません。
2. 以下の <init-param> 要素を定義して、CGIServlet の初期化パラメータを登録します。

   cgiDir

   CGI スクリプトが存在するディレクトリのパス。複数のディレクトリを指定するには、セミコロン「;」(Windows) またはコロン「:」(UNIX) で区切ります。CGI ディレクトリをアプリケーション ルートを基準にして指定する場合、または CGI ディレクトリが WAR ファイルの内部に存在する場合は、cgiDir を指定します。cgiDir については、以下の点に注意してください。

   WebLogic Server が CGI スクリプトを取り出すことができるよう、CGI スクリプトが存在するすべてのディレクトリを個別に指定する必要があります。

   CGIServlet は、cgiDir で指定されているパスのサブディレクトリからは、CGI を取り出しません。

   取り出された CGI スクリプトはすべて同じディレクトリに格納され、CGI スクリプトの検索にはスクリプト名だけが使用されて完全パスや相対パスは使用されないので、サブディレクトリの場所に関係なく、CGI スクリプトの名前はすべてユニークにする必要があります。

   たとえば、/foo/myscript.pl と /bar/myscript.pl は混同され、どちらが呼び出されるかわかりません。

   extension mapping

   スクリプトを実行するインタープリタまたは実行可能ファイルにファイル拡張子をマップします。スクリプトが実行可能ファイルを必要としない場合、この初期化パラメータは省略可能です。

拡張子マッピング用の <param-name> は、*.pl のように、アスタリスク、ドット、ファイル拡張子の順で指定する必要があります。

<param-value> には、スクリプトを実行するインタープリタまたは実行可能ファイルへのパスが含まれます。マッピングごとに個別の <init-param> 要素を作成すると、複数のマッピングを作成できます。

コード リスト 3-3 CGIServlet の登録に使用する Web アプリケーション デプロイメント記述子エントリのサンプル

```
<servlet>
 <servlet-name>CGIServlet</servlet-name>
 <servlet-class>weblogic.servlet.CGIServlet</servlet-class>
 <init-param>
  <param-name>cgiDir</param-name>
  <param-value>
   /bea/wlserver6.0/config/mydomain/applications/myWebApp/cgi-bin
  </param-value>
 </init-param>

  <init-param>
   <param-name>*.pl</param-name>
   <param-value>/bin/perl.exe</param-value>
  </init-param>
</servlet>

...

<servlet-mapping>
   <servlet-name>CGIServlet</servlet-name>
   <url-pattern>/cgi-bin/*</url-pattern>
</servlet-mapping>
```

CGI スクリプトの要求

perl スクリプトを要求するために使用する URL は、次のパターンに従う必要があります。

http://*host:port/myWebApp/cgi-bin/myscript.pl*

各要素の説明は次のとおりです。

host:port
: WebLogic Server のホスト名とポート番号

myWebApp
: Web アプリケーションの名前

cgi-bin
: CGIServlet にマップされる url-pattern 名

myscript.pl
: cgiDir 初期化パラメータで指定したディレクトリに存在する Perl スクリプトの名前

ClasspathServlet による CLASSPATH からのリソースの提供

システム CLASSPATH から、または Web アプリケーションの WEB-INF/classes ディレクトリからクラスまたは他のリソースを提供する必要がある場合、ClasspathServlet と呼ばれる特殊なサーブレットを使用できます。ClasspathServlet はアプレットまたは RMI クライアントを使用し、サーバサイド クラスへのアクセスを必要とするアプリケーションで役に立ちます。ClasspathServlet は暗黙的に登録され、任意のアプリケーションから利用できます。

ClasspathServlet を呼び出す一般的な形式は、次のとおりです。

```
http://server:port[/<context-path>]/classes[/<app>@[<webapp.]]/
<package>.<classname>>.class
```

各要素の内容は以下のとおりです。

context-path
: ClasspathServlet を呼び出す対象のアプリケーションを指定します。

app
: 提供するクラス（またはクラスを含む Web アプリケーション）を含むアプリケーションの名前です。

webapp
: クラスを含む Web アプリケーションの名前です。

ClasspathServlet を使用するには次の 2 つの方法があります。

- システム CLASSPATH からリソースを提供する場合、次のような URL でリソースを呼び出す。

  ```
  http://server:port/classes/weblogic/myClass.class
  ```

- Web アプリケーションの WEB-INF/classes ディレクトリからリソースを提供する場合、次のような URL でリソースを呼び出す。

  ```
  http://server:port/classes/myApp@myWebApp/examples/servlets/myClass.class
  ```

  この場合、リソースは Web アプリケーションのルートに相対した次のディレクトリにあります。

  ```
  WEB-INF/classes/my/resource/myClass.class
  ```

警告： ClasspathServlet はシステム CLASSPATH にあるすべてのリソースを提供するので、システム CLASSPATH には公開できないリソースは置かないでください。

Web アプリケーションでの外部リソースのコンフィグレーション

Web アプリケーションから JNDI を介して DataSource などの外部リソースにアクセスする場合、コード内でルックアップする JNDI 名を、JNDI ツリーにバインドされているとおりの実際の JNDI 名にマップできます。このマッピングを行うには、web.xml と weblogic.xml の両方のデプロイメント記述子を使用します。このマッピングにより、アプリケーション コードを変更せずにこれらのリソースを変更できるようになります。デプロイメント記述子には、Java コードで使用される名前、JNDI ツリーにバインドされているとおりのリソースの名前、リソースの Java タイプを指定します。また、リソースのセキュリティをサーブレットによってプログラマティックに処理するか、または HTTP リクエストに関連付けられる資格に基づいて処理するかを指定します。

外部リソースをコンフィグレーションするには、次の手順に従います。

1. コードで使用するリソース名、Java タイプ、およびセキュリティ認証タイプをデプロイメント記述子に入力します。デプロイメント記述子エントリの作成手順については、8-17 ページの「手順 16 : 外部リソースの参照」を参照してください。
2. リソース名を JNDI 名にマップします。デプロイメント記述子エントリの作成手順については、8-26 ページの「手順 3 : リソースの JNDI へのマッピング」を参照してください。

この例は、accountDataSource というデータ ソースが定義されていることを前提としています。詳細については、「JDBC データ ソース」を参照してください。

コード リスト 3-4　DataSource の使用例

```
Servlet code:
javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup
                          ("myDataSource");

web.xml のエントリ :

<resource-ref>
. . .
   <res-ref-name>myDataSource</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>CONTAINER</res-auth>
. . .
</resource-ref>

weblogic.xml エントリ

<resource-description>
   <res-ref-name>myDataSource</res-ref-name>
   <jndi-name>accountDataSource</jndi-name>
</resource-description>
```

Web アプリケーションでの EJB の参照

weblogic-ejb-jar.xml ファイル デプロイメント記述子に定義される EJB の JNDI 名にマップされる名前を Web アプリケーション デプロイメント記述子に指定すると、Web アプリケーションで EJB を参照できます。この手順は、Web アプリケーションと EJB との間接レベルを提供しており、サード パーティの

EJB や Web アプリケーションを使用していて、EJB を直接呼び出すコードを変更できない場合に便利です。ほとんどの場合、この間接機能を使用しなくても EJB は直接呼び出すことができます。詳細については、「デプロイされた EJB の呼び出し」を参照してください。

Web アプリケーションで使用するために EJB を参照するには、次の手順に従います。

1. コード内で EJB をルックアップするために使用する EJB 参照名、Java クラス名、EJB のホームおよびリモート インタフェース名を、Web アプリケーション デプロイメント記述子の `<ejb-ref>` 要素に入力します。デプロイメント記述子エントリの作成手順については、8-21 ページの「手順 21 : エンタープライズ JavaBean（EJB）リソースの参照」を参照してください。
2. WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<ejb-reference-description>` 要素にある参照名を `weblogic-ejb-jar.xml` ファイルに定義された JNDI 名にマップします。デプロイメント記述子エントリの作成手順については、8-26 ページの「手順 3 : リソースの JNDI へのマッピング」を参照してください。

   Web アプリケーションがエンタープライズ アプリケーション アーカイブ（`.ear` ファイル）の一部である場合、`.ear` の `<ejb-link>` 要素で使用されている名前によって EJB を参照できます。

HTTP リクエストのエンコーディングの識別

WebLogic Server は、HTTP リクエストに含まれている文字データを、そのネイティブのエンコーディングから Java サーブレット API が使用する Unicode エンコーディングに変換する必要があります。この変換を実行するために、WebLogic Server は、リクエストのエンコーディングでどのコードセットが使われたのかを知る必要があります。

コードセットを定義する方法は 2 種類あります。

- POST 処理では、HTML <form> タグでエンコーディングを設定できます。たとえば、次の form タグはコンテンツの文字セットに SJIS を設定しています。

  ```
  <form action="http://some.host.com/myWebApp/foo/index.html">
    <input type="application/x-www-form-urlencoded; charset=SJIS">
  </form>
  ```

  フォームは、WebLogic Server に読み込まれると、SJIS 文字セットを使ってデータを処理します。

- 前述の例で、セミコロンの後ろにある情報はすべての Web クライアントが転送するわけではないので、WebLogic 固有のデプロイメント記述子である weblogic.xml にある <input-charset> 要素を使って、リクエストに使用するコードセットを設定できます。<java-charset-name> 要素は、リクエストの URL が <resource-path> 要素で指定したパスを含んでいるときにデータを変換するエンコーディングを定義します。

  次に例を示します。

  ```
  <input-charset>
    <resource-path>/foo/*</resource-path>
    <java-charset-name>SJIS</java-charset-name>
  </input-charset>
  ```

  この方法は GET 処理と POST 処理の両方で使用できます。

  Web アプリケーション デプロイメント記述子の詳細については、8-23 ページの「WebLogic 固有のデプロイメント記述子（weblogic.xml）の記述」を参照してください。

IANA 文字セットの Java 文字セットへのマッピング

文字セットを記述するために Internet Assigned Numbers Authority（IANA）によって割り当てられる名前は、Java で使用される名前とは異なることがあります。すべての HTTP 通信が IANA 文字セット名を使用し、これらの名前は常に同じとは限らないので、WebLogic Server は IANA 文字セット名を Java 文字セッ

ト名に内部でマップし、通常は正しいマッピングを判断できます。ただし、IANA 文字セットを Java 文字セットの名前に明示的にマッピングすると、あいまいさを解決することができます。

IANA 文字セットを Java 文字セットにマップするために、文字セットは WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<charset-mapping>` 要素で名前を付けられます。`<iana-charset-name>` 要素に IANA 文字セット名を定義し、`<java-charset-name>` 要素に Java 文字セット名を定義します。次に例を示します。

```
<charset-mapping>
    <iana-charset-name>Shift-JIS</iana-charset-name>
    <java-charset-name>SJIS</java-charset-name>
</charset-mapping>
```

4 Web アプリケーションにおけるセッションとセッション永続性の使用

この章では、セッションとセッションの永続性を設定する方法について説明します。

HTTP セッションの概要

セッション トラッキングを使用すると、複数のサーブレットか HTML ページにわたって、本来はステートレスであるユーザの状況を追跡できます。セッションの定義は、ある一定期間中に同じクライアントから出される一連の関連性のあるブラウザ リクエストです。セッション トラッキングは、ショッピング カート アプリケーションのように、全体として何らかの意味を持つ一連のブラウザ リクエスト（これらのリクエストをページとみなす）を結合します。

セッション管理の設定

WebLogic Server は、デフォルトによってセッション トラッキングを処理するよう設定されています。セッション トラッキングを使用する際にプロパティを設定する必要はありません。ただし、WebLogic Server がどのようにセッションを

管理するかをコンフィグレーションすることは、最高のパフォーマンスを実現するためにアプリケーションをチューニングするときの重要な鍵となります。チューニングの内容は、以下のような要素によって異なります。

- サーブレットをヒットする予定ユーザ数
- サーブレットをヒットする同時ユーザ数
- 各セッションの継続時間
- 各ユーザに対する予定格納データ量
- WebLogic Server インスタンスに割り当てられるヒープ サイズ

HTTP セッション プロパティ

WebLogic Server のセッション トラッキングは、WebLogic 固有のデプロイメント記述子である `weblogic.xml` のプロパティでコンフィグレーションします。WebLogic 固有のデプロイメント記述子の編集手順については、8-28 ページの「手順 4 : セッション パラメータの定義」を参照してください。

セッション属性の全リストについては、B-4 ページの「session-descriptor 要素」を参照してください。

セッション タイムアウト

HTTP セッションが期限切れになるまでの間隔を指定できます。セッションが期限切れになると、そのセッションに格納されたデータは破棄されます。間隔は、以下の 2 通りの方法で設定できます。

- WebLogic 固有のデプロイメント記述子である `weblogic.xml` の B-4 ページの「session-descriptor 要素」の `TimeoutSecs` 属性を設定します。この値は秒単位で設定します。
- Web アプリケーション デプロイメント記述子である `web.xml` の `<session-timeout>`（A-14 ページの「session-config 要素」を参照）要素を設定します。

セッション クッキーのコンフィグレーション

WebLogic Server は、クライアント ブラウザによってサポートされる場合、クッキーを使用してセッション管理を行います。

WebLogic Server がセッション トラッキングに使用するクッキーは、デフォルトによって一時的なものとして設定されており、ブラウザの実行期間より長く存続することはありません。ユーザがブラウザを終了すると、クッキーは失われ、セッション有効期間が終了したものと見なされます。この動作はセッションの用途の基本であり、このようにセッションを使用することをお勧めします。

WebLogic 固有のデプロイメント記述子である `weblogic.xml` で定義される属性によって、セッションのトラッキングに使われるクッキーのさまざまな側面をコンフィグレーションできます。セッションの全リストとクッキーに関連する属性については、B-4 ページの「session-descriptor 要素」を参照してください。

WebLogic 固有のデプロイメント記述子の編集手順については、8-28 ページの「手順 4 : セッション パラメータの定義」を参照してください。

存続期間が長いクッキーの使い方

長期間存続するクライアントサイド ユーザ データの場合、WebLogic Server アプリケーションは HTTP サーブレット API を介してブラウザに独自のクッキーを作成および設定し、HTTP セッションに関連付けられたクッキーは使用しようとはしません。WebLogic Server アプリケーションがクッキーを使用して特定のマシンのユーザの自動ログインを行う場合、新しいクッキーの存続期間を長く設定します。クッキーは、クライアント マシンだけから送ることができます。WebLogic Server アプリケーションは、そのユーザが複数の場所からアクセスする必要がある場合、サーバにデータを格納する必要があります。

ブラウザ クッキーの存続期間を直接セッションの長さに関連付けることはできません。クッキーがそれに関連付けられているセッションより早く期限切れになった場合、そのセッションは切り離されてしまいます。セッションががそれに関連付けられているクッキーより早く期限切れになった場合、サーブレットはそのセッションを見つけることができません。この場合、新しいセッションは `request.getSession(true)` メソッドが呼び出されるときに割り当てられます。セッションの使用は一時的なものに限定する必要があります。

クッキーの最大存続時間は、weblogic.xml デプロイメント記述子のセッション記述子にある CookieMaxAgeSecs パラメータで設定できます。詳細については、8-28 ページの「手順 4 : セッション パラメータの定義」を参照してください。

セッションのログアウトと終了

ユーザ認証の情報は、ユーザのセッション データ、およびサーバのコンテキストまたは Web アプリケーションにより割り当てられた仮想ホストのコンテキストの両方に格納されます。ユーザのログアウトに多く使われる session.invalidate() メソッドを使用すると、ユーザの現在のセッションのみが無効になり、ユーザの認証情報は有効なまま、サーバまたは仮想ホストのコンテキストに格納されます。サーバまたは仮想ホストが 1 つの Web アプリケーションだけをホストしている場合は、session.invalidate() メソッドを実行するとユーザはログアウトされます。

複数の Web アプリケーションで認証を使用するときには、複数の Java メソッドと戦略を使用できます。詳細については、『HTTP サーブレット プログラマーズ ガイド』「複数のアプリケーションに対する単一のサインオンの実装」を参照してください。

セッションの永続性のコンフィグレーション

セッションの永続性によって、HTTP セッション オブジェクトに格納されたデータは永続的に格納され、WebLogic Server のクラスタ全体のフェイルオーバとロード バランシングを有効にします。

セッションの永続性の実装は、以下の 5 つです。

- メモリ（単一サーバ、非レプリケート）
- ファイル システムの永続性
- JDBC の永続性
- クッキーベースの永続性

- インメモリ レプリケーション（クラスタ全体）

最初の 4 つについてはここで説明します。インメモリ レプリケーションについては、「HTTP セッション ステートのレプリケーションについて」を参照してください。

ファイル、JDBC、クッキーベース、およびインメモリ レプリケーションの場合、`PersistentStoreType` などの追加属性を設定する必要があります。各メソッドは、次に示すように独自の属性セットを持ちます。

アプリケーションが HTTP セッション オブジェクトにデータを格納するとき、データをシリアライズ可能にする必要があります。

共通プロパティ

メモリ内に保持されるセッションの数は、WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<session-descriptor>` 要素で次のプロパティを設定してコンフィグレーションできます。これらのプロパティは、セッション永続性を使用している場合にだけ適用できます。

