

Java™ magazine

By the Java community 

The New Java Developers

1

JAVAMAILが届ける メッセージ

JavaMailとFaceletsを利用
した電子メール送信Web
アプリケーションの作成

5

アプリにWEBを取り 込む

JavaFXのWebViewを使ってア
プリにHTML、CSS、JavaScript
を埋め込めば、これまでにない
アプリの開発が可能に

9

JAVAFXとSWINGの 統合

JavaFXにより、視覚効果を
利用した動きのあるツールバー
を実現

11

アスペクト指向プログ ラミングは必要か？

Java EEでは、明らかにAOP
を使用すべき場面はほとんど
ない

16

GLASSFISHで実現す るシンプルなクラスタ リングと高可用性

GlassFishクラスタでアプリ
ケーションの高可用性をサポート
し、セッションのフェイルオー
バーを実現



T. LAMINE BA

BIO

JavaMailが届けるメッセージ

JavaMailとFaceletsを利用した電子メール送信Webアプリケーションの作成

この記事では、コアJavaMail APIを利用して電子メールを送信する、シンプルなWebアプリケーションを作成します。このアプリケーションには、3つのWebページが含まれます。それぞれトップページ、「送信完了」の確認ページ、「送信失敗」の通知ページです。

注:アプリケーションのソース・コードは、こちらからダウンロードできます。

Webアプリケーションのトップページ(図1参照)には、次のWebコンポーネントが含まれます。

- 送信者の電子メール・アドレスの入力フィールド
- 受信者の電子メール・アドレスの入力フィールド
- 電子メールの件名の入力フィールド
- 電子メールの本文の入力フィールド
- Simple Mail Transfer Protocol (SMTP) サーバーのIPアドレスまたはホスト名の入力フィールド
- SMTPサーバーが要求するユーザー名の入力フィールド
- SMTPサーバーが要求するパスワードの入力フィールド

テキストだけではありません
JavaMail APIは、**JavaBeans Activation Framework**を利用して、プレーンなテキスト以外のコンテンツも扱えます

- SMTPサーバーが使用するポート番号の入力フィールド
 - 電子メール送信ボタン
- 確認ページ(図2参照)と通知ページ(図3参照)では、ユーザーがトップページに戻るためのボタンのみ必要です。

JavaMail API

JavaMail APIは、電子メッセージの読取り、作成、送信などの一般的な電子メール機能を提供するパッケージです。JavaMailはプラットフォーム非依存、プロトコル非依存のフレームワークであり、Java EEに含まれています。

図4に示すとおり、JavaMailには、アプリケーション・コンポーネントが電子メールの送受信に使用するアプリケーション・レベルのインタフェースがあります。また、プロトコル固有の処理に対応するサービス・プロバイダ(SP)インタフェースもあります。たとえば、SMTPは電子メールの送信に使用されるプロトコルです。Post Office Protocol 3 (POP3)は、電子メール受信の標準になっています。Internet

Mail Access Protocol (IMAP)は、POP3の代わりに使用できます。

さらに、JavaMail APIにはJavaBeans Activation Framework (JAF)が含まれており、Multipurpose Internet Mail Extension (MIME)、URL、添付ファイルなど、プレーンなテキスト以外の電子メール・コンテンツを扱えます。

必要なソフトウェア

このチュートリアルでは、次のソフトウェアを使用します。

- Microsoft Windows プラットフォーム(ここではMicrosoft Windows 7を使用)
- Java EE 6以降のJDK
- NetBeans IDEなどの一般的なIDE(ここではNetBeans IDEバージョン7を使用)
- Webサーバー(Oracle GlassFish Server、GlassFish Server Open Source Edition、Apache Tomcatなど)

方法

このチュートリアルではJavaServer



図1

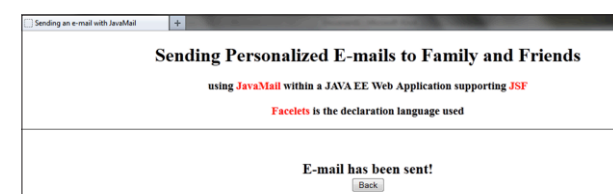


図2



図3

Faces(JSF)テクノロジーを利用してWebアプリケーションを作成します。そのため、次のワークフローを進めることにします。

- 手順1: バッキングBeanを作成する
- 手順2: コンポーネント・タグを使用してWebページを作成する

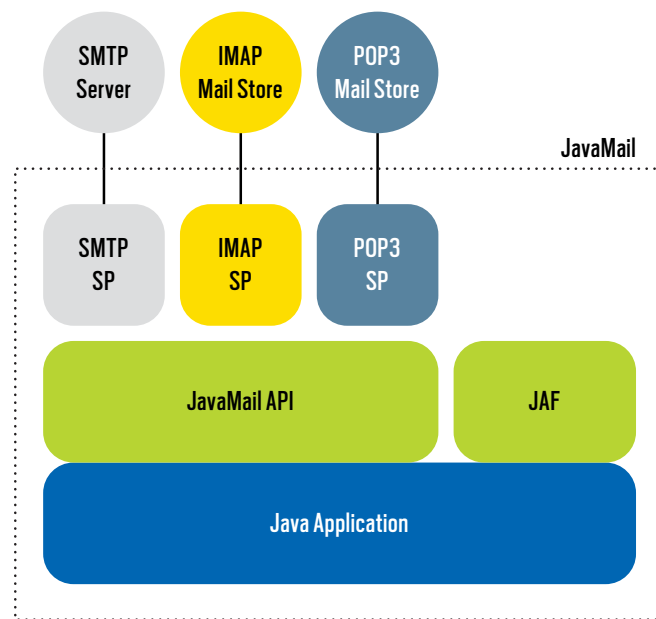


図4

- 手順3: **FacesServlet**インスタンスをマッピングする

バックングBeanの作成

バックングBeanは、JSFテクノロジー固有のマネージドBeanの一種です。Webアプリケーションのロジックを格納しており、Webページに含まれるWebコンポーネントとやり取りします。バックングBeanには、それぞれのWebコンポーネントに対応するprivate

属性、その属性に対応するgetterメソッドとsetterメソッド、さらに次の4つのタスクを実行するメソッドを含めることができます。

- Webページ間のナビゲーションに関連する処理を実行する
- アクション・イベントを処理する
- コンポーネントの値を検証する

- 値変更イベントを処理する
これに従い、**emailJSFManagedBean**というバックングBeanの中に、前述のWebコンポーネント(「**Send E-mail**」ボタンを除く)のそれぞれに必要なgetterメソッドとsetterメソッドを作成します。受信者の電子メール・アドレスが**String**型の**to**という変数だとすると、getterメソッドとsetterメソッドの定義方法は**リスト1**のようになります。

リスト1の**@ManagedBean**は、このバックングBeanをリソースとしてJSF実装に登録するための宣言です。また、**@RequestScoped**は、このマネージドBeanがリクエストの範囲でのみリソースとして存在することを指定しています。つまり、このBeanは、ユー

ザーがWebアプリケーションとやり取りするための1回のHTTPリクエストの期間だけ存在します。

このバックングBeanでは、さらに2つのメソッドを作成します。

1つ目のメソッドの役割は、ユーザーがWebアプリケーションに送信したすべての電子メールを検証することです(**リスト2**参照)。

- `message = "E-mail is required";`

この電子メール検証メソッドは、次の3つの引数をとります。

- JSF実装のコンテキスト。マネージドBeanからユーザー・インタフェースにエラー・メッセージを渡すためのものです。
- このメソッドを呼び出すWebコンポーネントの識別子である**UIComponenttoValidate**。この例でのWebコンポーネントは、ユーザー入力を受け取るテキスト入力フィールドです(**リスト1**参照)。ユーザー入力については**図1**を参照してください。

リスト1

リスト2

```
package useJavaMail;
/** Import all necessary libraries */

@ManagedBean
@RequestScoped
public class emailJSFManagedBean {
    private String to;
    /** Creates a new instance of emailJSFManagedBean */
    public emailJSFManagedBean() {
        to = null;
    }

    /**
     * @return the to
     */
    public String getTo() {
        return to;
    }
    /**
     * @param is the "to" to set
     */
    public void setTo(String to) {
        this.to = to;
    }
}
```