`CacheSize`

メモリ内で一度にアクティブにできるキャッシュされたセッションの数を制限します。同時に大量のアクティブ セッションが発生することが見込まれる場合、これらのセッションでサーバの RAM を満たしたくはありません。仮想メモリとのスワッピングにより、パフォーマンスが低下する可能性があるからです。キャッシュが満杯になると、最も古いセッションは永続ストレージに格納され、必要になったときに自動的に呼び戻されます。永続性を使用しない場合、このプロパティは無視され、メイン メモリに保持可能なセッション数はソフトウェアによって制限されなくなります。デフォルトでは、キャッシュされるセッションの数は 1024 です。最小値は 16、最大値は *`Integer.MAX_VALUE`* です。空のセッションは 100 バイト未満のメモリしか使用しませんが、データの追加に応じて大きくなります。

`SwapIntervalSecs`

`cacheEntries` の制限に達したときに、WebLogic Server が最も古いセッションをキャッシュから永続ストレージにパージするまでの待ち時間です。

このプロパティを設定しない場合、デフォルトは 10 秒です。最小値は 1 秒、最大値は 604800 秒（1 週間) です。

`InvalidationIntervalSecs`

WebLogic Server が、タイムアウトの無効なセッションに対してハウスクリーニング チェックを実行してから古いセッションを削除してメモリを解放するまでの待ち時間を秒単位で設定します。このパラメータは、`<session-timeout>` 要素の設定値より小さく設定します。このパラメータを使用すると、トラフィックの多いサイトで WebLogic Server の動作を最適化できます。

最小値は毎秒（1）です。最大値は、週に 1 回 (604800 秒) です。このパラメータを設定しない場合、デフォルトは 60 秒です。

Web アプリケーション デプロイメント記述子である `web.xml` の A-14 ページの「session-config 要素」に `<session-timeout>` を設定します。

メモリ ベース、単一サーバ、非レプリケート永続ストレージの使い方

メモリ ベース、単一サーバ、非レプリケート永続ストレージを使用するには、WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<session-descriptor>` 要素にある `PersistentStoreType` プロパティを `memory` に設定します。メモリ ベースのストレージを使用する場合、すべてのセッション情報はメモリに格納され、WebLogic Server を終了して再起動すると失われます。

注意： WebLogic Server を実行するときに十分なヒープ サイズを割り当てないと、負荷がかかったときにサーバのメモリが足りなくなることがあります。

ファイルベースの永続ストレージの使い方

ファイルベースの永続ストレージをセッション用に使用するには、次の手順に従います。

1. WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<session-descriptor>` 要素の `PersistentStoreType` プロパティを `file` に設定します。
2. WebLogic Server がセッションを格納するディレクトリを設定します。このディレクトリの設定の詳細については、B-8 ページの「`PersistentStoreDir`」を参照してください。

   この属性値を明示的に設定しなかった場合、WebLogic Server によって一時ディレクトリが自動的に作成されます。

クラスタ環境 (WebLogic Cluster の、または WebLogic Cluster ではない複数の WebLogic インスタンスの) でファイルベースの永続ストレージを使用するには、以下の点に注意します。

- 上記の属性を、クラスタ内のすべてのサーバがアクセスできる共有ディレクトリに明示的に設定しなければなりません。このディレクトリは手動で作成する必要があります。
- 維持型のセッションであり続けるためには、フロントエンドのプロキシ サーバも必要です。与えられたクライアントに対して、同じサーバでリクエストが処理されなければなりません。そうでないと、複数のサーバが同じストレージに書き込もうとしたときに問題が発生します。
- `weblogic.xml` で、`InvalidationIntervalSecs` 属性を非常に大きい値に設定する必要もあります。

データベースの永続ストレージとしての使い方（JDBC 永続性）

JDBC の永続性は、この目的のために提供されたスキーマを使ってデータベース テーブルにセッション データを格納します。JDBC ドライバを備えたデータベースはすべて使用できます。データベース アクセスは、接続プールを使ってコンフィグレーションします。

JDBC ベースの永続ストレージをセッション用にコンフィグレーションするには、次の手順に従います。

1. WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<session-descriptor>` 要素にある `PersistentStoreType` プロパティを `jdbc` に設定し、JDBC を永続ストレージの方法として設定します。
2. WebLogic 固有のデプロイメント記述子 `weblogic.xml` の `PersistentStorePool` プロパティを使用して、JDBC 接続プールを永続ストレージ用に使うよう設定します。WebLogic Server Administration Console で定義した接続プールの名前を使用します。

   データベース接続プールの設定の詳細については、「JDBC 接続の管理」を参照してください。
3. 許可を持っているユーザに対応する接続用 ACL を設定します。データベース接続の詳細については、「JDBC 接続の管理」を参照してください。
4. JDBC ベースの永続性用に `wl_servlet_sessions` というテーブルを設定します。データベースに接続する接続プールは、このテーブルの読み取り / 書き込みアクセス権を保有する必要があります。次の表では、このテーブルを作成するときに使用するカラム名とデータ型を示します。

表 4-1 JDBC ベースの永続性のための wl_servlet_sessions テーブル

カラム名	タイプ
wl_id	最大 100 文字の可変長の英数字カラム。 例 : Oracle VARCHAR2(100)。 主キーは次のように設定しなければならない。 wl_id + wl_context_path.
wl_context_path	最大 100 文字の可変長の英数字カラム。 例 : Oracle VARCHAR2(100)。このカラムは主キーの一部として使用する（wl_id カラムの説明を参照）。
wl_is_new	1 文字のカラム。例 : Oracle CHAR(1)。
wl_create_time	20 桁の数値カラム。例 : Oracle NUMBER(20)。
wl_is_valid	1 文字のカラム。例 : Oracle CHAR(1)。
wl_session_values	ラージ バイナリ カラム。例 : Oracle LONG RAW。
wl_access_time	20 桁の数値カラム。例 : NUMBER(20)。
wl_max_inactive_interval	整数のカラム。例 : Integer。セッションが無効になるまでのクライアント リクエストの間隔の秒数。値がマイナスの場合、セッションがタイムアウトしないことを示す。

Oracle DBMS を使用している場合は、次の SQL 文を使用して wl_servlet_sessions テーブルを作成できます。

```
create table wl_servlet_sessions
  ( wl_id VARCHAR2(100) NOT NULL,
    wl_context_path VARCHAR2(100) NOT NULL,
    wl_is_new CHAR(1),
    wl_create_time NUMBER(20),
    wl_is_valid CHAR(1),
    wl_session_values LONG RAW,
    wl_access_time NUMBER(20),
    wl_max_inactive_interval INTEGER,
   PRIMARY KEY (wl_id, wl_context_path) );
```

DBMS を使用して上記の SQL 文を変更できます。

注意： ユーザは、JDBConnectionTimeoutSecs 属性が設定されたセッション データのロードに失敗するまで、JDBC セッション永続性が接続プールからの JDBC 接続を待つ最大の期間をコンフィグレーションできます。詳細については、B-10 ページの「JDBConnectionTimeoutSecs」を参照してください。

クッキーベースのセッション永続性の使用

クッキーベースのセッション永続性は、ユーザのブラウザのクッキーにすべてのセッション データを格納することで、セッション永続性のステートレスなソリューションを提供します。クッキーベースのセッション永続性が最も役に立つのは、セッションに大量のデータを格納する必要がないときです。クッキーベースのセッション永続性は、クラスタ フェイルオーバ ロジックが必要ないので、WebLogic Server のインストール環境をより簡単に管理できます。セッションが格納されるのはブラウザ内であって、サーバ上ではありません。WebLogic Server の起動と停止は、セッションを失わずに行うことができます。

クッキーベースのセッション永続性を使用すると、リクエストのバランシングは、セッション単位ではなく、リクエスト単位に行われます。クッキーベースのセッション永続性は完全にステートレスであり、どのサーバがリクエストを処理しても違いはないので (データはクライアントサイドで格納される)、リクエスト単位のバランシングは正しい動作です。

クッキーベースのセッション永続性を使用するときには、次に示すいくつかの制限事項があります。

- セッションに格納できるのは文字列属性だけです。セッションに他の種類のオブジェクトを格納した場合、IllegalArgument 例外が送出されます。
- HTTP 応答はフラッシュできません（クッキーは応答が発行される前にヘッダ データに書き込まれる必要があるため）。
- 応答のコンテンツの長さがバッファ サイズを超える場合、応答は自動的にフラッシュされ、セッション データはクッキー内では更新できません。バッファ サイズはデフォルトで 8192 バイトです。バッファ サイズは javax.servlet.ServletResponse.setBufferSize() メソッドで変更できます。
- 使用できる認証は基本的なもの（ブラウザベース）だけです。
- セッション データはクリア テキストでブラウザに送信されます。

- ユーザのブラウザを、クッキーを受け付けるようにコンフィグレーションする必要があります。
- クッキーベースのセッション永続性を使用するときは、文字列の中でカンマ (,) を使用できません。使用すると、例外が発生します。

クッキーベースのセッション永続性を設定するには、次の手順に従います。

1. WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<session-descriptor>` 要素の `PersistentStoreType` パラメータを `cookie` に設定します。
2. 必要な場合は、`PersistentStoreCookieName` パラメータを使ってクッキーに名前を設定します。デフォルトは `WLCOOKIE` です。

安全なクッキー

サービス パック 2 では、`sessionCookie` を保護するように指定するための新しいパラメータが導入されました。このパラメータを設定すると、クライアントのブラウザは、HTTPS 接続を通してだけクッキーを返送します。この機能は、クッキーの ID が安全であることを保証するするもので、HTTPS だけを使用する Web サイトでのみ使用する必要があります。この機能を有効にすると、HTTP 経由でのセッション クッキーは動作しなくなり、HTTPS 以外を使用する場所にクライアントを渡そうとしても、セッションは送信されません。この機能を使用する場合は、URLRewriting をオフにすることを強くお勧めします。アプリケーションが URL をコード化しようとした場合、セッション ID は HTTP を介して共有されます。この機能を使用するには、`weblogic.xml` に以下のコードを追加してください。

```
<session-param>
<param-name>CookieSecure</param-name>
<param-value>true</param-value>
</session-param>
<session-param>
```

URL 書き換えの使い方

状況によっては、ブラウザまたは無線デバイスがクッキーを受け入れないこともあります。この場合、クッキーによるセッション トラッキングを行うことができません。URL 書き換えを使用すると、ブラウザがクッキーを受け入れないことを WebLogic Server が検出したときに、こうした状況を自動的に置き換えることができます。URL 書き換えでは、セッション ID を Web ページのハイパーリンクにエンコードし、サーブレットはそれらをブラウザに送り返します。ユーザが以後これらのリンクをクリックすると、WebLogic Server は URL アドレスからその ID を抽出し、サーブレットが `getSession()` メソッドを呼び出すと適切な `HttpSession` を見つけ出します。

WebLogic Server で URL 書き換えを有効にするには、B-10 ページの「`URLRewritingEnabled`」属性を `true` に設定します。WebLogic 固有のデプロイメント記述子である `weblogic.xml` の `<session-descriptor>` 要素に `URLRewritingEnabled` 属性を設定します（この属性のデフォルト値は、`true` です）。

URL 書き換えのコーディングに関するガイドライン

URL 書き換えをサポートするためにどのように URL をコードで処理するかについては、いくつかのガイドラインがあります。

- 次に示すように、URL を出力ストリームに直接書き出すことは避けます。

  ```
  out.println("<a href=\"/myshop/catalog.jsp\">catalog</a>");
  ```

  代わりに、`HttpServletResponse.encodeURL()` メソッドを使用します。次に例を示します。

  ```
  out.println("<a href=\""
       + response.encodeURL("myshop/catalog.jsp")
       + "\">catalog</a>");
  ```

  `encodeURL()` メソッドを呼び出すと、URL を書き換える必要があるかどうかが調べられます。必要である場合、URL にセッション ID を組み込むことによって書き換えを行います。セッション ID は URL に付加され、セミコロンで始まります。

- WebLogic Server への応答として返される URL に加えて、リダイレクトを送信する URL をエンコードします。次に例を示します。

  ```
  if (session.isNew())
    response.sendRedirect
  (response.encodeRedirectUrl(welcomeURL));
  ```

  WebLogic Server はセッションが新しいときには、ブラウザがクッキーを受け入れる場合でも URL 書き換えを使用します。これは、セッションの最初ではサーバはブラウザがクッキーを受け入れるかどうかを判断できないからです。

- サーブレットは、`HttpServletRequest.isRequestedSessionIdFromCookie()` メソッドから返されるブール値をチェックすることによって、特定のセッション ID がクッキーから受け取られたかどうかを確認できます。WebLogic Server アプリケーションは適切に応答するか、WebLogic Server による URL 書き換えに依存します。

URL 書き換えと Wireless Access Protocol（WAP）

WAP アプリケーションを作成する場合、WAP プロトコルはクッキーをサポートしていないため、URL を書き換える必要があります。また、一部の WAP デバイスでは、URL の長さが 128 文字（パラメータも含む）に制限されます。これにより、URL 書き換えによって転送できるデータ サイズが制限されます。パラメータ用の領域を増やすために、WebLogic Server によってランダムに生成されるセッション ID のサイズを制限できます。そのためには、B-10 ページの「`IDLength`」属性を使用してバイト数を指定します。

WAP 有効化属性を設定すると、余分な領域を節約できます。これによって、WebLogic Server は主情報と副情報を URL と共に送信しなくなります。WAP 有効化属性は、Administration Console で、[サーバ | コンフィグレーション | HTTP] タブを選択して設定できます。

5 Web アプリケーションでのセキュリティのコンフィグレーション

この章では、Web アプリケーションでセキュリティをコンフィグレーションする方法について説明します。

Web アプリケーションでのセキュリティのコンフィグレーションの概要

Web アプリケーションにセキュリティを設定するには、認証を使用するか、Web アプリケーション内の特定のリソースへのアクセスを制限するか、またはサーブレット コードでセキュリティの呼び出しを使用します。複数のタイプのセキュリティ レルムを使用できます。セキュリティ レルムについては、「セキュリティの基礎概念」を参照してください。セキュリティ レルムは、複数の仮想ホストの間で共有されます。

Web アプリケーション用の認証の設定

Web アプリケーションの認証をコンフィグレーションするには、web.xml デプロイメント記述子の <login-config> 要素を使用します。この要素では、ユーザの資格が収められるセキュリティ レルム、認証方式、および認証用リソースの場所を定義します。セキュリティ レルムの詳細については、「セキュリティの基礎概念」を参照してください。

Web アプリケーション用の認証を設定するには、次の手順を実行します。

1. テキスト エディタで web.xml デプロイメント記述子を開くか、Administration Console を使用します。詳細については、1-6 ページの「Web アプリケーション開発者向けツール」を参照してください。
2. <auth-method> 要素を使用して認証メソッドを指定します。選択できる方式は以下のとおりです。

   BASIC

   基本認証では、Web ブラウザを使用してユーザ名 / パスワード ダイアログ ボックスを表示します。このユーザ名とパスワードは、セキュリティ レルムに対して認証されます。

   FORM

   フォーム ベースの認証では、ユーザ名とパスワードが指定された HTML フォームを返す必要があります。フォーム要素から返されるフィールドは j_username と j_password で、アクション属性は j_security_check でなければなりません。次に、FORM 認証を使用するための HTML コードのサンプルを示します。

   ```
   <form method="POST" action="j_security_check">
      <input type="text" name="j_username">
      <input type="password" name="j_password">
   </form>
   ```

   この HTML フォームの生成に使用するリソースは、HTML ページ、JSP、またはサーブレットです。このリソースは、<form-login-page> 要素を使って定義します。

   HTTP セッション オブジェクトはログイン ページが提供されるときに作成されます。したがって、session.isNew() メソッド

は、認証の成功後に提供されるページから呼び出されると FALSE を返します。

CLIENT-CERT
クライアント証明書を使用してリクエストを認証します。詳細については、「SSL プロトコルのコンフィグレーション」を参照してください。

3. FORM 認証を選択した場合、HTML ページの生成に使用するリソースの場所（<form-login-page> 要素を使用）、および失敗した認証に応答するリソースの場所（<form-error-page> 要素を使用）も定義します。FORM 認証をコンフィグレーションする手順については、A-26 ページの「form-login-config 要素」を参照してください。

4. <realm-name> 要素を使用して認証のレルムを指定します。特定のレルムを指定しなかった場合は、Administration Console の [Web アプリケーション | コンフィグレーション | その他] タブにある [認証レルム名] フィールドで定義されたレルムが使用されます。詳細については、A-26 ページの「form-login-config 要素」を参照してください。

5. Web アプリケーションごとに別々にログインを定義する場合は、5-4 ページの「複数の Web アプリケーション、クッキー、および認証」を参照してください。定義しない場合は、同じクッキーを使用するすべての Web アプリケーションでの認証に 1 つのサインオンが使用されます。

複数の Web アプリケーション、クッキー、および認証

デフォルトでは、WebLogic Server はすべての Web アプリケーションに同じクッキー名（JSESSIONID）を割り当てます。どの種類の認証を使用する場合でも、同じクッキー名を使用する Web アプリケーションでは、認証用に 1 つのサインオンを使用します。ユーザが認証されると、その認証は、同じクッキー名を使用するすべての Web アプリケーションへのリクエストに対して有効になります。ユーザは再び認証を要求されることはありません。

Web アプリケーションごとに個別の認証が必要な場合は、Web アプリケーションにユニークなクッキー名またはクッキー パスを指定できます。CookieName パラメータでクッキー名を指定し、CookiePath パラメータでクッキー パスを指定します。これらのパラメータは、<session-descriptor> 要素の WebLogic 固有のデプロイメント記述子 weblogic.xml で定義されています。詳細については、B-4 ページの「session-descriptor 要素」を参照してください。

クッキー名を保持しつつ Web アプリケーションごとに別々の認証が必要な場合は、Web アプリケーションごとにクッキー パラメータ（CookiePath）を変えることができます。

Web アプリケーション リソースへのアクセスの制限

Web アプリケーションの特定のリソース（サーブレット、JSP、または HTML ページ）へのアクセスを制限するには、それらのリソースにセキュリティ制約を適用します。

セキュリティ制約をコンフィグレーションするには、次の手順に従います。

1. テキスト エディタで web.xml および weblogic.xml デプロイメント記述子を開くか、Administration Console を使用します。詳細については、1-6 ページの「Web アプリケーション開発者向けツール」を参照してください。

2. セキュリティ レルムの 1 つまたは複数のプリンシパルにマップされるロールを定義します。ロールを定義するには、Web アプリケーション デプロイメント記述子で A-26 ページの「security-role 要素」を使います。次に、WebLogic 固有のデプロイメント記述子 `weblogic.xml` で B-2 ページの「security-role-assignment 要素」を使用して、これらのロールをレルムのプリンシパルにマップします。

3. `<web-resource-collection>` 要素にネストされる `<url-pattern>` 要素を使用して、セキュリティ制約を適用する Web アプリケーション リソースを定義します。`<url-pattern>` は、ディレクトリ、ファイル名、または `<servlet-mapping>` です。

   また、セキュリティ制約を Web アプリケーション全体に適用する場合は、次のエントリを使用します。

   ```
   <url-pattern>/*</url-pattern>
   ```

4. `<web-resource-collection>` 要素にネストされる `<http-method>` 要素を使用して、セキュリティ制約を適用する HTTP メソッド（`GET` または `POST`）を定義します。HTTP メソッドごとに、別々の `<http-method>` 要素を使用します。

5. `<user-data-constraint>` メソッドにネストされる `<transport-guarantee>` 要素を使用して、クライアントとサーバ間の通信に SSL を使用するかどうかを定義します。

コード リスト 5-1　セキュリティ制約のサンプル

`web.xml` のエントリ :

```
<security-constraint>
     <web-resource-collection>
          <web-resource-name>SecureOrdersEast</web-resource-name>
          <description>
             Security constraint for
             resources in the orders/east directory
          </description>
          <url-pattern>/orders/east/*</url-pattern>
          <http-method>POST</http-method>
          <http-method>GET</http-method>
     </web-resource-collection>
     <auth-constraint>
          <description>
           constraint for east coast sales
          </description>
          <role-name>east</role-name>
          <role-name>manager</role-name>
```

```
      </auth-constraint>
      <user-data-constraint>
           <description>SSL not required</description>
           <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
</security-constraint>

...
```

サーブレットでのユーザとロールのプログラマティカルな使い方

javax.servlet.http.HttpServletRequest.isUserInRole(String role) メソッドを使用すると、サーブレット コード中のユーザとロールにプログラマティックにアクセスできるようサーブレットを記述できます。文字列の role は、Web アプリケーション デプロイメント記述子の <servlet> 宣言の <security-role-ref> 要素にネストされる <role-name> 要素に定義される名前にマップされます。<role-link> 要素は、Web アプリケーション デプロイメント記述子の <security-role> 要素に定義される <role-name> にマップされます。

次のリストで例を示します。

コード リスト 5-2 セキュリティ ロール マッピングの例

サーブレット コード

```
isUserInRole("manager");
```

web.xml エントリ

```
<servlet>
. . .
   <role-name>manager</role-name>
   <role-link>mgr</role-link>
. . .
</servlet>

<security-role>
   <role-name>mgr</role-name>
</security-role>
```

weblogic.xml エントリ

```
<security-role-assignment>
   <role-name>mgr</role-name>
   <principal-name>al</principal-name>
   <principal-name>george</principal-name>
   <principal-name>ralph</principal-name>
</security-role-ref>
```

6 アプリケーション イベントとリスナ

この章では、Web アプリケーションのイベントとリスナをコンフィグレーションおよび使用する方法について説明します。

アプリケーション イベントとリスナの概要

アプリケーション イベントとは、サーブレット コンテキストのステートの変更（各 Web アプリケーションは独自のサーブレット コンテキストを使用する）、または HTTP セッション オブジェクトのステートの変更を通知するものです。これらのステートの変更に応答するイベント リスナ クラスを作成して、Web アプリケーションでアプリケーション イベントとリスナ クラスをコンフィグレーションおよびデプロイします。

サーブレット コンテキスト イベントの場合、イベント リスナ クラスは、Web アプリケーションがデプロイされるときやアンデプロイされているとき（または WebLogic Server が停止するとき）、および、属性が追加、削除、置換されたときに、通知を受け取ることができます。

HTTP セッション イベントの場合、イベント リスナ クラスは、HTTP セッションがアクティブ化されたかパッシベーションされようとしているとき、および、HTTP セッション属性が追加、削除、置換されたときに、通知を受け取ることができます。

Web アプリケーション イベントは次の目的で使用します。

- Web アプリケーションがデプロイされるときや停止されるときにデータベース接続を管理する。
- カウンタを作成する。
- HTTP セッションとその属性のステートをモニタする。

注意： アプリケーション イベントは、Sun Microsystems が提供する Java サーブレット仕様バージョン 2.3 の新しい機能です。バージョン 2.3 は、サーブレット仕様で提案されている最新のドラフトです。アプリケーションでアプリケーション イベントを使用する計画がある場合、この仕様がまた確定しておらず、将来、変更される可能性があることに注意してください。

サーブレット 2.3 仕様は、J2EE 1.3 仕様の一部です。J2EE 1.3 機能の使用については、6-2 ページの「WebLogic Server 6.1 と J2EE 1.2 および J2EE 1.3」を参照してください。

WebLogic Server 6.1 と J2EE 1.2 および J2EE 1.3

BEA WebLogic Server 6.1 は、高度な J2EE 1.3 の機能を実装する最初の e- コマース トランザクション プラットフォームです。J2EE のルールに準拠するために、BEA Systems では 2 つの別個のダウンロードを用意しています。1 つは J2EE 1.3 の機能が有効になっているもの、1 つは J2EE 1.2 の機能に制限されているものです。いずれのダウンロードもコンテナは同じですが、利用可能な API だけ異なります。

J2EE 1.2 の機能に加えて J2EE 1.3 の機能を備える WebLogic Server 6.1

このダウンロードでは、WebLogic Server はデフォルトで J2EE 1.3 の機能を使用して動作します。それらの機能には、EJB 2.0、JSP 1.2、サーブレット 2.3、および J2EE コネクタ アーキテクチャ 1.0 が含まれます。J2EE 1.3 の機能を有効にして WebLogic Server 6.1 を実行しても、J2EE 1.2 アプリケーションはそのままフルサポートされます。J2EE 1.3 機能の実装では、適切な API 仕様の最終ではないバージョンが使用されます。したがって、J2EE 1.3 の新機能を使用する BEA WebLogic Server 6.1 用に開発されたアプリケーション コードは、BEA WebLogic Server の今後のリリースでサポートされる J2EE 1.3 プラットフォームとは互換性を持たない場合があります。

J2EE 1.2 認定の WebLogic Server 6.1

このダウンロードでは、WebLogic Server はデフォルトで J2EE 1.3 機能が無効な状態で動作し、J2EE 1.2 の仕様と規定に完全に準拠します。

サーブレット コンテキスト イベント

次の表は、サーブレット コンテキスト イベントのタイプ、イベント リスナ クラスがイベントに応答するために実装すべきインタフェース、およびイベントが発生したときに起動されるメソッドを示しています。

イベントのタイプ	インタフェース	メソッド
サーブレット コンテキストが作成された。	`javax.servlet.ServletContextListener`	`contextInitialized()`

イベントのタイプ	インタフェース	メソッド
サーブレット コンテキストが停止されようとしている。	javax.servlet.ServletContextListener	contextDestroyed()
属性が追加された。	javax.servlet. ServletContextAttributesListener	attributeAdded()
属性が削除された。	javax.servlet. ServletContextAttributesListener	attributeRemoved()
属性が置き換えられた。	javax.servlet. ServletContextAttributesListener	attributeReplaced()

HTTP セッション イベント

次の表は、HTTP セッション イベントのタイプ、イベント リスナ クラスがイベントへの応答に実装すべきインタフェース、およびイベントが発生したときに起動されるメソッドを示しています。

イベントのタイプ	インタフェース	メソッド
HTTP セッションがアクティブ化された。	javax.servlet.http. HttpSessionListener	sessionCreated()
HTTP セッションがパッシベーションされようとしている。	javax.servlet.http. HttpSessionListener	sessionDestroyed()
属性が追加された。	javax.servlet.http. HttpSessionAttributesListener	attributeAdded()
属性が削除された。	javax.servlet.http. HttpSessionAttributesListener	attributeRemoved()

イベントのタイプ	インタフェース	メソッド
属性が置き換えられた。	javax.servlet.http.HttpSessionAttributesListener	attributeReplaced()

注意： サーブレット 2.3 仕様にも javax.servlet.http.HttpSessionBindingListener インタフェースと javax.servlet.http.HttpSessionActivationListener インタフェースが含まれています。これらのインタフェースは、セッション属性として格納されるオブジェクトによって実装され、web.xml へのイベント リスナの登録を必要としません。詳細については、これらのインタフェースの Javadoc を参照してください。

イベント リスナのコンフィグレーション

イベント リスナをコンフィグレーションするには、次の手順に従います。

1. イベント リスナを作成する Web アプリケーションの web.xml デプロイメント記述子をテキスト エディタで開くか、Administration Console に統合されている Web アプリケーション デプロイメント記述子エディタを使います（詳細については、「Web アプリケーション デプロイメント記述子エディタ ヘルプ」を参照してください）。web.xml ファイルは、Web アプリケーションの WEB-INF ディレクトリにあります。
2. <listener> 要素を使ってイベント宣言を追加します。イベント宣言は、イベントが発生するときに起動されるリスナ クラスを定義します。<listener> 要素は、<filter> 要素と <filter-mapping> 要素のすぐ後ろで、<servlet> 要素のすぐ前に指定します。それぞれのイベント タイプに複数のリスナ クラスを指定できます。WebLogic Server は、デプロイメント記述子に記述されている順にイベント リスナを起動します（停止イベントだけは、逆順で起動されます）。次に例を示します。

```
<listener>
  <listener-class>myApp.myContextListenerClass</listener-class>
</listener>

<listener>
  <listener-class>myApp.mySessionAttributeListenerClass</listen
```

```
er-class>
</listener>
```

3. リスナ クラスを作成し、デプロイします。詳細については、次の「リスナ クラスの作成」を参照してください。

リスナ クラスの作成

リスナ クラスを作成するには、次の手順に従います。

1. クラスが応答するイベントのタイプに対して適切なインタフェースを実装する新しいクラスを作成します。これらのインタフェースのリストについては、6-3 ページの「サーブレット コンテキスト イベント」または 6-4 ページの「HTTP セッション イベント」を参照してください。作業の開始に利用できるサンプルのテンプレートについては、6-7 ページの「リスナ クラスのテンプレート」を参照してください。
2. 引数をとらないパブリック コンストラクタを作成します。
3. インタフェースの必須メソッドを実装します。詳細については、J2EE API リファレンス（Javadoc）を参照してください。
4. コンパイル済みのイベント リスナ クラスを Web アプリケーションの `WEB-INF\classes` ディレクトリにコピーするか、または、それらを `jar` ファイルにパッケージ化してからその `jar` ファイルを Web アプリケーションの `WEB-INF\lib` ディレクトリにコピーします。

次の便利なクラスは、リスナ クラスのリスナ メソッドに渡されます。

`javax.servlet.http.HttpSessionEvent`
: HTTP セッション オブジェクトへのアクセスを提供します。

`javax.servlet.ServletContextEvent`
: サーブレット コンテキスト オブジェクトへのアクセスを提供します。

`javax.servlet.ServletContextAttributeEvent`
: サーブレット コンテキストとその属性へのアクセスを提供します。

`javax.servlet.http.HttpSessionBindingEvent`
: HTTP セッションとその属性へのアクセスを提供します。

リスナ クラスのテンプレート

次の例は、リスナ クラスの基本的なテンプレートです。

サーブレット コンテキスト リスナの例

```
package myApp;
import javax.servlet.*;

public final class myContextListenerClass implements
   ServletContextListener {
    public void contextInitialized(ServletContextEvent event) {

      /* このメソッドは、サーブレット コンテキストが初期化されたとき
         (Web アプリケーションがデプロイされたとき）に呼び出される。
         この時点で、サーブレット コンテキストに関連するデータを初期化できる
      */

    }

    public void contextDestroyed(ServletContextEvent event) {

      /* このメソッドは、サーブレット コンテキスト（Web アプリケーション）が
         アンデプロイされたとき、または WebLogic Server が
         シャットダウンしたときに呼び出される
      */

    }
}
```

HTTP セッション属性リスナの例

```
package myApp;
import javax.servlet.*;

public final class mySessionAttributeListenerClass implements
   HttpSessionAttributesListener {

    public void attributeAdded(HttpSessionBindingEvent sbe) {
      /* このメソッドは、属性がセッションに追加されたときに
         呼び出される
      */
    }
```

```
    public void attributeRemoved(HttpSessionBindingEvent sbe) {
      /* このメソッドは、属性がセッションから削除されたときに
         呼び出される
      */
    }
    public void attributeReplaced(HttpSessionBindingEvent sbe) {
      /* このメソッドは、属性がセッションで置き換えられたときに
         呼び出される
      */
    }
}
```

その他の情報源

- 『Web アプリケーションのアセンブルとコンフィグレーション』
- 「Web アプリケーションのデプロイメント記述子の記述」
- Sun Microsystems のサーブレット 2.3 仕様
- J2EE API リファレンス（Javadoc）
- Sun Microsystems の The J2EE Tutorial

7　フィルタ

この章では、Web アプリケーションでのフィルタの使用に関する情報を提供します。

フィルタの概要

フィルタは、Web アプリケーションのリソースに対する要求に応答して呼び出される Java クラスです。リソースには、Java サーブレット、JavaServer Pages（JSP）、および HTTP ページや画像などの静的リソースがあります。フィルタを使用すると、要求をインターセプトして、応答オブジェクトおよび要求オブジェクトを検証したり変更したりするタスクなどを実行できます。

フィルタは、開発者が既存のリソースのコーディングを変更できず、そのリソースの動作を変更する必要がある状況を主に想定した、高度な J2EE 機能です。一般に、フィルタを使ってリソースを変更するよりは、コードを変更してリソースの動作自体を変更した方が効率的です。状況によっては、フィルタを使うことによって、アプリケーションが不必要に複雑になり、パフォーマンスが低下することがあります。

注意： フィルタは、Sun Microsystems が提供する Java サーブレット仕様バージョン 2.3 の新しい機能です。バージョン 2.3 は、サーブレット仕様で提案されている最新のドラフトです。アプリケーションでフィルタを使用する計画がある場合、この仕様がまた確定しておらず、将来、変更される可能性があることに注意してください。

サーブレット 2.3 仕様は、J2EE 1.3 仕様の一部です。J2EE 1.3 機能の使用については、7-2 ページの「WebLogic Server 6.1 と J2EE 1.2 および J2EE 1.3」を参照してください。

WebLogic Server 6.1 と J2EE 1.2 および J2EE 1.3

BEA WebLogic Server 6.1 は、高度な J2EE 1.3 の機能を実装する最初の e- コマース トランザクション プラットフォームです。J2EE のルールに準拠するために、BEA Systems では 2 つの別個のダウンロードを用意しています。1 つは J2EE 1.3 の機能が有効になっているもの、1 つは J2EE 1.2 の機能に制限されているものです。いずれのダウンロードもコンテナは同じですが、利用可能な API だけ異なります。

J2EE 1.2 の機能に加えて J2EE 1.3 の機能を備える WebLogic Server 6.1

このダウンロードでは、WebLogic Server はデフォルトで J2EE 1.3 の機能を使用して動作します。それらの機能には、EJB 2.0、JSP 1.2、サーブレット 2.3、および J2EE コネクタ アーキテクチャ 1.0 が含まれます。J2EE 1.3 の機能を有効にして WebLogic Server 6.1 を実行しても、J2EE 1.2 アプリケーションはそのままフルサポートされます。J2EE 1.3 機能の実装では、適切な API 仕様の最終ではないバージョンが使用されます。したがって、J2EE 1.3 の新機能を使用する BEA WebLogic Server 6.1 用に開発されたアプリケーション コードは、BEA WebLogic Server の今後のリリースでサポートされる J2EE 1.3 プラットフォームとは互換性を持たない場合があります。