全てのリストをテキストで表示

- **value**変数。検証する必要がある電子メール・アドレスを格納します。
したがって、**リスト2**に示すコードは、次のタスクを実行します。
- Webコンポーネントの値を取得する
- 値が**null**や空であるかどうかをチェックする
- 値が**null**または空の場合は、コンポーネントの**valid**プロパティの値を**false**に設定し、エラー・メッセージを「**E-mail address is required**」に設定する
- そうでない場合は、「@」とピリオド(「.」)が値に含まれているかどうかをチェック

する

- 含まれていない場合は、コンポーネントの**valid**プロパティの値を**false**に設定し、エラー・メッセージを「**E-mail address is invalid**」に設定する
エラー・メッセージは、**FacesContext**インスタンスに送られ、それによって呼び出し元のWebコンポーネントに関連付けられます。
2つ目のメソッドは、JavaMailを利用して電子メールを送信するロジックを扱います。このナビゲーション処理メソッドは、「**Send E-mail**」ボタンのクリックというアクションによってトリガーされます(**リスト3**参照)。

電子メールを簡単に
JavaMail APIは、
電子メッセージの
読取り、作成、送
信などの一般的な
電子メール機能を
提供するパッケー
ジです

このようなメソッドを、アクション・メソッドといいます。このメソッドは、引数をとらないpublicメソッドであり、Webアプリケーションで移動先となるページを表す文字列を返します。この例では、このメソッドで電子メールを生成して送信します。電子メールの送信が成功した場合、このメソッドは**g_response**（「good response」の略）を返し、その結果ブラウザでg_response.xhtmlページが表示されます（図2参照）。電子メールの送信が失敗した場合は、**b_response**（「bad response」の略）を返し、ブラウザではb_response.xhtmlページが表示されます（図3参照）。

JavaMailを利用して電子メールを送信するには、まず**Session**クラスの電子メール・セッション・インスタンスを初期化します。こ

の電子メール・セッションがJavaMailの起点となります。このメソッドでは、**java.util.Properties**クラスを使用して、電子メール・サーバー、ユーザー名、パス

ワードなどの情報を取得します。これらの情報は、アプリケーション全体で共有できます。この例では、次のようにして、**Session**クラスのデフォルト・インスタンスを作成します。

```
session =
Session.getDefaultInstance(props, null);
```

次に、作成したセッションを用いて、**Message**クラスの電子メールを生成します。ただし、**Message**は抽象クラスであるため、代わりにサブクラスの**MimeMessage**を使用します。このクラスのメッセージでは、さまざまなRFCで定義されているMIMEタイ

プとヘッダーを利用できます。次のように、セッションを引数としてメッセージを作成します。

```
MimeMessage message =
new MimeMessage(session)
```

電子メールを送信するには、**Transport**型のオブジェクトを操作します。メッセージは、SMTP転送プロトコルを使用して送信します。**Transport**クラスが送信を処理します。次のようにしてオブジェクトをインスタンス化します。

```
Transport transport =
session.getTransport("smtp");
```

このトランスポート・オブジェクトが、指定された認証情報（SMTPサーバーのアドレス、SMTP接続を受け付けるポート番号、ユーザー名、パスワード）を使用し、SMTPサーバーへの接続を試みます。

```
transport.connect(this.smtp,
this.port, this.username,
this.password);
```

SMTPサーバーが接続を許可したら、**send**を用いて電子メールが送信されます。

最後に、**close**コマンドを呼び出して、トランスポート・サービスをクローズします。

```
transport.sendMessage(message,
message.getAllRecipients());
transport.close();
```

注: このバックギンBeanを含むファイルは、WebアプリケーションのSources Packagesディレクトリの下に配置する必要があります。

リスト3

```
public String submitEmail() {
    // create e-mail and send
    /** Initialize variables */
    props = new Properties();
    // fill props with session and message information
    session = Session.getDefaultInstance(props, null);
    message = new MimeMessage(session);
    try {
        message.setContent(this.getDescr(), "text/plain");
        message.setSubject(this.getSubject());
        fromAddress = new InternetAddress(this.getFrom());
        message.setFrom(fromAddress);
        toAddress = new InternetAddress(this.getTo());
        message.setRecipient(RecipientType.TO, toAddress);

        // Transport message
        message.saveChanges(); // implicit with send()
        Transport transport = session.getTransport("smtp");
        transport.connect(this.smtp, this.port, this.username, this.password);
        if(transport.isConnected() == false)
            return "b_response";
        transport.sendMessage(message, message.getAllRecipients());
        transport.close();

    }
    catch (MessagingException me) {
        // handle catch
        return "b_response";
    }

    return "g_response";
}
```

全てのリストをテキストで表示

Webページの作成

このアプリケーションのWebページでは、Facelets宣言言語を活用することで、さまざまなWebコンポーネントのためのタグを生成します。

トップ・ページを作成します。このペー

ジでは、Webコンポーネントに関連するタグとして、**inputText**、**inputSecret**、**inputTextArea**、**commandButton**の4種類を使用します。

inputTextは、HTMLでtypeが**text**であるinputタグに相当します。この例では、送信者

独立した技術

JavaMailは、プラットフォーム非依存、プロトコル非依存のフレームワークです

のアドレス、受信者のアドレス、電子メールの件名、SMTPサーバーのアドレス、SMTPサーバーのユーザー名、SMTPサーバーのポート番号を取得するために、このタグを利用します。

inputSecretは、HTMLでtypeが**password**であるinputタグに相当します。これもユーザー入力を受け付けるフィールドです。ただし、**inputSecret**は、**inputText**タグとは異なり、ユーザーが入力した値を表示しません。SMTPサーバーのパスワードを取得するために、このタグを使用します。

inputTextAreaは、HTMLの**textarea**タグに相当します。送信する電子メールの本文を取得するために、このタグを使用します。

このアプリケーションでは、標準バリデータを利用するか、バックینگBeanに実装した検証メソッドを呼び出して、ユーザー入力を検証します(リスト2参照)。

図5

図6

たとえば、リスト4に示すコードを使用した「FROM」アドレス・フィールドでは、Faceletsを利用して**emailJSFManagedBean.validateEmail**検証メソッドを呼び出します。

なお、messageタグ(<h:message/>)の目的は、電子メール・アドレスが検証されなかったときにエラー・メッセージを表示することです(図5参照)。

別の例として、「SUBJECT」フィールドでは、リスト5に示すように、Faceletsの標準バリデータを利用しています。

idが**subject**である**inputText**に対して**validateRequired**タグ(<f:validateRequired/>)を適用しています。これにより、「SUBJECT」フィールドが空で、フォームが送信された場合、フォームが無効になります。この例では、messageタグ(<h:message/>)がある場所にエラー・メッセージが表示されます(図6参照)。

確認ページとエラー通知ページを作成します。確認ページ(g_response.xhtml)は、電子メールが送信されたときに呼び出されます(図2参照)。一方、エラー通知ページ(b_response.xhtml)は、電子メールが送信できなかった場合に呼び出されます(図3参照)。どちらのページでも、FaceletsによるWebコンポーネントは、次のような**commandButton**タグが1つだけ含まれます。

```
<h:form>
  <h:commandButton id="back"
    value="Back" action="index">
</h:form>
```

このコードが生成するボタンをユーザーがクリックすると、トップ・ページ(index.xhtml)に戻ります。

リスト4

リスト5

リスト6

```
<h:form>
  <table>
    <tr>
      <th style="width:100px" align="right">FROM:</th>
      <td>
        <h:inputText id="from" size="100"
          validator="#{emailJSFManagedBean.validateEmail}"
          value="#{emailJSFManagedBean.from}" />
        <span style="margin-left:10px"><h:message style="color:red"
          for="from"/></span>
      </td>
    </tr>
  </table>
</form>
```

全てのリストをテキストで表示

FacesServletインスタンスのマッピング

最後の手順として、Webデプロイメント・ディスクリプタであるweb.xmlファイルを編集し、**FacesServlet**インスタンスをマッピングします。リスト6に一般的な例を示します。

なお、NetBeansなどのIDEを使用していれば、このマッピングは自動的行われます。

まとめ

この記事では、コアJavaMail APIを利用して電子メールを送信する、シンプルなWebアプリケーションを作成する方法を学習しました。これを応用すれば、電子メールをカスタマイズしさまざまな形で統合できる、インタラクティブなWebアプリケーションを作成できます。</article>