J2EE 1.2 認定の WebLogic Server 6.1

このダウンロードでは、WebLogic Server はデフォルトで J2EE 1.3 機能が無効な状態で動作し、J2EE 1.2 の仕様と規定に完全に準拠します。

フィルタの動作としくみ

フィルタは、Web アプリケーションのコンテキストで定義します。フィルタは特定の名前のリソースまたはリソースのグループ（URL パターンに基づく）に対するリクエストを横取りして、フィルタ内でコードを実行します。それぞれのリソースまたはリソースのグループに対して、単一のフィルタ、またはチェーンと呼ばれる特定の順序で起動される複数のフィルタを指定できます。

フィルタは、リクエストを横取りするとき、HTTP リクエストと応答へのアクセスを提供する `javax.servlet.ServletRequest` オブジェクトと `javax.servlet.ServletResponse` オブジェクト、および `javax.servlet.FilterChain` オブジェクトにアクセスできます。`FilterChain` オブジェクトには、順番に起動できるフィルタのリストが含まれています。フィルタは、作業を終了すると、チェーン内の次のフィルタを起動する、リクエストをブロックする、例外を送出する、本来リクエストされていたリソースを起動する、のうちのいずれかの処理を行うことができます。

本来のリソースが起動されると、制御は、チェーン内のリストの最後にあるフィルタに返されます。そのあとで、このフィルタは、応答ヘッダとデータの検査および変更、リクエストのブロック、例外の送出、チェーンの最後より 1 つ手前にあるフィルタの起動のいずれかを行うことができます。この処理はフィルタのチェーン内において逆順で続行されます。

フィルタの用途

フィルタは次の機能を行うときに便利です。

- ロギング機能の実装
- ユーザが作成したセキュリティ機能の実装
- デバッグ
- 暗号化
- データの圧縮
- クライアントに送信される応答の変更（後処理であっても、応答はアプリケーションのパフォーマンスを劣化させるおそれがあります）

フィルタのコンフィグレーション

Web アプリケーションの web.xml デプロイメント記述子を使って、フィルタをアプリケーションの一部としてコンフィグレーションします。デプロイメント記述子では、フィルタを宣言してから、そのフィルタを Web アプリケーションの URL パターンまたは特定のサーブレットにマップします。宣言できるフィルタの数に制限はありません。

フィルタのコンフィグレーション

フィルタをコンフィグレーションするには、次の手順に従います。

1. テキスト エディタで web.xml デプロイメント記述子を開くか、Administration Console を使用します。詳細については、1-6 ページの「Web アプリケーション開発者向けツール」を参照してください。web.xml ファイルは、Web アプリケーションの WEB-INF ディレクトリにあります。
2. フィルタ宣言を追加します。<filter> 要素は、フィルタの宣言、フィルタの名前の定義、およびフィルタを実行する Java クラスの指定を行います。<filter> 要素は、<context-param> 要素のすぐ後ろで、<listener> 要素と <servlet> 要素のすぐ前に指定します。次に例を示します。

```
<filter>
  <icon>
    <small-icon>MySmallIcon.gif</small-icon>
    <large-icon>MyLargeIcon.gif</large-icon>
  </icon>
  <filter-name>myFilter1</filter-name>
  <display-name>filter 1</display-name>
  <description>This is my filter</description>
  <filter-class>examples.myFilterClass</filter-class>
</filter>
```

icon、description、display-name の各要素は省略可能です。

3. <filter> 要素の内部に 1 つまたは複数の初期化パラメータを指定します。次に例を示します。

```
<filter>
  <icon>
    <small-icon>MySmallIcon.gif</small-icon>
    <large-icon>MyLargeIcon.gif</large-icon>
  </icon>
  <filter-name>myFilter1</filter-name>
  <display-name>filter 1</display-name>
  <description>This is my filter</description>
  <filter-class>examples.myFilterClass</filter-class>
  <init-param>
    <param-name>myInitParam</param-name>
    <param-value>myInitParamValue</param-value>
  </init-param>
</filter>
```

Filter クラスは FilterConfig.getInitParameter() メソッドまたは FilterConfig.getInitParameters() メソッドを使って初期化パラメータを読み取ることができます。

4. フィルタ マッピングを追加します。<filter-mapping> 要素は、URL パターンまたはサーブレット名を基にしてどのフィルタを実行するかを指定します。<filter-mapping> 要素は、<filter> 要素のすぐ後ろに指定します。

   - URL パターンを使ったフィルタ マッピングを作成するには、フィルタの名前と URL パターンを指定します。URL のパターン マッチングは、Sun Microsystems のサーブレット 2.3 仕様のセクション 11.1 で指定されている規則に従って実行されます。たとえば、次の filter-mapping は /myPattern/ を含むリクエストに myFilter をマップします。

     ```
     <filter-mapping>
       <filter-name>myFilter</filter-name>
     ```

```
    <url-pattern>/myPattern/*</url-pattern>
</filter-mapping>
```

- 特定のサーブレットに対するフィルタ マッピングを作成するには、Web アプリケーションに登録されたサーブレットの名前にフィルタをマップします。たとえば、次のコードは `myServlet` というサーブレットに `myFilter` フィルタをマップします。

```
<filter-mapping>
  <filter-name>myFilter</filter-name>
  <servlet-hame>myServlet</servlet-name>
</filter-napping>
```

5. フィルタのチェーンを作成するには、複数のフィルタ マッピングを指定します。詳細については、7-6 ページの「フィルタのチェーンのコンフィグレーション」を参照してください。

フィルタのチェーンのコンフィグレーション

WebLogic Server は、送られてくる HTTP リクエストに一致するすべてのフィルタ マッピングのリストを作成することで、フィルタのチェーンを作成します。リストの順序は次の順番で決定します。

1. リクエストに一致する `url-pattern` を含む `filter-mapping` のあるフィルタは、`web.xml` デプロイメント記述子に記述された順序でチェーンに追加されます。
2. リクエストに一致する `servlet-name` を含む `filter-mapping` のあるフィルタは、URL パターンに一致するフィルタの後でチェーンに追加されます。
3. チェーン内の最後の項目は常に、本来リクエストされたリソースです。

フィルタ クラスでは、`FilterChain.doFilter()` メソッドを使ってチェーン内の次の項目を起動します。

フィルタの作成

フィルタ クラスを作成するには、javax.servlet.Filter インタフェースを実装します。このインタフェースの次のメソッドを実装する必要があります。

`doFilter()`
: このメソッドは、リクエスト オブジェクトと応答オブジェクトの検査と変更、ロギングなど他のタスクの実行、チェーン内の次のフィルタの起動、または、それ以上の処理のブロックのために使います。

`GetFilterConfig()`
: このメソッドは、javax.servlet.FilterConfig オブジェクトへのアクセスを取得するために使います。

`setFilterConfig()`
: このメソッドは、javax.servlet.FilterConfig オブジェクトを設定するために使います。

フィルタの名前、`ServletContext`、およびフィルタの初期化パラメータにアクセスするために、`FilterConfig` オブジェクトに対して利用できるメソッドが他にいくつかあります。詳細については、Sun Microsystems の `javax.servlet.FilterConfig` に関する J2EE Javadoc を参照してください。

チェーン内の次の項目（次のフィルタ、または本来のリソースがチェーン内の次の項目である場合は本来のリソース）にアクセスするには、`FilterChain.doFilter()` メソッドを呼び出します。

フィルタ クラスの例

次のコード例は、Filter クラスの基本構造を示しています。

コード リスト 7-1 フィルタ クラスの例

```
import javax.servlet.*;
public class Filter1Impl implements Filter
{
    private FilterConfig filterConfig;

    public void doFilter(ServletRequest req,
       ServletResponse res, FilterChain fc)
        throws java.io.IOException, javax.servlet.ServletException
    {
     // ロギングなどのタスクを実行
     //...

      fc.doFilter(req,res); // チェーン内の次の項目（別のフィルタ
                            // または元々要求されていたリソースの
                            // いずれか）を呼び出す

    }

    public FilterConfig getFilterConfig()
    {
      // タスクの実行
      return filterConfig;
    }

    public void setFilterConfig(FilterConfig cfg)
    {
      // タスクの実行
      filterConfig = cfg;
    }
}
```

サーブレット応答オブジェクトでのフィルタ処理

サーブレットによって生成された出力にデータを追加することで、フィルタをサーブレットの出力の後処理に使用できます。ただし、サーブレットの出力を取り込むには、応答にラッパーを作成する必要があります（サーブレットが実行を完了し、制御がチェーン内の最後のフィルタに戻される前に、サーブレットの出力バッファは自動的にフラッシュされ、クライアントに送信されるので、本来の応答オブジェクトは使用できません）。そのようなラッパーを作成すると、WebLogic Server はメモリで出力の追加コピーを処理する必要が生じ、パフォーマンスが低下することがあります。

応答オブジェクトやリクエスト オブジェクトのラッピングの詳細については、Sun Microsystems の J2EE Javadoc の `javax.servlet.http.HttpServletResponseWrapper` と `javax.servlet.http.HttpServletRequestWrapper` を参照してください。

その他の情報源

- 『Web アプリケーションのアセンブルとコンフィグレーション』
- 「Web アプリケーションのデプロイメント記述子の記述」
- Sun Microsystems のサーブレット 2.3 仕様
- J2EE API リファレンス（Javadoc）
- Sun Microsystems の The J2EE Tutorial

8 Web アプリケーションのデプロイメント記述子の記述

この章では、Web アプリケーション デプロイメント記述子を作成する方法について説明します。

Web アプリケーションのデプロイメント記述子の概要

WebLogic Server は、Web アプリケーションを定義するために標準 J2EE `web.xml` デプロイメント記述子を使用します。WebLogic 固有のデプロイメント記述子である `weblogic.xml` を合わせて必要とするアプリケーションもあります。これらのデプロイメント記述子を使用して、Web アプリケーション用のコンポーネントと操作パラメータを定義します。デプロイメント記述子は、標準のテキスト ファイルであり、XML 表記法を使ってフォーマットされています。ユーザは、これらを Web アプリケーションにパッケージ化します。Web アプリケーションの詳細については、「Web アプリケーションの基本事項」を参照してください。

デプロイメント記述子 `web.xml` は、Sun Microsystems のサーブレット 2.3 仕様で定義されています。このデプロイメント記述子を使用して、J2EE 準拠のアプリケーション サーバに Web アプリケーションをデプロイできます。

デプロイメント記述子 `weblogic.xml` は、WebLogic Server 上で稼働する Web アプリケーションに固有のデプロイメント プロパティを定義します。`weblogic.xml` は、すべての Web アプリケーションで必要なわけではありません。

デプロイメント記述子を編集するためのツール

デプロイメント記述子の編集には、次のツールのいずれかを使用できます。

- WebLogic Server Administration Console に統合されたデプロイメント記述子エディタ。詳細については、「Web アプリケーション デプロイメント記述子エディタ ヘルプ」を参照してください。
- Windows のメモ帳、`emacs`、`vi`、または使い慣れた IDE などのプレーンなテキスト エディタ。

- 現在、BEA WebLogic では、XML ファイルの作成と編集のために Ensemble のシンプルで使いやすいツールを用意しています。このツールを使うと、指定した DTD または XML スキーマに従って XML コードの有効性を検証できます。この XML エディタは、Windows または Solaris のマシンで使用でき、BEA dev2dev からダウンロードできます。
- スケルトン デプロイメント記述子を作成するときには、ANT ユーティリティを利用できます。ANT タスクによって、Web アプリケーションを含むディレクトリが調べられ、その Web アプリケーションで検出されたファイルを基にデプロイメント記述子が作成されます。ANT タスクでは、目的のコンフィグレーション、マッピング、その他の情報のすべてがわかるわけではないので、ANT タスクが作成するスケルトン デプロイメント記述子は不完全なものです。テキスト エディタ、XML エディタ、または Administration Console を使用して、デプロイメント記述子を使った Web アプリケーションのコンフィグレーションを完全なものにすることができます。

  詳細については、「Web アプリケーションのパッケージ化」を参照してください。

web.xml デプロイメント記述子の記述

この節では、`web.xml` デプロイメント記述子を作成する手順について説明します。Web アプリケーションのコンポーネントによっては、Web アプリケーションのコンフィグレーションとデプロイに、ここで示す要素のすべてが必要なわけではないことがあります。

`web.xml` ファイル内の要素は、このドキュメントで取り上げる順で入力しなければなりません。

web.xml ファイルの主な作成手順

- 8-5 ページの「手順 1 : デプロイメント記述子ファイルの作成」
- 8-5 ページの「手順 2 : DOCTYPE 文の作成」
- 8-6 ページの「手順 3 : web.xml ファイルの本文の作成」

- 8-7 ページの「手順 4 : デプロイメント時の属性の定義」
- 8-7 ページの「手順 5 : コンテキスト パラメータの定義」
- 8-8 ページの「手順 6 : フィルタのコンフィグレーション（サーブレット 2.3 仕様のみ）」
- 8-9 ページの「手順 7 : フィルタ マッピングの定義（サーブレット 2.3 仕様のみ）」
- 8-10 ページの「手順 8 : アプリケーション リスナのコンフィグレーション（サーブレット 2.3 仕様のみ）」
- 8-10 ページの「手順 9 : サーブレットのデプロイ」
- 8-13 ページの「手順 10:URL へのサーブレットのマッピング」
- 8-14 ページの「手順 11 : セッション タイムアウト値の定義」
- 8-14 ページの「手順 12 : MIME マッピングの定義」
- 8-15 ページの「手順 13 : ウェルカム ページの定義」
- 8-15 ページの「手順 14 : エラー ページの定義」
- 8-16 ページの「手順 15 : JSP タグ ライブラリ記述子の定義」
- 8-17 ページの「手順 16 : 外部リソースの参照」
- 8-17 ページの「手順 17 : セキュリティ制約の設定」
- 8-19 ページの「手順 18 : ログイン認証の設定」
- 8-20 ページの「手順 19 : セキュリティ ロールの定義」
- 8-21 ページの「手順 20 : 環境エントリの設定」
- 8-21 ページの「手順 21 : エンタープライズ JavaBean（EJB）リソースの参照」

WebLogic Server のサンプルおよび例をインストールした場合は、Pet Store サンプルの `web.xml` および `weblogic.xml` ファイルで、Web アプリケーションのデプロイメント記述子の実際の例を参照できます。これらのファイルは、WebLogic Server の配布ディレクトリ、`\samples\PetStore\source\dd\war\WEB-INF` にあります。

8-4 Web アプリケーションのアセンブルとコンフィグレーション

web.xml ファイルの詳しい作成手順

手順 1 : デプロイメント記述子ファイルの作成

ファイル名を web.xml として、Web アプリケーションの WEB-INF ディレクトリに入れます。任意のテキスト エディタを使用します。

手順 2 : DOCTYPE 文の作成

DOCTYPE 文は、デプロイメント記述子のドキュメント タイプ定義（DTD）ファイルの場所とバージョンを指しています。このヘッダは外部 URL の java.sun.com を参照していますが、WebLogic Server には独自の DTD ファイルが用意されているので、ホスト サーバがインターネットにアクセスする必要はありません。ただし、この <!DOCTYPE...> 要素を web.xml ファイルに入れて、外部 URL を参照するようにしなければなりません。この要素内の DTD のバージョンはこのデプロイメント記述子のバージョンを識別するためのものだからです。

次の DOCTYPE 文のいずれかを使用してください。

- フィルタやアプリケーション イベントのようなサーブレット 2.3 仕様の機能を使用している場合、次の DOCTYPE 文を使用します。

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">
```

  注意 : サーブレット仕様バージョン 2.3 の実装は、サーブレット仕様の最終草案 1 をベースにしており、変更される可能性があります。バージョン 2.3 で導入された機能を使用する計画がある場合、この仕様がまた確定しておらず、将来、変更される可能性があることに注意してください。最終草案 2 で追加された機能はサポートされていません。

- サーブレット 2.3 仕様の機能を使用する必要がない場合は、次の DOCTYPE 文を使用します。

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//
DTD WebApplication 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
```

手順 3 : web.xml ファイルの本文の作成

<web-app> タグの開始タグと終了タグを 1 組にしてすべてのエントリを挟みます。

<web-app>

この Web アプリケーションを記述するすべての要素は

<web-app> 要素内に入る。

</web-app>

XML では、上記のように、プロパティ名または値を開始および終了タグで囲むことでプロパティを定義します。開始タグ、本文（プロパティ名または値）、および終了タグは、ひとまとめにして要素と呼ばれます。一部の要素は開始タグと終了タグを組にしていませんが、空タグと呼ばれる属性を持つ 1 つのタグを使用します。このテキストでは、わかりやすいように、その他の要素に含まれる要素がインデントされています。XML ファイルではインデントしなくともかまいません。