LEARN MORE

- [NetBeansオンライン・ヘルプ](#)
- [The Java EE 6 Tutorial: Basic Concepts, fourth edition \(Prentice Hall, 2010\)](#)
- [「JavaMail APIの概要」](#)



JOSH MARINACCI



アプリにWebを取り込もう

JavaFXのWebViewを使ってアプリにHTML、CSS、JavaScriptを埋め込めば、これまでにないアプリの開発が可能に

JavaFX 2.0は、リッチ・クライアント・アプリケーションの開発方法を「再発明」します。新しい機能とグラフィック基盤により、開発者は魅力的で反応のよい、優れたインタフェースを作成できるようになります。JavaFX 2.0の優れた新機能の1つに、**WebView**コントロールがあります。**WebView**を使用すると、アプリケーションの中にHTMLコンテンツをそのまま埋め込むことができるため、デスクトップとWebが協調するまったく新たな世界が開かれます。

JavaFXチームは、開発者がHTMLを、本来のWebブラウザでの機能と正確さをそのままに、直接アプリケーションに埋め込むことができるようにしたいと考えました。これは、従来のSwing **JEditorPane**ではできなかったことです。そのために、オープンソースのHTMLレンダリング・エンジンである**WebKit**を利用することにしましたこれは、さまざまなデスクトップおよびモバイル・プラットフォームで、急速に標準的なレンダラとなっています。Safari、Google Chrome、iOS、Android、webOS、さらにはAmazon Kindleでも使われています。

JavaFXチームはまた、新しいWebコントロールを、ネイティブ**WebKit**ラ

イブラリの単なるラッパー以上のものにしたいと考えました。**WebView**は、画面上の描画とすべてのネットワーク・アクセスにJavaを使用することで、単なるラッパーで可能なことに比べ、ずっと緊密な統合を実現しています。**WebView**では、この回転するWebページの例が示すように、3Dシーンの中に描画することもできます。

WebViewコントロールのさまざまな使用方法

WebViewをアプリケーションで使用するのとは、とても簡単です。**WebView** Javaオブジェクトを作成し、そこから実際に**WebKit**に接続するJavaオブジェクトである**WebEngine**への参照を取得するだけです。ページをロードするには、WebページのURLを用いて**engine.load()**を呼び出します。**engine.loadContent()**を使用して、アプリ内で文字列から生成したコンテンツを直接ロードすることもできます。そして、作成した**WebView**を、他のコントロールと同じようにJavaFXシーンに配置します。これで完了です。

もちろん、**WebView**のAPIがそれだけのものではあれば、この記事は必要ないでしょう。Webコンテンツをア

プリの中に持ち込むことができれば、それで実際に何ができるでしょうか。おもなユースケースには、次の4つがあると思います。

- Webの一部をデスクトップ・アプリに埋め込む。Google マップをアプリに埋め込むのが良い例です。
- Javaを使用してリモートのWebコンテンツを操作する。つまり、コンテンツはWebからコンテンツをロードしますが、Javaコードを使ってそれを変更します。
- 内部で生成したコンテンツやローカルにロードしたコンテンツを表示する。リアルタイムのレポート生成が良い例です。
- JavaScript側からJavaコンテンツを操作する。**WebView**では、JavaScriptからJava側にアクセスする方法は提供されていませんが、少し頭をひねればいろいろと面白いことができます。

Webコンテンツの読み取りと埋込み

最初のユースケースから始めましょう。Webの一部をデスクトップ・アプリに埋め込みます。単にWebページを

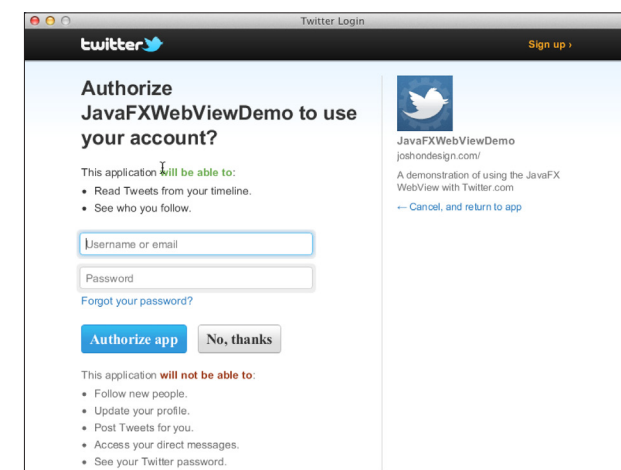


図1

ラップするのではなく、私が本当に悩まされている問題であるTwitterの認証に対処できれば、素晴らしいことではないでしょうか。

Twitterは、OAuthを使用してサード・パーティのアプリを認証しています。OAuthでは、アプリケーションはユーザーをTwitter.com上の特別なURLへ転送する必要があります。そのページでユーザーがアプリを承認すると(図1を参照)、ユーザーはアクセスを要求したアプリに戻ります。これは、アプリがWebサイトであれば問題ありませんが、アプリがデスクトップ・プログラムである場合、ユーザーが戻るべきサーバーURLがありません。この問題に対処するため、Twitter



図2

からユーザーにPINコードが付与され、それをユーザーがアプリに入力します。この方法は正しく機能はしますが、非常に煩雑で手間がかかります。また、画面が切り替わり、ページがリダイレクトされる間に、ユーザーが迷子になってしまうことがよくあります。

この問題を解決するため、**WebView**を使ってみましょう。新しくWebブラウザを開くと、ユーザーがコンテキストを見失ってしまうかもしれません。そこで、Twitterのページをアプリに直接埋め込みます。ユーザーがアプリを承認すると、ドキュメントからPINコードを取得できます。ユーザーが手動でPINコードを入力する必要はありません。これには、**WebView.document**プロパティを使用します。その方法を以下に示します。

まず、OAuth認証を実行するためのTwitterライブラリが必要です。すべてを手動で行うことも可能ですが、そのためには暗号化やハッシュ・コード生成を処理する必要があり、手間がかかるだけでなく、エラーが起きやすくなります。このような細かいことを処理してくれるものとして、ここではオープンソースの**Twitter4J**ライブラリを使用します(**リスト1**参照)。

最初に、アプリ固有のConsumer keyとConsumer secretを使用してTwitter4Jを初期化します。これらの値は、dev.twitter.comのアプリ設定ページから取得できます。次

に、認証用のURLをリクエストします。このURLをユーザーに表示することで、ユーザーがアプリを承認できるようにする必要があります。現在はmainスレッドにいますが、すべてのJavaFXコントロールにはGUIスレッドからアクセスする必要があるため、**リスト2**に示すように、**Application.launch()**を呼び出して、適切なスレッドでウィンドウを開きます。

URLを開くのは簡単です。**WebView**を作成し、そのengineを取得して、**load()**を呼び出します。次に、ユーザーがユーザー名とパスワードを入力して「OK」ボタンを押したときに、それを検知して、PINを取得できるようにします。そのためには、**リスト3**に示すように、engineのdocumentプロパティにリスナーを追加します。

ユーザーがログインし、アプリを承認すると、**図2**のようなページが表示されます。この中ほどに、次のようなマークアップがあります。

```
<div id="oauth_pin">
<code>12334928</code></div>
```

このdocumentを実際に処理するのが、**grabPin**メソッドです。**WebEngine**から得られたdocumentは、実際にはW3C Document Object Model (DOM)のdocumentオブジェクトであり、**getElementById()**などの標準的なAPIを使ってパースできます。この例では、ユーザーに付与されたPINを含む**DIV**要素である**oauth_pin**要素を探します。この中のcode要素のテキスト・コンテンツが、求めているものです(**リスト4**参照)。

documentがnullであったり、OAuth PINがなかったりしたときは、途中でreturnしていることに注意してください。これは、

リスト1 / リスト2 / リスト3 / リスト4 / リスト5 / リスト6

```
public static void main(String ... args) throws Exception {

    //connect to twitter and get an authorization URL
    twitter = new TwitterFactory().getInstance();
    twitter.setOAuthConsumer(consumerKey,consumerSecret);
    requestToken = twitter.getOAuthRequestToken();
    authURL = requestToken.getAuthorizationURL();

    ...
}
```

全てのリストをテキストで表示

WebEngineのdocumentプロパティが何度か変更される可能性があるためです。まずアプリが起動したとき、次に最初のページをロードしたとき、そしてPINページを取得したときです。ここで必要なのはPINページだけです。他の場合では、このコードは何もしません。

PINテキストが取得できたら、Twitterの認証プロセスを完了して、承認されたアプリのみが取得できるユーザー情報を要求できます。この例では、**リスト5**に示すように、友達の数を探しています。

このように、アプリでのWebコンテンツの埋込みと読取りがとても簡単であることが分かりました。documentの変更を監視し、標準的なDOM APIを使用してページのコンテンツを読み取ります。

Webコンテンツの変更

では、ページのコンテンツを読み取るだけでなく、コンテンツを変更したい場合は、どうでしょう

か。**WebView**では、返された**Document**オブジェクトを用いてDOMを変更したり、ページにJavaScriptを注入してDOMを操作したりすることができます。

次の例では、Webページをロードして、注入の手法を用いてその場でスタイルを変更し、ページを読みやすくします(**リスト6**参照)。

手順は先ほどと同じです。ページをロードして、documentにリスナーを追加します。この例では、私の技術系ブログである

joshondesign.comをロードします。このデザインはなかなかよいのですが、長い文章を読むには少し派手すぎます。このページを、簡素なフォントを使ったモノクロの「読み物」モードに変えることができれば、すばらしいでしょう。これは印刷にも適したものになります。**modifyDoc**メソッドを使えば、ページのbodyのスタイルを変更するJavaScriptを注入できます(**リスト7**参照)。

注入できるのは文字列だ

これがすごい
JavaFX 2.0の
新しい機能と
グラフィック基
盤により、開発
者は魅力的で
反応のよい、
優れたインタ
フェースを作成
できるようにな
ります。

けです。そこで、最初にすべてのコードを **StringBuffer** に入れます(これをより強力なバージョンにするとしたら、JavaScriptをコードの中で生成するのではなく、ファイルからロードすることができます)。スクリプトの準備ができれば、**engine.executeScript()** を呼び出すことで注入できます。このメソッドは、コードの戻り値をラップしているオブジェクトを返します。この例では、**bodys** オブジェクトです。この戻り値を使用して、JavaScript

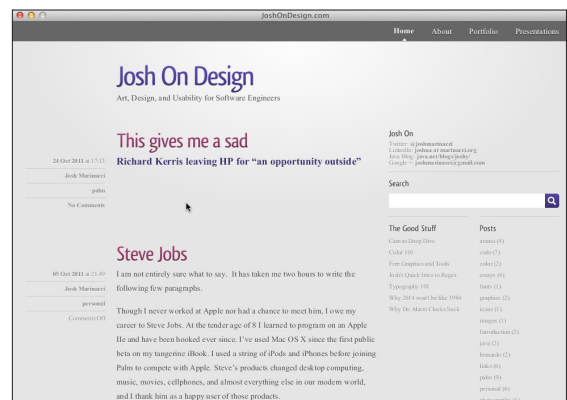


図3

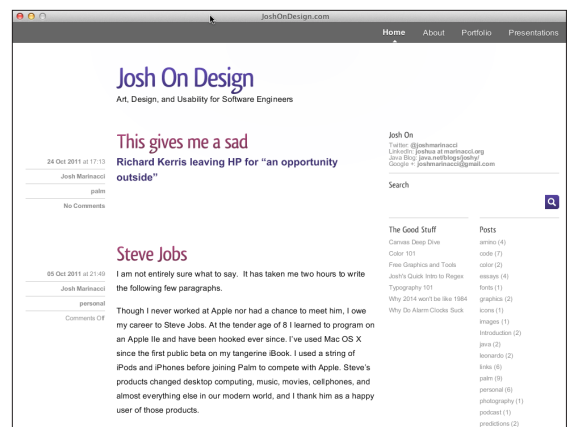


図4

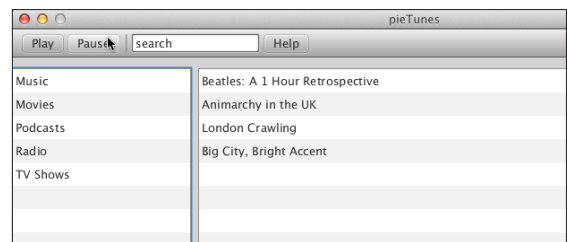


図5

側からJava側に簡単なデータを渡すことができますが、この例では、コードが正常に実行されたことを確かめているだけです。

図3に、私のブログのデザインを示します。そして図4に、新しいスタイルのブログを示します。

生成されたコンテンツの操作

最後の例として、**WebView** を使ってアプリ自身が持つコンテンツを表示するヘルプ・システ

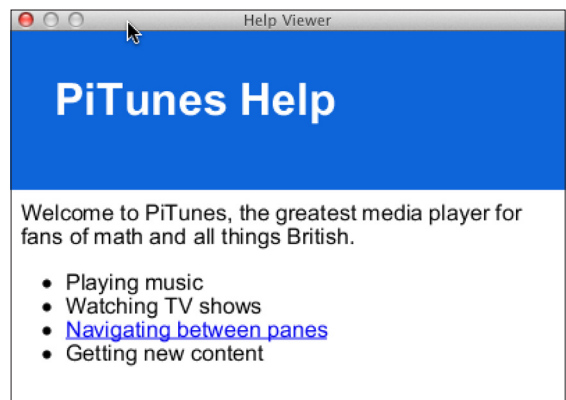


図6

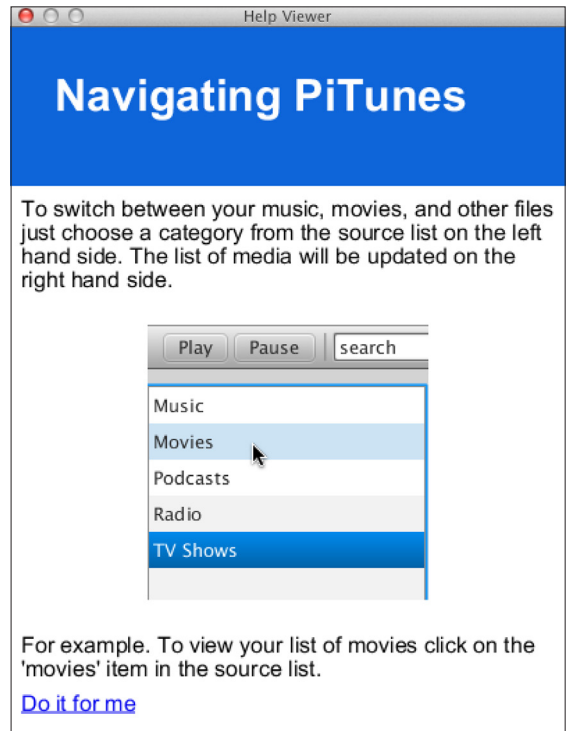


図7

リスト7

リスト8

```
private void modifyDoc(Document newDoc, WebEngine engine) {
    StringBuffer script = new StringBuffer();
    //grab the body
    script.append("var bodys = document.getElementsByTagName('body');");
    //change the colors
    script.append("bodys[0].style.backgroundColor = '#FFFFFF';");
    script.append("bodys[0].style.color = '#000000';");
    //set a new default font
    script.append("bodys[0].style.fontFamily = 'sans-serif';");

    script.append("bodys;");
    //execute
    Object retval = engine.executeScript(script.toString());
    //return value doesn't matter in this case
    p("return value = " + retval);
    JSONObject obj = (JSONObject) retval;
}
```



全てのリストをテキストで表示

ムを示します。かつては、アプリの中でリッチテキストのヘルプを表示するには、独自のヘルプ・エンジンを使用する必要がありました。しかし、今ではすべてをHTMLとして記述するだけで、それをウィンドウに表示できます。

この例では、PiTunesという架空のメディア・プレーヤを作成しました。図5に示すように、最上部にツールバーがあり、左にメディア・タイプのリスト(ソース・リストと呼びます)、右にメディア・ファイルのリストがあります。

ムービーのリストを表示するには、リストにある「**Movies**」という項目をクリックします。しかし、この操作は分かりにくいかもしれないので、説明するためのオンライン・ヘルプを追加します。ユーザーが「Help」ボタンをクリックすると、ヘルプ・コンテンツを表示する第2のウィンドウが開きます。リスト8を参照してください。