<web-app> 要素自体の本文には、WebLogic Server 上で Web アプリケーションが動作する方法を決定する追加要素が入っています。ファイル内のタグ要素の順序は、このドキュメントに示されている順序に従っていなければなりません。この順序は、ドキュメント タイプ定義（DTD）ファイルで定義されます。

手順 4 : デプロイメント時の属性の定義

これらのタグは、デプロイメント ツールまたはアプリケーション サーバのリソース管理ツールの情報を提供します。このリリースでは、これらの値は WebLogic Server で使用されません。

<icon>	（省略可能）
<small-icon> iconfile.gif(jpg) **</small-icon>**	（省略可能）
<large-icon> iconfile.gif(jpg) **</large-icon>**	（省略可能）
</icon>	
<display-name> application-name **</display-name>**	（省略可能）
<description> descriptive-text **</description>**	（省略可能）
<distributable/>	（省略可能）

手順 5 : コンテキスト パラメータの定義

context-param 要素では、Web アプリケーションのサーブレット コンテキストの初期化パラメータを宣言します。これらのパラメータはユーザが定義するものであり、Web アプリケーション全体で使用可能となります。<param-name> 要素と <param-value> 要素を使用して、各 context-param を 1 つの context-param 要素内に設定します。コードでは、javax.servlet.ServletContext.getInitParameter() メソッドおよび javax.servlet.ServletContext.getInitParameterNames() メソッドを使用して、これらのパラメータにアクセスできます。

weblogic.http.clientCertProxy コンテキスト パラメータは、WL-Proxy-Client-Cert ヘッダ内のクライアント証明書を信頼することを指定します。WebLogic Server のこれまでのリリースでは、WL-Proxy-Client-Cert ヘッダのクライアント証明書はデフォルトで信頼されていました。

<context-param>	詳細については、A-5 ページの「context-param 要素」を参照。
<param-name> user-defined param name **</param-name>**	（必須）
<param-value> user-defined value **</param-value>**	（必須）
<description text description **</description>**	（省略可能）
</context-param>	

手順 6 : フィルタのコンフィグレーション（サーブレット 2.3 仕様のみ）

それぞれのフィルタには名前とフィルタ クラスがあります。フィルタの詳細については、「フィルタのコンフィグレーション」を参照してください。フィルタにも初期化パラメータを使用できます。次の要素はフィルタを定義するものです。

<filter>	詳細については、A-6 ページの「filter 要素」を参照してください。
<icon> **<small-icon>** iconfile **</small-icon>** **<large-icon>** iconfile **</large-icon>** **</icon>**	（省略可能）

<filter-name> Filter name **</filter-name>**	（必須）
<display-name> Filter Display Name **</display-name>**	（省略可能）
<description> ...text... **</description>**	（省略可能）
<filter-class> package.name.MyFilterClass **</filter-class>**	（必須）
<init-param>	（省略可能）
<param-name> name **</param-name>**	（必須）
<param-value> value **</param-value>**	（必須）
</init-param>	（省略可能）
</filter>	

手順 7 : フィルタ マッピングの定義（サーブレット 2.3 仕様のみ）

フィルタの宣言をした後で、各フィルタを URL パターンにマップします。

<filter-mapping>	詳細については、A-7 ページの「filter-mapping 要素」を参照。
<filter-name> name **</filter-name>**	（必須）
<url-pattern> pattern **</url-pattern>**	（必須）
</filter-mapping>	

手順 8 : アプリケーション リスナのコンフィグレーション（サーブレット 2.3 仕様のみ）

それぞれのリスナ クラスに対して、独立した <listener> 要素を使って、Web アプリケーション イベント リスナをコンフィグレーションします。

詳細については、次を参照してください。「アプリケーション イベントとリスナ」

<listener>	詳細については、A-8 ページの「listener 要素」を参照。
<listener-class> my.foo.listener **</listener-class>**	（必須）
</listener>	

手順 9 : サーブレットのデプロイ

サーブレットをデプロイするには、サーブレットに名前を付けて、その動作を実装するためのクラス ファイルまたは JSP を指定し、その他のサーブレット固有のプロパティを設定します。Web アプリケーション内の各サーブレットを <servlet>...</servlet> 要素内にリストします。すべてのサーブレットのエントリを作成したら、サーブレットを URL パターンにマッピングする要素を含める必要があります。これらのマッピング要素については、「手順 10:URL へのサーブレットのマッピング」で説明しています。

詳細については、「サーブレットのコンフィグレーション」を参照してください。

次の要素を使用して、サーブレットを宣言します。

<servlet>	詳細については、「servlet 要素」を参照。
<servlet-name> name **</servlet-name>**	（必須）

<servlet-class> package.name.MyClass **</servlet-class>** または **<jsp-file>** /foo/bar/myFile.jsp **</jsp-file>**	（必須）
<init-param>	（省略可能） 詳細については、「init-param 要素」を参照。
<param-name> name **</param-name>**	（必須）
<param-value> value **</param-value>**	（必須）
<description> ...text... **</description>** **</init-param>**	（省略可能）
<load-on-startup> loadOrder **</load-on-startup>**	（省略可能）
<security-role-ref>	（省略可能） 詳細については、「security-role-ref 要素」を参照してください。
<description> ...text... **</description>**	（省略可能）
<role-name> rolename **</role-name>**	（必須）

<role-link> rolelink **</role-link>**	（必須）
</security-role-ref>	
<small-icon> iconfile **</small-icon>**	（省略可能）
<large-icon> iconfile **</large-icon>**	（省略可能）
<display-name> Servlet Name **</display-name>**	（省略可能）
<description> ...text... **</description>**	（省略可能）
</servlet>	

初期化パラメータを含むサーブレット要素の例を次に示します。

```
<servlet>
   ...
   <init-param>
       <param-name>feedbackEmail</param-name>
       <param-value>feedback123@beasys.com</param-value>
       <description>
         The email for web-site feedback.
       </description>
   </init-param>
   ...
   </servlet>
```

手順 10:URL へのサーブレットのマッピング

<servlet> 要素を使用してサーブレットまたは JSP を宣言したら、それを 1 つまたは複数の URL パターンにマッピングして、パブリック HTTP リソースにします。URL パターンの用途は、Sun Microsystems のサーブレット 2.3 仕様で定義されています。それぞれをマッピングするには、<servlet-mapping> 要素を使用します。

<servlet-mapping>	詳細については、「servlet-mapping 要素」を参照。
<servlet-name> name **</servlet-name>**	（必須）
<servlet-name> pattern **</url-pattern>**	（必須）
</servlet-mapping>	

前述の <servlet> 宣言例の <servlet-mapping> の例を次に示します。

```
<servlet-mapping>
  <servlet-name>LoginServlet</servlet-name>
  <url-pattern>/login</url-pattern>
</servlet-mapping>
```

手順 11 : セッション タイムアウト値の定義

<session-config>	（省略可能）
<session-timeout> minutes **</session-timeout>**	詳細については、「session-config 要素」を参照。
</session-config>	

手順 12 : MIME マッピングの定義

MIME マッピングを作成するには、ファイル拡張子を MIME タイプにマップします。

<mime-mapping>	（省略可能） MIME タイプを定義する。 詳細については、「mime-mapping 要素」を参照。
<extension> ext **</extension>**	
<mime-type> mime type **</mime-type>**	
</mime-mapping>	

手順 13 : ウェルカム ページの定義

詳細については、「ウェルカム ページのコンフィグレーション」を参照してください。

<welcome-file-list>	（ウェルカム ページは省略可能） 詳細については、「welcome-file-list 要素」を参照。
<welcome-file> myWelcomeFile.jsp **</welcome-file>** **<welcome-file>** myWelcomeFile.html **</welcome-file>**	3-7 ページの「ウェルカム ページのコンフィグレーション」と、「WebLogic Server による HTTP リクエストの解決方法」も参照。
</welcome-file-list>	

手順 14 : エラー ページの定義

詳細については、「HTTP エラー応答のカスタマイズ」を参照してください。

<error-page>	（省略可能） エラーに応答するためのカスタマイズされたページを定義する。 詳細については、「error-page 要素」と「WebLogic Server による HTTP リクエストの解決方法」を参照。

```
    <error-code>
       HTTP error code
    </error-code>

    または

    <exception-type>
       Java exception class
    </exception-type>
    <location>
       URL
    </location>
</error-page>
```

手順 15 : JSP タグ ライブラリ記述子の定義

詳細については、「JSP タグ ライブラリのコンフィグレーション」を参照してください。

<taglib>	（省略可能） JSP タグ ライブラリを識別する。 詳細については、「taglib 要素」を参照。
<taglib-uri> string_pattern **</taglib-uri>**	（必須）
<taglib-location> filename **</taglib-location>**	（必須）
</taglib>	

JSP で使用する taglib ディレクティブの例を示します。

```
<%@ taglib uri="string_pattern" prefix="taglib" %>
```

詳細については、『JSP Tag Extentions プログラマーズ ガイド』を参照してください。

手順 16 : 外部リソースの参照

詳細については、「Web アプリケーションでの外部リソースのコンフィグレーション」を参照してください。

<resource-ref>	（省略可能） 詳細については、「resource-ref 要素」を参照。
<res-ref-name> name **</res-ref-name>**	（必須）
<res-type> Java class **</res-type>**	（必須）
<res-auth> CONTAINER | SERVLET **</res-auth>**	（必須）
<res-sharing-scope> Sharable | Unsharable **</res-sharing-scope>**	（省略可能）
</resource-ref>	（必須）

手順 17 : セキュリティ制約の設定

セキュリティを用いる Web アプリケーションでは、ユーザは、リソースにアクセスするためにログインする必要があります。ユーザの資格はセキュリティ レルムに照らして検証され、認可されると、ユーザは Web アプリケーション内の指定されたリソースにのみアクセスできるようになります。

Web アプリケーションのセキュリティは、3 つの要素を使用してコンフィグレーションします。

- `<login-config>` 要素では、ユーザにログインを求める方法とセキュリティ レルムの場所を指定します。この要素が指定されている場合、ユーザが Web アプリケーション内で定義されている `<security-constraint>` によって制約されたすべてのリソースにアクセスするには認証を受ける必要があります。

- <security-constraint> 要素では、URL マッピングを使用したリソースの集合へのアクセス特権を定義します。
- <security-role> 要素は、レルム内のグループまたはプリンシパルを表します。このセキュリティ ロール名は <security-constraint> 要素で使用され、<security-role-ref> 要素を介してサーブレットのコードで使用される代替ロール名にリンクされます。

詳細については、「Web アプリケーション リソースへのアクセスの制限」を参照してください。

<security-constraint>	（省略可能） 詳細については、「security-constraint 要素」を参照。
<web-resource-collection>	（必須） 詳細については、「web-resource-collection 要素」を参照。
<web-resource-name> name **</web-resource-name>**	（必須）
<description> ...text... **</description>**	（省略可能）
<url-pattern> pattern **</url-pattern>**	（省略可能）
<http-method> GET | POST **</http-method>**	（省略可能）
</web-resource-collection>	
<auth-constraint>	（省略可能） 詳細については、「auth-constraint 要素」を参照。

<role-name> group | principal **</role-name>**	（省略可能）
</auth-constraint>	
<user-data-constraint>	（省略可能） 詳細については、「user-data-constraint 要素」を参照。
<description> ...text... **</description>**	（省略可能）
<transport-guarantee> NONE|INTEGRAL|CONFIDENTIAL **</transport-guarantee>**	（必須）
</user-data-constraint>	
</security-constraint>	

手順 18：ログイン認証の設定

詳細については、「Web アプリケーション用の認証の設定」を参照してください。

<login-config>	（省略可能） 詳細については、「login-config 要素」を参照。
<auth-method> BASIC|FORM|CLIENT-CERT **</auth-method>**	（省略可能） ユーザの認証に使用する方法を指定する。
<realm-name> realmname **</realm-name>**	（省略可能） 詳細については、「セキュリティ レルムの指定」を参照。

<form-login-config>	（省略可能） 詳細については、「form-login-config 要素」を参照。 <auth-method> を FORM にコンフィグレーションする場合に、この要素を使用。
<form-login-page> URI **</form-login-page>**	（必須）
<form-error-page> URI **</form-error-page>**	（必須）
</form-login-config>	
</login-config>	

手順 19 : セキュリティ ロールの定義

詳細については、「Web アプリケーションでのセキュリティのコンフィグレーション」を参照してください。

<security-role>	（省略可能） 詳細については、「security-role 要素」を参照。
<description> ...text... **</description>**	（省略可能）
<role-name> rolename **</role-name>**	（必須）
</security-role>	

手順 20 : 環境エントリの設定

詳細については、「Web アプリケーションでの外部リソースのコンフィグレーション」を参照してください。

<env-entry>	（省略可能） 詳細については、「env-entry 要素」を参照。
<description> ...text... **</description>**	（省略可能）
<env-entry-name> name **</env-entry-name>**	（必須）
<env-entry-value> value **</env-entry-value>**	（必須）
<env-entry-type> type **</env-entry-type>**	（必須）
</env-entry>	

手順 21 : エンタープライズ JavaBean（EJB）リソースの参照

詳細については、「Web アプリケーションでの EJB の参照」を参照してください。

<ejb-ref>	（省略可能） 詳細については、「ejb-ref 要素」を参照。
<description> ...text... **</description>**	（省略可能）
<ejb-ref-name> name **</ejb-ref-name>**	（必須）

<ejb-ref-type> Java type **</ejb-ref-type>**	（必須）
<home> mycom.ejb.AccountHome **</home>**	（必須）
<remote> mycom.ejb.Account **</remote>**	（必須）
<ejb-link> ejb.name **</ejb-link>**	（省略可能）
<run-as> security role **</run-as>**	（省略可能）
</ejb-ref>	（必須）

web.xml のサンプル

コード リスト 8-1 サーブレット マッピング、ウェルカム ファイル、エラー ページを使用した web.xml ファイルのサンプル

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//
DTD Web Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

<!-- 次のサーブレット要素は、servletA と呼ばれるサーブレットを定義する。
 このサーブレットの Java クラスは servlets.servletA>
  <servlet>
    <servlet-name>servletA</servlet-name>
    <servlet-class>servlets.servletA</servlet-class>
  </servlet>

<!-- 次のサーブレット要素は、servletB と呼ばれるサーブレットを定義する。
このサーブレットの Java クラスは servlets.servletB>
  <servlet>
    <servlet-name>servletB</servlet-name>
    <servlet-class>servlets.servletB</servlet-class>
  </servlet>
```

```
<!-- 次のサーブレットマッピングは、servletA と呼ばれるサーブレット
(サーブレット要素を参照）を「blue」の URL パターンにマップする。
 この URL パターンは、このサーブレットを要求しているときに使用される。
 例 : http://host:port/myWebApp/blue -->
  <servlet-mapping>
    <servlet-name>servletA</servlet-name>
    <url-pattern>blue</url-pattern>
  </servlet-mapping>

<!-- 次のサーブレットマッピングは、servletB と呼ばれるサーブレット
(サーブレット要素を参照）を「yellow」の URL パターンにマップする。
 この URL パターンは、このサーブレットを要求しているときに使用される。
 例 : http://host:port/myWebApp/yellow -->
  <servlet-mapping>
    <servlet-name>servletB</servlet-name>
    <url-pattern>yellow</url-pattern>
  </servlet-mapping>

<!-- 次の welcome-file-list で welcome-file を指定する。
 ウェルカム ファイルについてはこのマニュアルの別の場所で説明 -->
  <welcome-file-list>
    <welcome-file>hello.html</welcome-file>
  </welcome-file-list>

<!-- 次の error-page 要素で、標準の
 HTTP エラー応答ページ（この場合は HTTP エラー 404）に代わる
 ページを指定する -->
  <error-page>
    <error-code>404</error-code>
    <location>/error.jsp</location>
  </error-page>

</web-app>
```

WebLogic 固有のデプロイメント記述子（weblogic.xml）の記述

weblogic.xml ファイルには、Web アプリケーション用の WebLogic 固有の属性が入っています。このファイルでは、HTTP セッション パラメータ、HTTP クッキー パラメータ、JSP パラメータ、リソース参照、セキュリティ ロール割り当て、文字セット マッピング、およびコンテナ属性を定義します。

DataSource、EJB、セキュリティ レルムなどの外部リソースを web.xml デプロイメント記述子に定義する場合は、任意の記述名を使用してリソースを定義できます。リソースにアクセスするには、weblogic.xml ファイルを使用して、このリソース名を JNDI ツリーの実際のリソース名にマッピングします。このファイルは、Web アプリケーションの WEB-INF ディレクトリに入れます。

WebLogic Server のサンプルおよび例をインストールした場合は、Pet Store サンプルの web.xml および weblogic.xml ファイルで、Web アプリケーションのデプロイメント記述子の実際の例を参照できます。これらのファイルは、WebLogic Server の配布ディレクトリ、\samples\PetStore\source\dd\war\WEB-INF にあります。

weblogic.xml ファイル内のタグ要素の順序は、このドキュメントに示されている順序に従っていなければなりません。

weblogic.xml ファイル作成の主な手順

weblogic.xml ファイルの詳しい作成手順

手順 1 : weblogic.xml ファイルの DOCTYPE ヘッダからの開始

このヘッダは、デプロイメント記述子の DTD ファイルの場所とバージョンを指しています。このヘッダは外部 URL の www.beasys.com を参照していますが、WebLogic Server には独自の DTD ファイルが用意されているので、ホスト サーバがインターネットにアクセスする必要はありません。ただし、この DOCTYPE 要素を web.xml ファイルに入れて、外部 URL を参照するようにしなければなりません。この要素内の DTD バージョンはこのデプロイメント記述子のバージョンを識別するためのものだからです。

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA
 Systems, Inc.//DTD Web Application 6.0//EN"
 "http://www.bea.com/servers/wls610/dtd/
weblogic-web-jar.dtd">
<weblogic-web-app>

     <description>
        Text description of the Web App
     </description>
     <weblogic-version>

     </weblogic-version>
```