ここで注意すべき重要なことが2つあります。第1に、ヘルプ・ファイルを、

getResource() で得られるURLからロードしているという点です。これは、ディスク上のコンテンツだけでなく、JARファイル内部のコンテンツもロードできることを意味しています。第2に、最初のページでリンクをクリックして次のページに移ると、画像が表示されるという点です。この画像は、リソース・パッケージの中にあるPNGファイルです。マークアップでは、**** のように、これをローカルに参照しています。

WebView は、すべてのリソースのロードにJavaを使用するため、JARファイルやデータベースの中などJavaに特有の場所やデータベースを含め、あらゆるところからコンテンツをロードできます。すべてJavaと同様です。図6と図7を参照してください。

これで、リッチなヘルプ・コンテンツをアプリの中に入れることができました。さらに一段階進めることもできます。何かを実行する方法をユーザーに伝えるだけでなく、それを

リスト9

```
engine.setOnAlert(new EventHandler<WebEvent<String>>() {
    @Override
    public void handle(WebEvent<String> t) {
        sourceList.getSelectionModel().select(1);
    }
});
```

👉 全てのリストをテキストで表示

ユーザーの代わりに実行することもできます。ヘルプ・テキストが説明している操作をアプリに実行させるリンクを作ってみましょう。

もちろん、このような自動化のためには、JavaScript側のコードがJava側のコードにアクセスできなければなりません。それは現時点では不可能です(このような機能は、JavaFXの将来のリリースで実現されるかもしれません)。しかし、うまい回避策があります。JavaScriptコードでは、標準の `alert()` 関数を呼び出すことが可能です。一方、**WebEngine** では、alertイベントを処理するコールバック・ハンドラを作成できます。この関数を、実際にalertダイアログを開くのではなく、JavaScript側がJava側と通信するための方法として利用できます。この例では、ユーザーがリンクをクリックしたときに、それを検知してGUIを更新します。

マークアップ側では、次のように、JavaScriptへのリンクを用意します。

```
<a href="javascript:alert('movies')">Do it for me</a>
```

Java側では、**リスト9**に示すような、alertイベントに対するハンドラを作成しました。

これで、ユーザーがリンクをクリックすると、GUIで選択されている項目が2行目(「Movies」)に変わります。ごくわずかなコーディングで、ユーザーはアプリケーションの操作を読むだけでなく見ることもできるようになりました。

まとめ

HTML、CSS、そしてJavaScriptは非常に強力なテクノロジーであり、そのテクノロジーをJavaアプリに埋め込んで、これまでになかったアプリの開発ができるようになりました。この記事は、可能性のほんの一端に触れたに過ぎません。ほかに試してほしいこととしては、RSSリーダーの埋込み、Webページをオフラインで読むためのキャッシュ、ダブルクリック可能なJavaアプリとしてローカルに動作する完全にWebベースのインタフェースの作成などがあります。可能性は無限です。</article>

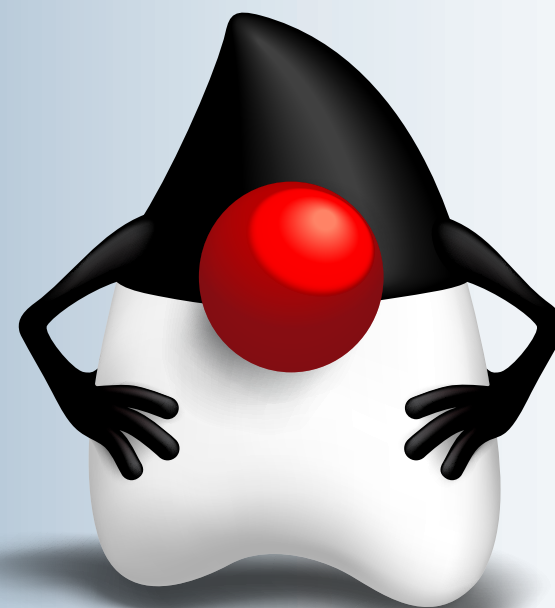
LEARN MORE

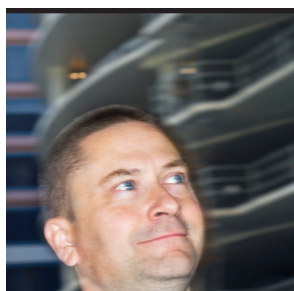
- [JavaFX](#)
- [Josh On Designブログ](#)



お住まいの地域のJAVA ユーザー・グループ では、あなたを必要と しています

JUGを検索





SIMON RITTER



パート3 JavaFXとSwingの統合

JavaFXにより、視覚効果を利用した動きのあるツールバーを実現

この記事は、JavaFXと既存のSwingアプリケーションとの統合に関する全3回シリーズの最終回です。パート1では、JavaFXシーンをJComponentとしてラップするための基本事項とイベントの処理方法を見ました。パート2では、サンプルアプリケーションの中でJavaFXのテーブルを使用する方法を見ました。今回、株式の詳細情報を提供する同じサンプルアプリケーションのための面白いツールバーを作成するために、視覚効果を利用します。

注: サンプルアプリケーションのコードは、[こちら](#)からダウンロードできます。

このアプリケーションには、ポートフォリオの表示、株価の更新、新規ポートフォリオの作成といったコマン

ドにすばやくアクセスするためのアイコンを用いたツールバーがあります。この機能は、SwingのJToolBarコンポーネントを利用しています。このツールバーを視覚的により楽しくするため、すべての機能を維持しながら、JavaFXのToolBarコントロールに置き換えます。

機能の追加

まずJavaFXのコードから始め、その後でコードをアプリケーションに統合する方法を説明します。アイコンに機能を追加するため、ActiveIconという新しいクラスを作成します。コンストラクタの概要をリスト1に示します。

リスト1のコードは単純に、コンストラクタに渡される画像名を引数と

してとり、ツールバーで利用できるImageViewノードを作成しています。また、これから作成するエフェクトの1つのために、画像の幅も必要です。コードから分かるように、アイコンに設定可能なエフェクトとして、フェード、スピン、グロー、スライドの4種類を定義しています。

これらのエフェクトの実装をリスト2とリスト3に示します。これもコンストラクタの一部です。

視覚効果のためには、Timelineを利用してアイコンのプロパティを変更します。使用するプロパティはそれぞれがDoublePropertyであり、ImageViewオブジェクトの適切なメソッドを呼び出すことでアクセスできます。プロパティを取得したら、0秒に1つのKeyFrameがあり、1,000ミリ秒(1秒)にもう1つのKeyFrameがあるTimelineを作成します。それぞれのKeyFrameでKeyValueを使用して、変更の対象とするプロパティを関連付け、Timelineで定義した時点におけるそのプロパティの値を定義します。エフェクトを連続的にするため、Timelineに自動反転を設定し、繰り返し回数を無限に設定します。タイムラインの中で変更されるアイコンのプロパティとその値は、暗黙的にバインドしています。

フェード・エフェクトではアイコンの

透明度が変化し、グロー・エフェクトではx軸とy軸の拡大率が変わり、スピン・エフェクトではアイコンが360度回転します。

スライド・エフェクトはもう少し複雑で、アイコンがy軸に沿ってアイコンの幅の2倍だけ移動します。他のアイコンと重ならないように、アイコンの背面に透明な四角形を配置して、十分なスペースを確保します。そのために、この四角形の不透明度を0に設定します(visibilityプロパティをfalseに設定するのではうまくいかないことに注意してください)。

新しいツールバーの作成

これで、リスト4に示すように、ActiveIconクラスを使用してツールバーを作成できます。リスト4のコードでは、StocksMonitorMainWindowクラスに、メインGUIの作成を担当する新しいメソッドを追加しています。ここでは、パート1で説明した手法を使用して、JavaFXノードをラップし、他のComponentオブジェクトと同様にSwing GUIで操作できるJFXPanelを作成します。

ツールバー・ノードを作成するには、JFXPanelの新規finalインスタンスを作成し、JavaFXのスレッドで実行するタスクを設定します。JavaFX



JavaFXとSwingに関する全3回の連載記事のハイライトを語るSimon Ritter

写真: BOB ADLER





ADAM BIEN



アスペクト指向プログラミングは必要か？

Java EEでは、明らかにAOPを使用すべき場面はほとんどない

アスペクト指向プログラミングの目的は、再利用可能な機能をビジネス・ロジックから分離することです。Wikipediaには、次のように書かれています。「コンピューティング分野におけるアスペクト指向プログラミング(AOP)は、クラス間を横断するような機能を分離することでモジュール性を高めることを目的としたプログラミング・パラダイムである。AOPはアスペクト指向ソフトウェア開発の基礎となっている」

AOPは数年前に話題となったものの、J2EEで盛んに使用されるようにはなりません。それには理由があります。

注:この記事のサンプルJava EEプロジェクト([AOP and Java EE](#))はProject KenaiのWebページから入手できます。

アスペクトはコモディティ

アスペクトは、1998年初めのEnterprise JavaBeans(EJB)1.0仕様

(JSR 318)の不可欠な部分でした。開発者は、**Bean**クラスと**Remote**および**Home**インタフェースを開発する必要がありましたが、横断的なアスペクトの開発は不要でした。スレッド管理、プーリング、セキュリティ、トランザクション、セッション処理、そして永続化までもが、ビジネス・ロジックから明確に分離された構成可能なアスペクトでした。初期のEJB 1.0の時代は、横断的な機能をPOJOで構成できました。

EJB 1.1とJ2EEが登場すると、JavaクラスはXMLに置き換わり、最近のJava EE 5およびJava EE 6仕様ではXMLが必須ではなくなりました。大部分のユースケースをアノテーションで構成でき、さらにXMLを使用してアノテーションをオーバーライドできます。

EJB 1.0の全体的な考え方はアスペクトに基づいていますが、アスペクトという語自体は仕様に2回しか登場せず、その文脈も異なります。AOPと

EJB 1.0はほぼ同時期に「発明」されたのです。

J2EEでは、EJB Beanをトランザクションなどによって構成可能な方法でデコレートすることが、開発における不可欠な部分でした。他のフレームワークも同じ機能を提供していましたが、その機能をAOPと呼んでいました。

EJB 3.1では、単純なBeanを定義することで、さまざまなアスペクトを利用できます。たとえば、EJB 3.1のステートレスBeanには、最初からトランザクションとライフサイクルのアスペクトが備わっています。

```
@Stateless public class Greeter {
    public String greet() {
        return "Morning!";
    }
}
```

この**greet()**メソッドは、直接呼び出されることはありません。コンテナがプールからEJBインスタンスを取り出し、呼び出しをディスパッチします。また、各メソッド呼び出しの前にトランザクションを開始します。メソッドの

呼び出しは、デフォルトの構成可能なアスペクトのチェーンによって処理され、最終的に実際のビジネス・ロジックが実行されます。アプリケーション・サーバーの外側で同じ機能を実装しようとするば、デコレータ・パターンや、AOPなどの汎用的なソリューションで実装することになるでしょう。

汎用デコレータ:インターセプタ

リスト1に示すように、この**Greeter** EJB Beanを直接サブレットにインジェクションした場合でも、呼び出しをデコレートできます。

インターセプタは、JDK 1.3で導入された**Proxy**クラスに似ていますが、より使いやすくなっています。必要なことは、1つのメソッドを持つ1つのクラスを実装することだけです。**リスト2**に、一般的なビジネス・メソッド・インターセプタを示します。

メソッド名は自由に決められますが、戻り型、例外、パラメータは**リスト2**に定義されているシグネチャと同じにする必要があります。インターセプタは、**@Interceptors**アノテーションがマネージドBeanまたはEJB Beanと関連付けられます。

普及には至らず
AOPは数年前に
話題となったもの
の、J2EEで盛んに
使用されるようには
なりません。それには十分な
理由があります。


```
@Stateless
@Interceptors
(NicenessExtender.class)
public class Greeter {
```

@Interceptors アノテーションの値はクラス配列です。宣言の順序によってインターセプタの実行順序が決まります。このアプローチではGreeter EJB BeanがNicenessExtenderに強く依存しますが、ほとんどの場合はこのアプローチが最良の選択肢です。デコレータの内容が明白であるため、意外なことが起きるおそれはありません。

最近のIDEでは、バイナリの直接結合を犠牲とすることで、EJB Beanからアスペクトを簡単に利用できます。

クラス・レベルで宣言した場合、すべての public メソッドがインターセプトされます。
@ExcludeClassInterceptors アノテーションが指定されたメソッドは、クラス・レベルで定義されたインターセプタによってインターセプトされません。同様に、**@ExcludeDefaultInterceptors** アノテーションは、XML で定義されたインターセプタを無効にします。

すべてのEJB Beanに適用されるデフォルトのインターセプタは、[WAR]/WEB-INFディレクトリに置かれたejb-jar.xmlデプロイメント・ディスクリプタで定義します(**リスト3参照**)。

ejb-jar.xmlファイルで宣言できるインターセプタは、デフォルトのインターセプタに限りません。Bean名やメソッド名を指定して、メソッドのインターセプトを細かく制御できます。ejb-jar.xmlファイルで

ポイントカットを見つける

AOPでは、**正規表現**、**カスタム言語**、**API**を使用して、ポイントカットを見つけることができますが、Java EE 6では通常、**アノテーション**または**XMLデプロイメント・ディスクリプタ**を使用してインターセプトを制御します

アノテーションをオーバーライドすることも可能です。これはステージング環境で有効です。本番環境の設定をejb-jar.xmlの設定で容易に上書きできます。その名前が示すとおり、ejb-jar.xmlファイルで構成できるのはEJB Beanだけです。

アノテーションとXMLの関係

結合度を下げ柔軟性を高めるために、**@InterceptorBinding**メタアノテーションを利用できます。**リスト4**に、属性を1つ持つカスタムの**@InterceptorBinding**を示します。

@InterceptorBindingで独自のアノテーションを作成し、それをインターセプトの制御に利用できます。その原理は**@Qualifier**と同様であり、**@InterceptorBinding**によって指定されるカスタム・アノテーションは、インターセプト対象のクラスに加えてインターセプタにも適用されます。解決プロセスでは、アノテーション自体に加えてその内部要素も考慮されます。

アノテーションの内部に要素を追加して複数のインターセプタを切り替えることもできます。**リスト5**を参照してください。これは

USUALインターセプタを示しています。

前述の**NicenessExtender**は**@Nice(USUAL)**アノテーションが指定されたすべてのクラスをインターセプトし、**PersonalNicenessExtender**は**@Nice(PERSONAL)**アノテーションが指定されたすべてのマネージドBeanとEJB Beanをインターセプトします(**リスト6**参照)。


@Niceアノテーションで適切なLevel列挙値を設定することで、複数のインターセプタ

リスト1 / リスト2 / リスト3 / リスト4 / リスト5 / リスト6

```
@WebServlet(name = "GreetingController", urlPatterns = {"/GreetingController"})
public class GreetingController extends HttpServlet {

    @Inject
    Greeter greeter;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<body>");
            out.println("<h1> " + greeter.greet() + "</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }
}
```

 [全てのリストをテキストで表示](#)

の順序を設定できます。このアノテーションを **Greeter** に適用する例は、次のとおりです。

```
@Nice(PERSONAL)
public class Greeter {}
```

インターセプタは、宣言するだけでは十分ではなく、アクティブ化も必要です。インターセプタをアクティブ化するには、それぞれをbeans.xml設定ファイルで宣言します。**リスト7**に、WEB-INF/beans.xmlによるインターセプタのアクティブ化を示します。

beans.xmlディスクリプタでの宣言の順序も、デコレータの順序を規定します。

@InterceptorBindingを使用すると、インターセプト対象のクラスがインターセプ

タ・クラスから完全に切り離されます。インターセプトタ・クラスとインターセプト対象のクラスは、**@InterceptorBinding** アノテーションのみに依存します。アノテーションが双方をリンクします。

ステレオタイプ・インターセプタ

JavaServer Faces (JSF) から **Greeter** にアクセスするには、**@Named** アノテーションによって JSF 式言語 (EL) で利用できるようにし、**@RequestScoped** などのスコープを定義する必要があります。JSF 統合のための明示的なアノテーションの使用例は、次のとおりです。

```
@Named
@RequestScoped
@Nice(PERSONAL)
public class Greeter {}
```

InterceptorBinding アノテーションをステレオタイプと組み合わせることもできます。それにより、アノテーション宣言の数を大幅に削減できます。次に、デコレートされたプレゼンタ・ステレオタイプを示します。

```
@Model
@Nice(PERSONAL)
@Stereotype
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface NicePresenter {}
```