この要素は、WebLogic Server では使用されない。

手順 2 : セキュリティ レルムへのセキュリティ ロール名のマッピング

<security-role-assignment>	
<role-name> name **</role-name>**	（必須） 詳細については、「security-role-assignment 要素」を参照。
<principal-name> name **</principal-name>**	（必須）
</security-role-assignment>	

複数のロールを定義する必要がある場合は、<role-name> タグおよび <principal-name> タグの対を別々の <security-role-assignment> 要素内に追加して定義します。

手順 3 : リソースの JNDI へのマッピング

この手順では、Web アプリケーションで使用するリソースを JNDI ツリーにマッピングします。web.xml デプロイメント記述子に <ejb-ref-name> または <res-ref-name> を定義する場合は、これらの名前を weblogic.xml でも参照し、WebLogic Server で使用可能な実際の JNDI 名をマッピングします。次の例では、データ ソースは myDataSource という名前のサーブレットで参照され、続いて web.xml で定義されているデータ型で参照されています。最後に、weblogic.xml ファイルで、myDataSource は JNDI ツリー内で使用可能な JNDI 名の accountDataSource にマッピングされています。JNDI 名は、JNDI ツリー内にバインドされているオブジェクトの名前と一致している必要があります。オブジェクトの JNDI ツリーへのバインドは、プログラムで行うことも、Administration Console でコンフィグレーションすることも可能です。詳細については、『WebLogic JNDI プログラマーズ ガイド』を参照してください。

サーブレットのコード :

```
javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup
                          ("myDataSource");
```

web.xml のエントリ :

```
<resource-ref>
. . .
   <res-ref-name>myDataSource</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>CONTAINER</res-auth>
. . .
</resource-ref>
```

weblogic.xml のエントリ：

```
<resource-description>
   <res-ref-name>myDataSource</res-ref-name>
   <jndi-name>accountDataSource</jndi-name>
</security-role-ref>
```

EJB も同様のパターンで JNDI ツリーにマッピングしますが、<resource-description> 要素の <res-ref-name> 要素の代わりに <ejb-reference-description> 要素の <ejb-ref-name> 要素を使用します。

<reference-descriptor>	詳細については、「reference-descriptor 要素」を参照。
<resource-description>	詳細については、「resource-description 要素」を参照。
<res-ref-name> name **</res-ref-name>**	（必須）
<jndi-name> JNDI name of resource **</jndi-name>**	（必須）
</resource-description>	
<ejb-reference-description>	
<ejb-ref-name> name **</ejb-ref-name>**	（必須） 詳細については、「ejb-reference-description 要素」を参照。

```
        <jndi-name>                                          （必須）
            JNDI name of EJB
        </jndi-name>
    </ejb-reference-
      description>
</reference-descriptor>
```

手順 4 : セッション パラメータの定義

Web アプリケーションの HTTP セッション パラメータを <session-param> タグ内に定義します。このタグは <session-descriptor> タグ内でネストします。各 <session-param> には、定義するパラメータの名前となる <param-name>...</param-name> 要素と、パラメータの値を提供する <param-value>...</param-value> 要素を指定する必要があります。HTTP セッション パラメータの一覧と設定方法の詳細については、「session-descriptor 要素」を参照してください。

```
<session-descriptor>                                 詳細については、
                                                     「session-descriptor
                                                     要素」を参照。
    <session-param>

        <param-name>
           session param name
        </param-name>

        <param-value>
            my value
        </param-value>

    </session-param>

</session-descriptor>
```

手順 5 : JSP パラメータの定義

Web アプリケーションの JSP コンフィグレーション パラメータを <jsp-param> タグ内に定義します。このタグは <jsp-descriptor> タグ内でネストします。各 <jsp-param> には、定義するパラメータの名前となる <param-name>...</param-name> 要素と、パラメータの値を提供する

<param-value>...</param-value> 要素を指定する必要があります。JSP パラメータの一覧と設定方法の詳細については、「jsp-descriptor 要素」を参照してください。

```
<jsp-descriptor>                                    詳細については、「jsp-descriptor 要素」を参照。
     <jsp-param>
          <param-name>
            jsp param name
          </param-name>
          <param-value>
             my value
          </param-value>
     </jsp-param>
</jsp-descriptor>
```

手順 6 : コンテナ パラメータの定義

<container-descriptor> 要素に入力できる有効で省略可能な要素として、<check-auth-on-forward> 要素があります。

```
<container-descriptor>                              詳細については、「container-descriptor 要素」を参照。
     <check-auth-on-forward/>
     <redirect-with-absolute-url>
        true|false
     </redirect-with-absolute-url>
</container-descriptor>
```

手順 7 : 文字セット パラメータの定義

省略可能な <charset-params> 要素は、文字セット マッピングを定義するために使用します。

```
<charset-params>                                          詳細については、「charset-params 要素」を参照。
     <input-charset>
          <resource-path>
            path to match
          </resource-path>
          <java-charset-name>
             name of Java
             character set
          </java-charset-name>
     </input-charset>
     <charset-mapping>
          <iana-charset-name>
               name of IANA
               character set
          </iana-charset-name>
           <java-charset-name>
               name of Java
               character set
           </java-charset-name>
     </charset-mapping>
</charset-params>
```

手順 8 : 記述子ファイルの終了

次のタグを使用して記述子ファイルを閉じます。

```
</weblogic-web-app>
```

A web.xml デプロイメント記述子の要素

以下の節では、web.xml ファイルで定義されているデプロイメント記述子の要素について説明します。web.xml のルート要素は <web-app> です。次の要素が <web-app> 要素の内部に定義されています。

- A-4 ページの「icon 要素」
  ```
  <small-icon>
  <large-icon>
  ```
- A-4 ページの「display-name 要素」
- A-5 ページの「description 要素」
- A-5 ページの「distributable 要素」
- A-5 ページの「context-param 要素」
  ```
  <param-name>
  <param-value>
  <description>
  ```
- A-6 ページの「filter 要素」
  ```
  <icon>
  <filter-name>
  <display-name>
  <description>
  <filter-class>
  <init-param>
  ```
- A-7 ページの「filter-mapping 要素」
  ```
  <filter-name>
  <url-pattern>
  <servlet>
  ```
- A-8 ページの「listener 要素」
  ```
  <listener-class>
  ```
- A-8 ページの「servlet 要素」
  ```
  <icon>
  ```

```
        <small-icon>
        <large-icon>
    <servlet-name>
    <display-name>
    <description>
    <servlet-class>
    <jsp-file>
    <init-param>
        <param-name>
        <param-value>
        <description>
    <load-on-startup>
    <security-role- ref>
        <description>
        <role-name>
        <role-link>
```

- A-12 ページの「servlet-mapping 要素」

```
<servlet-name>
<url-pattern>
```

- A-14 ページの「session-config 要素」

```
<session-timeout>
```

- A-15 ページの「mime-mapping 要素」

```
<extension>
<mime-type>
```

- A-16 ページの「welcome-file-list 要素」

```
<welcome-file>
```

- A-17 ページの「error-page 要素」

```
<error-code>
<exception-type>
<location>
```

- A-18 ページの「taglib 要素」

```
<taglib-location>
<taglib-uri>
```

- A-19 ページの「resource-ref 要素」

```
<description>
<res-ref-name>
<res-type>
<res-auth>
<res-sharing-scope>
```

- A-20 ページの「security-constraint 要素」

```
<web-resource- collection>
       <web-resource- name>
       <description>
       <url-pattern>
       <http-method>
<auth-constraint>
       <description>
       <role-name>
<user-data- constraint>
       <description>
       <transport- guarantee>
```

- A-24 ページの「login-config 要素」

```
<auth-method>
<realm-name>
<form-login- config>
       <form-login-page>
       <form-error-page>
```

- A-26 ページの「security-role 要素」

```
<description>
<role-name>
```

- A-27 ページの「env-entry 要素」

```
<description>
<env-entry-name>
<env-entry-value>
<env-entry-type>
```

- A-28 ページの「ejb-ref 要素」

```
<description>
<ejb-ref-name>
<ejb-ref-type>
<home>
<remote>
<ejb-link>
<run-as>
```

icon 要素

icon 要素では、GUI ツールで Web アプリケーションを表示する場合に使用される画像（小さいアイコンと大きいアイコン）の、Web アプリケーション内での位置を指定します（servlet 要素にも icon という要素があり、GUI ツール内にアイコンを提供してサーブレットを表すために使用されます）。

この要素は現在、WebLogic Server では使用されていません。

次の表では、icon 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<small-icon>	省略可能	GUI ツールで Web アプリケーションを表す小さい（16x16 ピクセル）.gif 画像または .jpg 画像の位置。この要素は現在、WebLogic Server では使用されていない。
<large-icon>	省略可能	GUI ツールで Web アプリケーションを表す大きい（32x32 ピクセル）.gif 画像または .jpg 画像の位置。この要素は現在、WebLogic Server では使用されていない。

display-name 要素

省略可能な display-name 要素では、Web アプリケーションの表示名（GUI ツールで表示できる短い名前）を指定します。

要素	必須 / 省略可能	説明
<display-name>	省略可能	この要素は現在、WebLogic Server では使用されていない。

description 要素

省略可能な description 要素では、Web アプリケーションに関する説明文を示します。

要素	必須 / 省略可能	説明
<description>	省略可能	この要素は現在、WebLogic Server では使用されていない。

distributable 要素

distributable 要素は、WebLogic Server では使用されていません。

要素	必須 / 省略可能	説明
<distributable>	省略可能	この要素は現在、WebLogic Server では使用されていない。

context-param 要素

省略可能な context-param 要素では、Web アプリケーションのサーブレット コンテキストの初期化パラメータを宣言します。<param-name> 要素と <param-value> 要素を使用して、各コンテキスト パラメータを 1 つの context-param 要素内に設定します。コードでは、javax.servlet.ServletContext.getInitParameter() メソッドおよび javax.servlet.ServletContext.getInitParameterNames() メソッドを使用して、これらのパラメータにアクセスできます。

次の表では、context-param 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<param-name>	必須	パラメータの名前。
<param-value>	必須	パラメータの値。
<description>	省略可能	パラメータの説明文。

filter 要素

filter 要素は、フィルタ クラスとその初期化パラメータを定義します。フィルタの詳細については、7-4 ページの「フィルタのコンフィグレーション」を参照してください。

次の表では、servlet 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<icon>	省略可能	GUI ツールでフィルタを表示する場合に使用される画像（小さいアイコンと大きいアイコン）の、Web アプリケーション内での位置を指定する。small-icon 要素と large-icon 要素がある。 この要素は現在、WebLogic Server では使用されていない。
<filter-name>	必須	フィルタの名前を定義する。この名前は、デプロイメント記述子内のほかの場所でそのフィルタ定義を参照する場合に使用される。
<display-name>	省略可能	GUI ツールによって表示されることを想定した短い名前。
<description>	省略可能	フィルタの説明文。
<filter-class>	必須	フィルタの完全修飾クラス名。

要素	必須 / 省略可能	説明
<init-param>	省略可能	フィルタの初期化パラメータの名前と値の組み合わせを指定する。 パラメータごとに <init-param> タグの別個のセットを使用する。

filter-mapping 要素

次の表では、filter-mapping 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<filter-name>	必須	URL パターンまたはサーブレットのマッピング先のフィルタの名前。この名前は、<filter-name> 要素で <filter> 要素に割り当てられている名前に対応する。
<url-pattern>	必須 - または <servlet> によってマップされる	URL を解決する場合に使用されるパターンを記述する。http://host:port（「host」はホスト名、「port」はポート番号）+ ContextPath に続く URL の部分は、WebLogic Server によって <url-pattern> と比較される。パターンが一致すれば、この要素でマップされているフィルタが呼び出される。 サンプル パターンを以下に示す。 /soda/grape/* /foo/* /contents *.foo URL は、サーブレット仕様 2.2 の第 10 節で指定されているルールに準拠している必要がある。
<servlet>	必須 - または <url-pattern> によってマップされる	呼び出された場合に、このフィルタを実行するサーブレットの名前。

listener 要素

listener 要素を使うアプリケーション リスナを定義します。

要素	必須 / 省略可能	説明
<listener-class>	省略可能	Web アプリケーション イベントに応答するクラスの名前。

詳細については、6-5 ページの「イベント リスナのコンフィグレーション」を参照してください。

servlet 要素

servlet 要素では、サーブレットの宣言的なデータを指定します。

jsp-file 要素および <load-on-startup> 要素が指定されている場合、その JSP は、WebLogic Server の起動時にあらかじめコンパイルされ、ロードされます。

次の表では、servlet 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<icon>	省略可能	GUI ツールでサーブレットを表示する場合に使用される画像（小さいアイコンと大きいアイコン）の、Web アプリケーション内での位置。small-icon 要素と large-icon 要素がある。 この要素は現在、WebLogic Server では使用されていない。
<servlet-name>	必須	サーブレットの標準名を定義する。この名前は、デプロイメント記述子内の他の場所でそのサーブレット定義を参照する場合に使用される。
<display-name>	省略可能	GUI ツールによって表示されることを想定した短い名前。

要素	必須 / 省略可能	説明
<description>	省略可能	サーブレットの説明文。
<servlet-class>	必須（または <jsp-file> を使用）	サーブレットの完全修飾クラス名。 servlet の本体では、<servlet-class> タグまたは <jsp-file> タグのいずれか一方のみを使用する。
<jsp-file>	必須（または <servlet-class> を使用）	Web アプリケーションのルート ディレクトリを基準にした、Web アプリケーション内の JSP ファイルへの絶対パス。 servlet の本体では、<servlet-class> タグまたは <jsp-file> タグのいずれか一方のみを使用する。
<init-param>	省略可能	サーブレットの初期化パラメータの名前と値の組み合わせを指定する。 パラメータごとに <init-param> タグの別個のセットを使用する。
<load-on-startup>	必須	この要素が指定されたサーブレットは、WebLogic Server の起動時に WebLogic Server によって初期化される。この要素のコンテンツは、サーブレットがロードされる順序を示す正の整数である。整数の小さい方から順にロードされる。値の指定がない、または値が正の整数でない場合は、WebLogic Server によって、起動シーケンスにある任意の順序でサーブレットがロードされる。
<security-role-ref>	省略可能	<security-role> で定義されたセキュリティ ロール名を、サーブレットのロジックでハード コード化される代替ロール名にリンクする場合に使用される。この特別な抽象化レイヤによって、サーブレット コードを変更しなくてもデプロイメント時にサーブレットをコンフィグレーションできるようになる。

icon 要素

これは、A-8 ページの「servlet 要素」内の要素です。

icon 要素では、GUI ツールでサーブレットを表示する場合に使用される画像（小さいアイコンと大きいアイコン）の、Web アプリケーション内での位置を指定します。

次の表では、icon 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<small-icon>	省略可能	GUI ツールでサーブレットを表示する場合に使用される小さい（16x16 ピクセル）.gif 画像または .jpg 画像の Web アプリケーション内での位置を指定する。 この要素は現在、WebLogic Server では使用されていない。
<large-icon>	省略可能	GUI ツールでサーブレットを表示する場合に使用される大きい（32x32 ピクセル）.gif 画像または .jpg 画像の Web アプリケーション内での位置を指定する。 この要素は現在、WebLogic Server では使用されていない。

init-param 要素

これは、A-8 ページの「servlet 要素」内の要素です。

省略可能な init-param 要素では、サーブレットの初期化パラメータの名前と値の組み合わせを指定します。パラメータごとに init-param タグの別個のセットを使用します。

javax.servlet.ServletConfig.getInitParameter() メソッドを使用して、これらのパラメータにアクセスできます。

次の表では、init-param 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<param-name>	必須	このパラメータの名前を定義する。
<param-value>	必須	このパラメータの String 値を定義する。
<description>	省略可能	初期化パラメータの説明文。

WebLogic Server では、使用可能な実行キューにサーブレットまたは JSP を割り当てる特別な初期化パラメータ、wl-dispatch-policy が認識されます。次の例では、CriticalWebApp という名前の実行キューの実行スレッドを使用するようにサーブレットを割り当てています。

```
<servlet>
   ...
   <init-param>
      <param-name>wl-dispatch-policy</param-name>
      <param-value>CriticalWebApp</param-value>
   </init-param>
</servlet>
```

CriticalWebApp キューが使用できない場合は、デフォルトの WebLogic Server 実行キュー内の使用可能な実行キューを使用します。WebLogic Server での実行キューのコンフィグレーションについては、「スレッド数の設定」を参照してください。キューの作成と使用については、「実行キューによるスレッド使用の制御」を参照してください。

security-role-ref 要素

これは、A-8 ページの「servlet 要素」内の要素です。

security-role-ref 要素は、<security-role> で定義されたセキュリティ ロール名を、サーブレットのロジックでハード コード化される代替ロール名にリンクします。この特別な抽象化レイヤによって、サーブレット コードを変更しなくてもデプロイメント時にサーブレットをコンフィグレーションできるようになります。

次の表では、security-role-ref 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<description>	省略可能	ロールの説明文。
<role-name>	必須	サーブレット コード内で使用されるセキュリティ ロールまたはプリンシパルの名前を定義する。
<role-link>	必須	後にデプロイメント記述子内の <security-role> 要素で定義されるセキュリティ ロールの名前を定義する。

servlet-mapping 要素

servlet-mapping 要素では、サーブレットと URL パターンの間のマッピングを定義します。

次の表では、servlet-mapping 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<servlet-name>	必須	URL パターンのマッピング先のサーブレットの名前。この名前は、<servlet> 宣言タグでサーブレットに割り当てた名前に対応する。

要素	必須 / 省略可能	説明
<url-pattern>	必須	URL を解決する場合に使用されるパターンを記述する。 http://host:port（「host」はホスト名、「port」はポート番号）+ WebAppName（Web アプリケーション名）に続く URL の部分は、WebLogic Server によって <url-pattern> と比較される。パターンが一致すれば、この要素でマップされているサーブレットが呼び出される。 サンプル パターンを以下に示す。 /soda/grape/* /foo/* /contents *.foo URL は、サーブレット仕様 2.2 の第 10 節で指定されているルールに準拠している必要がある。 サーブレットのマッピングのその他の例については、3-2 ページの「サーブレット マッピング」を参照。

session-config 要素

session-config 要素では、Web アプリケーションのセッションのパラメータを定義します。

次の表では、session-config 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<session-timeout>	省略可能	この Web アプリケーション内のセッションが期限切れになってからの分数。この要素で設定する値は、次に示す特殊な値のいずれか 1 つが入力されない限り、WebLogic 固有のデプロイメント記述子である weblogic.xml の <session-descriptor> 要素の TimeoutSecs パラメータに設定された値をオーバーライドする。 デフォルト値 : -2 最大値 : Integer.MAX_VALUE ÷ 60 特殊な値 : ■ -2 = weblogic.xml の <session-descriptor> 要素にある TimeoutSecs によって設定された値を使用する。 ■ -1 = セッションはタイムアウトしない。weblogic.xml の <session-descriptor> 要素に設定された値は無視される。 詳細については、B-4 ページの「session-descriptor 要素」を参照。

mime-mapping 要素

mime-mapping 要素では、拡張子と MIME タイプの間のマッピングを定義します。

次の表では、mime-mapping 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<extension>	必須	拡張子を記述する文字列（例 : txt）。

<mime-type>	必須	定義されている MIME タイプを記述する文字列 (例 : text/plain)。WebLogic Server では、以下のデフォルト MIME タイプが提供されている (これらはオーバーライドできる)。
		`setIfNone(mimeTypesMap, "html", "text/html");`
		`setIfNone(mimeTypesMap, "htm", "text/html");`
		`setIfNone(mimeTypesMap, "gif", "image/gif");`
		`setIfNone(mimeTypesMap, "jpeg", "image/jpeg");`
		`setIfNone(mimeTypesMap, "jpg", "image/jpeg");`
		`setIfNone(mimeTypesMap, "pdf", "application/pdf");`
		`setIfNone(mimeTypesMap, "zip", "application/zip");`
		`setIfNone(mimeTypesMap, "class", "application/x-java-vm");`
		`setIfNone(mimeTypesMap, "jar", "application/x-java-archive");`
		`setIfNone(mimeTypesMap, "ser", "application/x-java-serialized-object");`
		`setIfNone(mimeTypesMap, "exe", "application/octet-stream");`
		`setIfNone(mimeTypesMap, "txt", "text/plain");`
		`setIfNone(mimeTypesMap, "java", "text/plain");`
		`// これは、インストール時のままの状態で JavaWebStart が動作するための`
		`// ものである`
		`setIfNone(mimeTypesMap, "jnlp", "application/x-java-jnlp-file");`

welcome-file-list 要素

省略可能な welcome-file-list 要素では、welcome-file 要素の順序付きリストを指定します。

URL 要求がディレクトリ名の場合、この要素で指定された最初のファイルが WebLogic Server によって返されます。そのファイルが見つからない場合は、WebLogic Server によってリスト内の次のファイルが返されます。

詳細については、3-7 ページの「ウェルカム ページのコンフィグレーション」と「WebLogic Server による HTTP リクエストの解決方法」を参照してください。

次の表では、welcome-file-list 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<welcome-file>	省略可能	デフォルトのウェルカム ファイルとして使用するファイル名（例 : index.html）。

error-page 要素

省略可能な error-page 要素では、エラーコードや例外のタイプと Web アプリケーションにあるリソースのパスの間のマッピングを指定します。

WebLogic Server が HTTP リクエストに応答しているときにエラーが発生した場合や、Java 例外の結果としてエラーが発生した場合は、WebLogic Server によって HTTP エラー コードまたは Java エラー メッセージのいずれかを表示する HTML ページが返されます。独自の HTML ページを定義して、これらのデフォルトのエラー ページの代わりとして、または Java 例外の応答ページとして表示することができます。

詳細については、3-10 ページの「HTTP エラー応答のカスタマイズ」と「WebLogic Server による HTTP リクエストの解決方法」を参照してください。

次の表では、error-page 要素内で定義できる要素について説明します。

注意： <error-code> と <exception-type> のどちらかを定義します。両方は定義しないでください。

要素	必須 / 省略可能	説明
<error-code>	省略可能	有効な HTTP エラー コード（例：404）。
<exception-type>	省略可能	Java 例外の完全修飾クラス名（例：java.lang.string）。
<location>	必須	エラーに応答して表示されるリソースの位置（例：/myErrorPg.html）。

taglib 要素

省略可能な taglib 要素では、JSP のタグ ライブラリを指定します。

JSP タグ ライブラリ記述子（TLD）の位置を URI パターンに関連付けます。TLD は、WEB-INF ディレクトリを基準にした相対位置にある JSP 内に指定できますが、Web アプリケーションをデプロイするときに、<taglib> タグを使用して TLD をコンフィグレーションすることもできます。TLD ごとに別個の要素を使用します。

次の表では、taglib 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<taglib-location>	必須	Web アプリケーションのルートを基準にしたタグ ライブラリ記述子の相対ファイル名を指定する。タグ ライブラリ記述子ファイルを WEB-INF ディレクトリの下に格納して、HTTP リクエストを通じて外部から入手できないようにしたほうがよい。
<taglib-uri>	必須	web.xml ドキュメントの位置を基準にした相対位置にある URI を指定する。この URI によって、Web アプリケーションで使用されるタグ ライブラリが識別される。 URI が JSP ページの taglib ディレクティブで使用されている URI 文字列と一致する場合、このタグ ライブラリが使用される。

resource-ref 要素

省略可能な resource-ref 要素では、外部リソースへの参照ルックアップ名を定義します。この定義により、サーブレット コードは、デプロイメント時に実際の位置にマップされる「仮想的な」名前でリソースをルックアップできるようになります。

別個の <resource-ref> 要素を使用して、各外部リソース名を定義します。外部リソース名は、デプロイメント時に WebLogic 固有のデプロイメント記述子 weblogic.xml でリソースの実際の位置名にマップされます。

次の表では、resource-ref 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<description>	省略可能	説明文。
<res-ref-name>	必須	JNDI ツリー内で使用されるリソースの名前。Web アプリケーション内のサーブレットはこの名前を使用して、リソースへの参照をルックアップする。
<res-type>	必須	参照名に対応するリソースの Java クラスのタイプ。Java の完全パッケージ名を使用する。
<res-auth>	必須	セキュリティのためのリソース サインオンの指定に使用される。 APPLICATION を指定した場合、アプリケーション コンポーネント コードによってプログラムでリソース サインオンが行われる。CONTAINER を指定した場合、WebLogic Server では、login-config 要素で定義されたセキュリティ コンテキストが使用される。A-24 ページの「login-config 要素」を参照。
<res-sharing-scope>	省略可能	指定されたリソース マネージャ接続ファクトリ参照を経由して取得された接続を共有するかどうかを指定する。 有効な値 : ■ Shareable ■ Unshareable

security-constraint 要素

security-constraint 要素では、<web-resource-collection> 要素で定義されたリソースの集合へのアクセス特権を定義します。

詳細については、5-1 ページの「Web アプリケーションでのセキュリティのコンフィグレーション」を参照してください。

次の表では、security-constraint 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<web-resource-collection>	必須	このセキュリティ制約が適用される Web アプリケーションのコンポーネントを定義する。
<auth-constraint>	省略可能	このセキュリティ制約で定義される Web リソースの集合にアクセスするグループまたはプリンシパルを定義する。A-22 ページの「auth-constraint 要素」も参照。
<user-data-constraint>	省略可能	クライアントによるサーバとの通信方法を定義する。 A-23 ページの「user-data-constraint 要素」も参照。

web-resource-collection 要素

各 <security-constraint> 要素には、1 つまたは複数の <web-resource-collection> 要素が必要になります。これらの要素では、このセキュリティ制約が適用される Web アプリケーションの領域を定義します。

これは、A-20 ページの「security-constraint 要素」内の要素です。

次の表では、web-resource-collection 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<web-resource-name>	必須	Web リソースの集合の名前。
<description>	省略可能	セキュリティ制約の説明文。
<url-pattern>	省略可能	<url-pattern> 要素を 1 つまたは複数使用して、このセキュリティ制約の適用先となる URL パターンを宣言する。この要素を 1 つも使用しない場合、この <web-resource-collection> は WebLogic Server から無視される。
<http-method>	省略可能	<http-method> 要素を 1 つまたは複数使用して、認可制約の対象になる HTTP メソッド（通常は GET または POST）を宣言する。<http-method> 要素を省略した場合には、デフォルトの動作として、セキュリティ制約がすべての HTTP メソッドに適用される。

auth-constraint 要素

これは、A-20 ページの「security-constraint 要素」内の要素です。

省略可能な auth-constraint 要素では、このセキュリティ制約で定義された Web リソースの集合にアクセスするグループまたはプリンシパルを定義します。

次の表では、auth-constraint 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<description>	省略可能	セキュリティ制約の説明文。
<role-name>	省略可能	このセキュリティ制約で定義されたリソースにアクセスできるセキュリティ ロールを定義する。セキュリティ ロール名は、security-role-ref 要素を使用してプリンシパルにマップされる。A-11 ページの「security-role-ref 要素」を参照。

user-data-constraint 要素

これは、A-20 ページの「security-constraint 要素」内の要素です。

user-data-constraint 要素では、クライアントによるサーバとの通信方法を定義します。

次の表では、user-data-constraint 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<description>	省略可能	説明文。
<transport-guarantee>	必須	クライアントとサーバの間の通信方法を指定する。 INTEGRAL または CONFIDENTIAL の転送保証を使用してユーザが認証を受けた場合、WebLogic Server はセキュア ソケット レイヤ（SSL）接続を確立する。 指定できる値 : ■ NONE — 転送の保証が不要な場合に指定する。 ■ INTEGRAL — クライアントとサーバの間で、転送中にデータが変更されない方法でデータを転送する必要がある場合に指定する。 ■ CONFIDENTIAL — 転送中にデータの中味を覗かれないようにデータを転送する必要がある場合に指定する。

login-config 要素

省略可能な login-config 要素を使って、ユーザの認証方法、このアプリケーションで使用されるレルムの名前、およびフォームによるログイン機能で必要になる属性をコンフィグレーションします。

この要素が指定されている場合、ユーザが Web アプリケーション内で定義されている <security-constraint> によって制約されたすべてのリソースにアクセスするには認証を受ける必要があります。認証されると、ユーザは、ほかのリソースにアクセスする権限が与えられる場合もあります。

次の表では、login-config 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<auth-method>	省略可能	ユーザを認証する場合に使用される方式を指定する。指定できる値は次のとおり。 BASIC - ブラウザの認証を使用する。 FORM - ユーザによって記述された HTML フォームを使用する。 CLIENT-CERT
<realm-name>	省略可能	ユーザの資格を認証する場合に参照されるレルムの名前。省略した場合、Administration Console の [Web アプリケーション	コンフィグレーション	その他] タブにある [認証レルム名] フィールドで定義されたレルムがデフォルトで使用される。詳細については、「セキュリティ レルムの指定」を参照。 **注意 :** <realm-name> 要素は、WebLogic Server 内のセキュリティ レルムを参照しない。この要素では、HTTP 基本認証で使用するレルム名が定義されている。 **注意 :** システム セキュリティ レルムは、サーバで特定の操作が実行されるときにチェックされるセキュリティ情報の集合である。サーブレット セキュリティ レルムは、これとは別のセキュリティ情報の集合であり、ページがアクセスされて基本認証が使われる際にチェックされる。

要素	必須 / 省略可能	説明
<form-login-config>	省略可能	<auth-method> を FORM にコンフィグレーションする場合に、この要素を使用する。A-26 ページの「form-login-config 要素」を参照。

form-login-config 要素

これは、A-24 ページの「login-config 要素」内の要素です。

<auth-method> を FORM にコンフィグレーションする場合に、<form-login-config> 要素を使用します。

要素	必須 / 省略可能	説明
<form-login-page>	必須	ユーザを認証する場合に使用される、ドキュメント ルートを基準にした Web リソースの相対的な URI。これは、HTML ページ、JSP、または HTTP サーブレットのいずれかになり、特定の命名規約に従うフォームを表示する HTML ページを返す。詳細については、5-2 ページの「Web アプリケーション用の認証の設定」を参照。
<form-error-page>	必須	失敗した認証ログインに応答してユーザに送信される、ドキュメント ルートを基準にした Web リソースの相対的な URI。

security-role 要素

次の表では、security-role 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<description>	省略可能	セキュリティ ロールの説明文。
<role-name>	必須	ロール名。ここで使用する名前は、WebLogic 固有のデプロイメント記述子 weblogic.xml で対応するエントリが必要になる。weblogic.xml によって、ロールはセキュリティ レルムにあるプリンシパルにマップされる。詳細については、B-2 ページの「security-role-assignment 要素」を参照。

env-entry 要素

省略可能な env-entry 要素では、アプリケーションの環境エントリを宣言します。環境エントリごとに別個の要素を使用します。

次の表では、env-entry 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<description>	省略可能	説明文。
<env-entry-name>	必須	環境エントリの名前。
<env-entry-value>	必須	環境エントリの値。
<env-entry-type>	必須	環境エントリのタイプ。 次の Java クラスのタイプからいずれか 1 つを選択できる。 java.lang.Boolean java.lang.String java.lang.Integer java.lang.Double java.lang.Float

ejb-ref 要素

省略可能な ejb-ref 要素では、EJB リソースへの参照を定義します。この参照は、WebLogic 固有のデプロイメント記述子ファイル weblogic.xml でマッピングを定義することにより、デプロイメント時に EJB の実際の位置にマップされます。別個の <ejb-ref> 要素を使用して、各参照 EJB 名を定義します。

次の表では、ejb-ref 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<description>	省略可能	参照の説明文。
<ejb-ref-name>	必須	Web アプリケーションで使用される EJB の名前。この名前は、WebLogic 固有のデプロイメント記述子 weblogic.xml で JNDI ツリーにマップされる。詳細については、B-4 ページの「ejb-reference-description 要素」を参照。
<ejb-ref-type>	必須	参照 EJB の期待される Java クラスのタイプ。
<home>	必須	EJB ホーム インタフェースの完全修飾クラス名。
<remote>	必須	EJB リモート インタフェースの完全修飾クラス名。
<ejb-link>	省略可能	含まれている J2EE アプリケーション パッケージでの EJB の <ejb-name>。
<run-as>	省略可能	参照される EJB にセキュリティ コンテキストが適用されるセキュリティ ロール。<security-role> 要素で定義されたセキュリティ ロールである必要がある。

B weblogic.xml デプロイメント記述子の要素

この章では、weblogic.xml ファイルで定義するデプロイメント記述子の要素について説明します。weblogic.xml のルート要素は <weblogic-web-app> です。次の要素が <weblogic-web-app> 要素の内部に定義されています。