@Stereotype の働きはマクロに似ています。**@Stereotype** に入れたすべてのアノテーションが展開されます。**@Stereotype** は単なるアノテーションのコンテナであり、他の意味はありません。展開は再帰的であるため、ステレオタイプに他のステレオタイプ・アノテーションを含めることができます。**リスト8**に、組み込みの **@Stereotype javax.enterprise.inject.Model** を示します。

前述の **NicePresenter** には **@Model** アノテーションが指定されていますが、これは、**@Named** アノテーションと **@RequestScoped** アノテーションを含む組み

みステレオタイプです (**リスト8** 参照)。また、ステレオタイプの場合、beans.xml ファイルでインターセプタをアクティブ化する必要があります。

横断的動作の分離

インターセプタのコードの大部分は、汎用的な呼び出しとリフレクションに関わるものです。**javax.interceptor.InvocationContext** パラメータは、対象となるメソッドとそのパラメータ、および追加のコンテキスト・データを含んでいます。これらは通常、呼び出しの前処理と後処理で使用されます。このメタデータ処理は通常再利用できず、テストが困難です。**リスト9**に、別のクラスへの機能の委譲を示します。


インターセプタは、対象となる Bean のスコープで実行されます。また、トランザクションに参加し、セキュリティ・コンテキスト内で実行され、ライフサイクルはインターセプト対象の Bean と同一です。再利用可能な横断的機能を、独立した再利用可能な Bean に簡単に分離して、インターセプタにインジェクションできます。**リスト10**に、再利用可能な監査クラスを示します。

再利用可能なクラスに対する特別な要件はありません。**リスト10**のような POJO でも、Contexts and Dependency Injection (CDI) マネージド Bean でも、EJB 3.1 Bean でもかまいません。インジェクションされたクラスには、現在の実行コンテキスト (トランザクション、セキュリティ、コンテキスト・データ) も渡されます。

@AroundInvoke アノテーションが指定されたインターセプタ・メソッドでは、**InvocationContext** の解釈に必要となる汎用コードのみを実装するべきです。再利用可能なアスペクトや機能は、別のメソッドや独立したクラスに分離します。それにより、再利用可能コードの単体テストやインターセプ

リスト7 / リスト8 / リスト9 / リスト10 / リスト11

```
<beans>
  <interceptors>
    <class>(...).aop.interceptors.NicenessExtender</class>
    <class>(...).aop.interceptors.PersonalNicenessExtender</class>
  </interceptors>
</beans>
```

 [全てのリストをテキストで表示](#)

タの実装自体に対する統合テストが単純になります。

いくつかの制約

「前」や「後」といったアドバイス (つまりインターセプト・ポイント) のある AOP フレームワークと異なり、インターセプタで可能なことは、メソッド全体を **@AroundInvoke** アノテーションで囲むことのみです。呼び出しの前後でカスタム・アクションを実行することはできますが、手動で **InvocationContext#proceed()** を呼び出すことで対象の呼び出しをトリガーする必要があります。

また、AOP フレームワークには、インターセプト・ポイント (AOP の用語ではポイントカット) を見つけるための強力なツールがあ

ります。AOP フレームワークでは、Java EE 6 とは異なり、メソッドだけでなくコンストラクタやフィールドもデコレートできます。Java EE 6 のインターセプタは、メソッドのデコレートのみに限定されています。

AOP では、正規表現、カスタム言語、API を使用してポイントカットを見つけることができますが、Java EE 6 では通常、アノテーション (**@Interceptor**、**@InterceptorBinding**、**@Stereotype**) か XML デプロイメント・ディスクリプタ (ejb-jar.xml と beans.xml) を使用してインターセプトを制御します。**リスト11**に、インターセプタのライフサイクル・コールバックを示します。

ただし、インターセプトの対象はビジネス・メソッドに限られません。また、ライフサ

イクル・コールバック・インターセプタによってオブジェクトの作成と破棄をインターセプトすることもできます(リスト11参照)。[@AroundInvoke](#)アノテーションを使用する代わりに[@PostConstruct](#)アノテーションや[@PreDestroy](#)アノテーションを使用すれば、インターセプト対象オブジェクトの作成後や破棄後に呼び出されるメソッドを指定できます。ライフサイクル・コールバック・メソッドでは、[InvocationContext](#)パラメータはオプションです。

不可能なことはない

これまで、通常のインターセプト方法について見てきました。Java EE 6では、サービス・プロバイダ・インタフェース(SPI)によって、デプロイメント時のイベントをインターセプトし、システムの構成を制御できます。アプリケーションのブート時に、型、アノテーション、beans.xmlファイルからCDIがメタモデルを構築します。

コンテナは、これを認識すると、メタ情報を含むイベントをスローします。このイベントは読取り専用ではありません。メタ情報に情報を加えたり、メタ情報を完全に書き換えたりすることができます。[@InterceptorBinding](#)などのアノテーションを追加したり削除したりすれば、Beanの検出やインジェクションに対してアクションを実行できます。カスタム・アノテーションへ応答したり、クラス名やクラスの場所などあらゆるトリガーに依存したインターセプタを追加することも簡単です。

CDI拡張を登録するには、JDKのJAR拡張メカニズムを使用します。ファイル名がこの拡張の名前([javax.enterprise.inject.spi.Extension](#))であり、内容として実装(拡張)の名前を含むファイルが必要です。この「拡張」ファイルを、WEB-INF/classes/META-INF/services/フォルダのWARファイル内

に置きます。[Extension](#)インタフェースは単なるマーカーであり、メソッドを実装する必要はありません。拡張はイベントをリスンし、自身の状態を変更します。リスト12に、[HiPrefixer](#)の動的な追加を示します。リスト12の[BootObserver](#)アノテーションは、すべての[ProcessAnnotatedType](#)イベントをリスンします。[ProcessAnnotatedType](#)イベントは、Javaクラス(通常はマネージドBeanまたはEJB Bean)を表す[AnnotatedType](#)インスタンスをラップしています。

残念ながら、[AnnotatedType](#)はイミュータブルな読取り専用クラスです。アノテーションを追加するには、リスト13に示すように、ミュータブルな[AnnotatedType](#)実装を独自に作成する必要があります。


起動時にコードを変更することなくカスタム・インターセプタを追加するには、追加の[InterceptorBinding](#)アノテーションである[@Hi](#)をブート時に加える必要があります。[UniversalGreeter](#) Beanに既に存在するアノテーションのリストに[@Hi](#)を追加します。[AnnotatedTypeWrapper](#)(リスト13)の[Set<Annotation> annotations](#)フィールドは、この目的のためのものです。[getAnnotations\(\)](#)メソッドは、カスタム・アノテーションとCDIランタイムで検出されたアノテーションを統合します。

アノテーションのインスタンスを直接作成することはできません。次のようなヘルパー・クラスを使用する必要があります。

```
public class HiInstance implements
Hi{
    @Override
    public Class<? extends
Annotation> annotationType() {
        return Hi.class;
    }
}
```

リスト12

```
public class BootObserver implements Extension{
    void processUniversalGreeter(@Observes ProcessAnnotatedType event) {
        AnnotatedType annotatedType = event.getAnnotatedType();
        Class javaClass = annotatedType.getJavaClass();
        if(javaClass.getName().endsWith("UniversalGreeter")){
            AnnotatedTypeWrapper atw = new AnnotatedTypeWrapper(annotatedType);
            atw.addAnnotation(new HiInstance());
            event.setAnnotatedType(atw);
        }
    }
}
```

 [全てのリストをテキストで表示](#)

[BootObserver](#)クラスでは、この[HiInstance](#)ヘルパーを、[@Hi](#)アノテーションの代わりに渡します。

デコレータでコードをきれいに

このインターセプタのサンプルには基本的な問題があります。インターセプタは汎用デコレータとしての使用に適していますが、このサンプルは特定のメソッド・シグネチャに依存しています。メソッドの戻り型が異なると([int](#)など)、実行時に[ClassCastException](#)が発生します。また、カスタムの横断的な機能で特定のメソッドをデコレートすると、通常は組立てが煩雑になります。リスト14は、メソッド固有の振る舞