- B-2 ページの「description 要素」
- B-2 ページの「weblogic-version 要素」
- B-2 ページの「security-role-assignment 要素」
  - <role-name>
  - <principal-name>
- B-3 ページの「reference-descriptor 要素」
  - resource-description 要素
    - <res-ref-name>
    - <jndi-name>
  - ejb-reference-description 要素
    - <ejb-ref-name>
    - <jndi-name>
- B-4 ページの「session-descriptor 要素」
  - セッション パラメータの名前と値
- B-11 ページの「jsp-descriptor 要素」
  - JSP パラメータの名前と値
- B-14 ページの「container-descriptor 要素」
  - check-auth-on-forward 要素
- B-15 ページの「charset-params 要素」
  - input-charset 要素
    - <resource-path>
    - <java-charset-name>
  - charset-mapping 要素
    - <iana-charset-name>

```
<java-charset-name>
```

http://www.bea.com/servers/wls610/dtd/weblogic-web-jar.dtd で weblogic.xml でドキュメント タイプ定義（DTD) を参照することもできます。

description 要素

description 要素は、Web アプリケーションの説明文です。

weblogic-version 要素

weblogic-version 要素は、この Web アプリケーションをデプロイする WebLogic Server のバージョンを示します。この要素は参照用で、現在 WebLogic Server では使用されていません。

security-role-assignment 要素

security-role-assignment 要素は、次の例で示すように、レルム内のセキュリティ ロールと 1 つまたは複数のプリンシパルの間のマッピングを宣言します。

```
<security-role-assignment>
  <role-name>PayrollAdmin</role-name>
  <principal-name>Tanya</principal-name>
  <principal-name>Fred</principal-name>
  <principal-name>system</principal-name>
</security-role-assignment>
```

次の表では、security-role-assignment 要素内で定義できる要素について説明します。

要素	必須 省略可能	説明
<role-name>	必須	セキュリティ ロール名を指定する。
<principal-name>	必須	セキュリティ レルムで定義されるプリンシパルの名前を指定する。複数の <principal-name> 要素を使用してプリンシパルをロールにマップできる。セキュリティ レルムの詳細については、『WebLogic Security プログラマーズ ガイド』を参照。

reference-descriptor 要素

reference-descriptor 要素は、Web アプリケーションで使用される名前をサーバ リソースの JNDI 名にマップします。reference-description 要素には次の 2 つの要素が含まれています。resource-description 要素は DataSource などのリソースをその JNDI 名にマップします。ejb-reference 要素は、EJB を JNDI 名にマップします。

resource-description 要素

次の表では、resource-description 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<res-ref-name>	必須	リソース参照名を指定する。
<jndi-name>	必須	リソースの JNDI 名を指定する。

ejb-reference-description 要素

次の表では、ejb-reference-description 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<ejb-ref-name>	必須	Web アプリケーションで使用する EJB 参照の名前を指定する。
<jndi-name>	必須	参照の JNDI 名を指定する。

session-descriptor 要素

session-descriptor 要素は、次の例で示すように HTTP セッションのパラメータを定義します。

```
<session-descriptor>
  <session-param>
     <param-name>
       CookieDomain
     </param-name>
     <param-value>
       myCookieDomain
     </param-value>
  </session-param>
</session-descriptor>
```

セッション パラメータの名前と値

次の表では、session-param 要素内で定義できるセッション パラメータの名前と値について説明します。

パラメータ名	デフォルト値	パラメータ値
CookieDomain	Null	クッキーが有効になるドメインを指定する。たとえば、CookieDomain を .mydomain.com に設定すると、*.mydomain.com ドメイン内のサーバにクッキーが返される。 ドメイン名には少なくとも 2 つのコンポーネントが必要である。名前を *.com または *.net に設定すると無効になる。 このパラメータを設定しない場合、デフォルトは、クッキーを発行したサーバのドメイン。 詳細については、Sun Microsystems のサーブレット仕様にある Cookie.setDomain() を参照。
CookieComment	Weblogic Server Session Tracking Cookie	クッキー ファイル内のセッションをトラッキングするクッキーを識別するコメントを指定する。 このパラメータを設定しない場合、デフォルトは WebLogic Session Tracking Cookie。アプリケーションに対して、より詳細な名前を指定できる。

パラメータ名	デフォルト値	パラメータ値
CookieMaxAgeSecs	-1	セッション クッキーの有効期間を秒単位で設定する。時間が経過すると、クッキーはクライアントで期限切れになる。 値が 0 の場合、クッキーはすぐに期限切れになる。 最大値は Integer.MAX_VALUE で、クッキーは永久に期限切れにならない。 -1 に設定した場合、クッキーはユーザがブラウザを終了すると期限切れになる。 クッキーの詳細については、4-1 ページの「Web アプリケーションにおけるセッションとセッション永続性の使用」を参照。
CookieName	JSESSIONID	セッション クッキー名を定義する。設定しない場合、デフォルトは JSESSIONID。アプリケーションに対して、より詳細な名前を指定できる。ProxyByExtension を使用するときは、;jsessionid 識別子または ?jsessionid 識別子を使用して、書き直された URL を渡すことができる。
CookiePath	Null	ブラウザによるクッキーの送信先のパス名を指定する。 このパラメータを設定しない場合、デフォルトは / （スラッシュ）。デフォルト値では、ブラウザは、WebLogic Server で指定されているすべての URL にクッキーを送信する。マップ対象を絞り込んだパスを設定し、リクエスト URL を、ブラウザがクッキーを送信するものに限定できる。
CookiesEnabled	true	セッション クッキーの使用はデフォルトで有効になっているが（推奨）、このプロパティを false に設定して無効にすることも可能。URL 書き換えの使い方をテストするためにこのオプションをオフにする場合もある。

パラメータ名	デフォルト値	パラメータ値
CookieSecure	false	このパラメータを設定すると、クライアントのブラウザは HTTPS 接続を通してだけクッキーを返送する。これにより、クッキー ID が安全であること、および HTTPS だけを使用する Web サイトにおいてのみクッキー ID が使用されることが保証される。この機能を有効にすると、HTTP 経由でのセッション クッキーは機能しなくなり、クライアントが HTTPS ではない場所に送られた場合、セッションは送信されない。
InvalidationIntervalSecs	60	WebLogic Server が、タイムアウトの無効なセッションに対してハウスクリーニング チェックを実行してから古いセッションを削除してメモリを解放するまでの待ち時間を秒単位で設定する。このパラメータを使用すると、トラフィックの多いサイトで WebLogic Server の動作を最適化できる。 最小値は毎秒（1）。最大値は、週に 1 回（604800 秒）。このパラメータを設定しない場合、デフォルトは 60 秒。

パラメータ名	デフォルト値	パラメータ値
PersistentStoreDir	session_db	PersistentStoreType を file に設定した場合、ディレクトリ パスは、WebLogic Server がセッションを保存する場所に設定される。ディレクトリ パスには、temp ディレクトリの相対パスか絶対パスのどちらかを使用。temp ディレクトリは、Web アプリケーションの WEB-INF ディレクトリ下に生成されたディレクトリか、コンテキスト パラメータ javax.servlet.context.tmpdir で指定されたディレクトリ。 各セッションのサイズに有効なセッション数をかけたサイズを保存できるだけのディスク スペースを確保する必要がある。PersistentStoreDir に作成されたファイルを見ると、セッションのサイズがわかる。各セッションのサイズは、シリアライズされたセッション データの変更のサイズによって異なる。 このディレクトリを複数サーバ間での共有ディレクトリにすると、ファイル永続セッションをクラスタ対応にできる。 このディレクトリは手動で作成する必要がある。
PersistentStoreTable	wl_servlet_sessions	PersistentStoreType が jdbc に設定されているときにだけ適用される。デフォルト以外のデータベース テーブル名を選択するときに使用する。
PersistentStorePool	なし	永続ストレージに使用される JDBC 接続プールの名前を指定する。 データベース接続プールの設定の詳細については、「JDBC 接続の管理」を参照。

パラメータ名	デフォルト値	パラメータ値
PersistentStoreType	memory	永続ストレージの方法を次のいずれかに設定する。 ■ memory ― 永続セッション ストレージを無効にする。 ■ file ― ファイル ベースの永続性を使用する（上記の「PersistentStoreDir」を参照）。 ■ jdbc ― データベースを使用して永続セッションを保存する（上記の「PersistentStorePool」を参照）。 ■ replicated ― memory と同じだが、セッション データはクラスタ化されたサーバ間でレプリケートされる。 ■ cookie ― すべてのセッション データはユーザのブラウザ内のクッキーに格納される。 ■ replicated_if_clustered ― Web アプリケーションがクラスタ構成のサーバにデプロイされている場合は、有効になっている PersistentStoreType がレプリケートされる。クラスタでない場合は、memory がデフォルトである。
PersistentStoreCookieName	WLCOOKIE	クッキーベースの永続性に使用するクッキーの名前を設定する。詳細については、4-10 ページの「クッキーベースのセッション永続性の使用」を参照。

パラメータ名	デフォルト値	パラメータ値
IDLength	52	セッション ID のサイズを設定する。 最小値は 8 バイト、最大値は Integer.MAX_VALUE で指定した値。 WAP アプリケーションを作成する場合、WAP プロトコルはクッキーをサポートしていないため、URL を書き換える必要がある。また、一部の WAP デバイスでは、URL の長さに 128 文字（パラメータも含む）の制限がある。この制限によって、URL 書き換えで転送できるデータ サイズが限られる。パラメータ用の領域を確保するには、WebLogic Server でランダムに生成されるセッション ID のサイズをこのパラメータで制限する。
TimeoutSecs	3600	WebLogic Server がセッションをタイムアウトするまでの待ち時間を秒単位で設定する（秒数)。 最小値は 1、デフォルト値は 3600、最大値は MAX_VALUE で指定した整数値。 トラフィックの多いサイトでは、セッションのタイムアウトを調整すると、アプリケーションの動作を最適化できる。ブラウザ クライアントでいつでもセッションを終了できるようにする必要がある場合でも、ユーザがサイトを離れるか、ユーザのセッションがタイムアウトになれば、サーバに接続する必要はなくなる。 このパラメータは、web.xml の session-timeout 要素（分単位で定義）によってオーバーライドされる可能性がある。詳細については、A-14 ページの「session-config 要素」を参照。
JDBCConnectionTimeoutSecs	120	WebLogic Server が JDBC 接続をタイムアウトするまでの待ち時間を秒単位で設定する（秒数)。
URLRewritingEnabled	true	URL 書き換えを有効にする。これによって、セッション ID が URL にエンコーディングされ、クッキーがブラウザで無効の場合にセッション トラッキングが実行される。

パラメータ名	デフォルト値	パラメータ値
ConsoleMainAttribute		WebLogic Server Administration Console のセッション モニタを有効にした場合、このパラメータを、モニタリングされた各セッションを認識するためのセッション パラメータの名前に設定する。詳細については、「WebLogic ドメインのモニタ」を参照。

jsp-descriptor 要素

jsp-descriptor 要素は、JSP のパラメータ名と値を定義します。パラメータは名前と値の組み合わせで定義します。次の例では、complieCommand パラメータのコンフィグレーション方法を示します。この例のパターンを使って、すべての JSP コンフィグレーションを入力してください。

```
<jsp-descriptor>
     <jsp-param>
          <param-name>
            compileCommand
          </param-name>
          <param-value>
            sj
          </param-value>
     </jsp-param>
</jsp-descriptor>
```

JSP パラメータの名前と値

次の表では、<jsp-param> 要素内で定義できるパラメータの名前と値について説明します。

パラメータ名	デフォルト値	パラメータ値
compileCommand	javac、または WebLogic Server Administration Console の [サーバ] の [コンフィグレーション] の [コンパイラ] タブでサーバ用に定義した Java コンパイラ	生成された JSP サーブレットのコンパイルに使用する標準 Java コンパイラの絶対パスを指定する。たとえば、標準 Java コンパイラを使用するには、以下のようにシステム内の場所を指定する。 <param-value> /jdk130/bin/javac.exe </param-value> パフォーマンスを向上させるために、IBM Jikes や Symantec sj などの別のコンパイラを指定する。
compileFlags	なし	1 つまたは複数のコマンドライン フラグをコンパイラに渡す。フラグが複数の場合は、スペースで区切る。次に例を示す。 <jsp-param> <param-name>compileflags</param-name> <param-value>-g -v</param-value> </jsp-param>
compilerclass	なし	WebLogic Server の仮想マシンで実行される Java コンパイラの名前（javac または sj のような実行可能コンパイラの代わりに使用する）。このパラメータが設定されている場合、compileCommand パラメータは無視される。

パラメータ名	デフォルト値	パラメータ値
encoding	ユーザのプラットフォームのデフォルト エンコーディング	JSP ページで使用されるデフォルトの文字セットを指定する。標準 Java 文字セット名を使用する。 このパラメータを設定しない場合、デフォルトはユーザのプラットフォームのエンコーディング。 JSP コードに含まれる JSP ページ ディレクティブはこの設定をオーバーライドする。次に例を示す。 <%@ page contentType="text/html; charset=custom-encoding"%>
compilerSupportsEncoding	true	true に設定すると、JSP コンパイラは、JSP ページの page ディレクティブに含まれる contentType 属性で指定されたエンコーディングを使用する。contentType が指定されていない場合は、jsp-descriptor の encoding パラメータで定義されたエンコーディングを使用する。 false に設定すると、JSP コンパイラは、中間の .java ファイルを作成するときに JVM 用のデフォルト エンコーディングを使用する。
keepgenerated	false	JSP コンパイル プロセスの間に生成される Java ファイルを保存する。このパラメータを true に設定しない限り、生成された Java ファイルはコンパイル後に削除される。
noTryBlocks	false	JSP ファイルに多数のまたは深くネストされたカスタム JSP タグがあり、コンパイル時に java.lang.VerifyError 例外を受け取った場合は、このフラグを使って JSP を正しくコンパイルできるようにする。
packagePrefix	jsp_servlet	すべての JSP ページがコンパイルされるパッケージを指定する。

パラメータ名	デフォルト値	パラメータ値
pageCheckSeconds	1	JSP ファイルが変更されたために再コンパイルする必要があるかどうかをチェックする間隔を秒単位で設定する。変更されている場合は、依存関係もチェックされ、再帰的に再ロードされる。 0 に設定した場合は、リクエストされたときにページがチェックされる。-1 に設定した場合は、サーバが再起動するまでページはチェックされない。JSP ページによって使用されるクラスでサーブレットのクラスパスに存在するものもすべて再ロードされる。
precompile	false	true に設定すると、Web アプリケーションのデプロイ (再デプロイ) 時または WebLogic Server の起動時に、変更されたすべての JSP が自動的にあらかじめコンパイルされる。
verbose	true	true に設定すると、デバッグ情報がブラウザ、コマンドプロンプト、および WebLogic Server ログ ファイルに出力される。
workingDir	内部に生成されるディレクトリ	WebLogic Server が、JSP 用に生成された Java と コンパイル済みのクラス ファイルを保存するディレクトリの名前。

container-descriptor 要素

<container-descriptor> 要素は、Web アプリケーション用の汎用パラメータを定義します。

check-auth-on-forward 要素

<check-auth-on-forward/> 要素は、サーブレットまたは JSP から転送されたリクエストの認証を必要とするときに追加します。再認証を必要としない場合、このタグは省略します。次に例を示します。

```
<container-descriptor>
    <check-auth-on-forward/>
</container-descriptor>
```

デフォルトの動作はサーブレット 2.3 仕様のリリースで変更されたことに注意してください。この仕様では、転送されたリクエストに認証は必要ありません。

redirect-with-absolute-url

<redirect-with-absolute-url> 要素は、javax.servlet.http.HttpServletResponse.SendRedirect() メソッドでのリダイレクトに相対 URL と絶対 URL のどちらを使用するかを制御します。プロキシ HTTP サーバを使用しており、URL を非相対リンクに変換したくない場合は、この要素を false に設定します。

デフォルトの動作では、URL が非相対リンクに変換されます。

charset-params 要素

<charset-params> 要素は、Unicode 以外の処理でコードセット動作を定義するために使います。

input-charset 要素

<input-charset> 要素を使って、GET データと POST データの読み取りにどの文字セットを使用するのかを定義します。次に例を示します。

```
<input-charset>
    <resource-path>/foo</resource-path>
    <java-charset-name>SJIS</java-charset-name>
</input-charset>
```

詳細については、3-16 ページの「HTTP リクエストのエンコーディングの識別」を参照してください。

次の表では、<input-charset> 要素内で定義できる要素について説明します。.

要素	必須 / 省略可能	説明
<resource-path>	必須	リクエストの URL に含まれている場合、<java-charset-name> で指定されている Java 文字セットを使用するように WebLogic Server に知らせるパス。
<java-charset-name>	必須	使用する Java 文字セットを指定する。

charset-mapping 要素

<charset-mapping> 要素を使って、IANA 文字セット名を Java 文字セット名にマップします。次に例を示します。

```
<charset-mapping>
    <iana-charset-name>Shift-JIS</iana-charset-name>
    <java-charset-name>SJIS</java-charset-name>
</charset-mapping>
```

詳細については、3-17 ページの「IANA 文字セットの Java 文字セットへのマッピング」を参照してください。

次の表では、<charset-mapping> 要素内で定義できる要素について説明します。

要素	必須 / 省略可能	説明
<iana-charset-name>	必須	<java-charset-name> 要素で指定された Java 文字セットにマップされる IANA 文字セット名を指定する。
<java-charset-name>	必須	使用する Java 文字セットを指定する。

索引

し

せ

て

と

に

ふ

ま

り