いの(不恰好な)実装です。

リスト14のコードでは、インターセプタは、メソッド固有の振る舞いを適用するために、メソッド名やパラメータを調べる必要があります。メソッド固有のインターセプタをそれぞれのメソッドに適用することも可能ですが、デコレータ・パターンのタイプセーフ実装の方が、保守がはるかに簡単です。

メソッドに依存する横断的機能は、[@Decorator](#)を使用すればきれいに実装できます。CDIの[@Decorator](#)アノテーションは、タイプセーフ・インターセプタと見なすことができます。リスト15に、[@Decorator](#)を用いたメソッド固有のデコレートを示します。

このインタフェースを実装する場合、[@](#)

Decoratorの中ですべてのメソッドを実装する必要があります(**リスト15**参照)。アプリケーション・サーバーは、実際のクラスの代わりにこのデコレータをインジェクションします。これを可能にするために、このデコレータと**Greeter** Beanが共有するインタフェースを導入する必要があります。次のようにすると、**GreetDecorator**と**UniversalGreeter**の両方のクラスによってインタフェースが実装されます。

```
public class UniversalGreeter
implements Greeter {
```

Greeterインタフェースは、**UniversalGreeter**実現ではなく**GreetingController**にインジェクションされます。インターセプタの場合と同様に、次のようにしてbeans.xmlでデコレータをアクティブ化する必要があります。

```
<beans>
<decorators>
  <class>(...).aop.decorators.
  GreetDecorator</class>
</decorators>
</beans>
```

デコレータを抽象化すれば、**Greeter**インタフェースのすべてのメソッドを実装する必要がなくなります。**リスト16**に、抽象デコレータを用いた部分的なデコレータを示します。

デコレートするメソッドを選んで、他のメソッドを無視できます。メソッドのサブセットに対して複数の抽象デコレータを適用することで、さまざまなアスペク

リスト16

```
@Decorator
public abstract class GreetDecorator implements Greeter{
    @Inject @Delegate
    Greeter greeter;

    @Override
    public String greet() {
        return "Good " + greeter.greet();
    }
}
```

全てのリストをテキストで表示

トに対応できます。デコレータは、インターセプタとは異なり、**@Stereotype**を用いて適用することはできません。これはデコレートには不十分です。デコレータにはインタフェースが必要ですが、**@Stereotype**アノテーションではインタフェースを指定できません。

現実世界に戻って

Java EE 6では、インジェクション、デコレート、インターセプトに関連するほとんどすべての問題に対応できます。そのため、問題を精査して、カスタム拡張なしに可能かどうか検討することが重要です。

Java EEではすべてが可能なように思えますが、実際のプロジェクトで必要となるのは、ここで紹介したAOP機能の一部だけでしょう。実際、筆者はめったに**@InterceptorBinding**を使用せず、たいへん純粋な**@Interceptors**アノテーションを使っています。また、インターセプトするときはクラスのすべてのメソッドをインターセプトすることが多く、**@Interceptors**を利用するのは、組立てが容易になりそうな場合だけです。

Java EEでは、一般的な横断的機能の大部分がプラットフォームで提供されているため、明らかにAOPを使用すべき場面が見つかることはほとんどなく、したがってインターセプタやデコレータを使用する場面もめったに見つからないでしょう。</article>

LEARN MORE

- [JSR 330\(Dependency Injection for Java\)](#)
- [Seam Solder](#)
- [CODI](#)
- [CanDI](#)
- [Real World Java EE Night Hacks—Dissecting the Business Tier](#) (press.adam-bien.com, 2011)



GIVE BACK! ADOPT A JSR

JSRを検索





JULIEN PONGE



Part 2

GlassFishで実現するシンプルなクラスタリングと高可用性

GlassFishクラスタでアプリケーションの高可用性をサポートし、セッションのフェイルオーバーを実現

これはGlassFishクラスタリング機能に関する説明を行う、全2回シリーズのパート2の記事です。パート1では、GlassFishサーバの一元的なクラスタ・プロビジョニングと管理機能について紹介しました。この記事はGlassFishクラスタ中で、アプリケーションの高可用性を実現する方法について解説します。

注: GlassFishはGlassFish Server Open Source EditionとOracle GlassFish Serverの2つのエディションがありますが、本記事に記載する内容は何れのエディションにも適用可能です。

パート1では、アプリケーションのパート1では本番環境で発生する、アプリケーションの高可用性と負荷軽減といった2つの異なる問題について説明しました。そしてこれらの問題は、単一サーバによる稼働ではなく、複数台のサーバで水平スケーラビリティを持つクラスタ環境を構築し稼働させることによって解決する事を紹介しました。

注: この記事で紹介するサンプルのソース・コードは[こちら](#)からダウンロードできます。

TaskEE: ステートフル・アプリケーション

パート1で示したClockEEサンプルのアプリケーションは、GlassFishクラスタリング機能の一元的な管理を説明する際には有効なアプリケーションでした。しかしこのサンプルは実行時の日時を表示するだけで、セッション状態を保持しないため、高可用性機能の説明には適していません。そこで本記事では新たにTaskEEというアプリケーションを導入します(図1参照)。

TaskEEはタスク管理を行うための簡単なWebアプリケーションで、タスクの追加や削除を行うことができます。リレーショナルデータベースなどの永続化機能は利用しません。状態の伝播を単純化して説明するため、ユーザのタスク・リストはWebのセッション情報から特定することにします。従って、仮に同一のマシン上で異なる2つのWebブラウザのプロセスを起動した場合も、異なる2つのタスク・リストを管理できます。

TaskEE は非常に簡単な Web アプリケーションで、**リスト1**に示すセッション・スコープの Context and Dependency Injection (CDI) Bean でタスク・リストを定義しています。

CDI を利用し、セッション・スコープを指定したクラスのインスタンスに対してインジェクションすると、透過的に

Servlet のセッション・コンテキストに関連付けられます。リスト2に示すように、コントローラとして単一のサーブレットを実装し、その中でインジェクションし CDI に委譲します。そしてビューの処理を行うために JavaServer Pages に対してセッション・スコープを持つ **TaskList** のインスタンス(**リスト**)を渡します。

その他のサーブレットと JavaServer Pages のコードはここでは省略します。全ソース・コードの一覧は TaskEE フォルダに含まれています。

メモリを使用したセッション状態レプリケーション

アプリケーションをクラスタ化したインスタンスに対してデプロイした場合、デフォルトでは実行時の状態情報はそれぞれのインスタンス内でのみ保持され、共有される事はありません。とりわけ Web のセッションに含まれる状態情報、もしくはステートフル Enterprise Beans (EJB) はローカルにのみ保持されます。

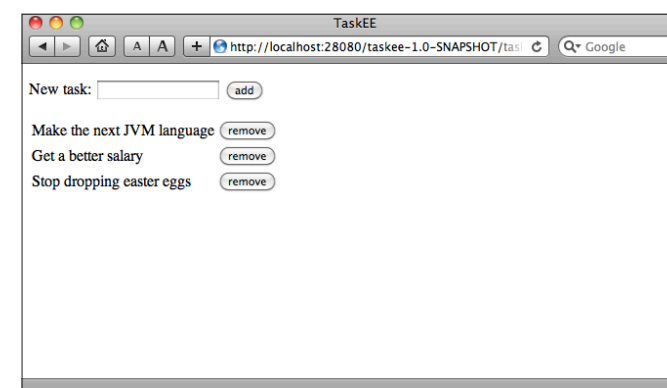


図1

クラスタで高可用性もしくは負荷分散を実現する事はとても有用です。そしてこれを実現するためには、ある特定の受信したリクエストがクラスタ中に含まれる任意のインスタンスで実行できる必要があります。

受信リクエストを任意のインスタンスで処理できる事によって、メンテナンス目的の計画停止が必要な場合、もしくはサービスを提供するインスタンスで何らかの不具合が発生した場合に、フェイルオーバーを実現できるようになります。従って、アプリケーションのセッション情報を透過的にフェイルオーバーができるように、クラスタのインスタンス間でセッション情報を複製する必要があります。GlassFish はHTTP セッションとステートフル EJB の状態デー

タの両方をクラスタのインスタンス間で複製できます。これを実現するのは [Shoal グループ管理サービス \(GMS\)](#) で全インスタンスが参照します。このグループ管理サービスは1対多の通信にはマルチキャスト UDP (User Datagram Protocol) を使用し、1対1の通信に関しては [Grizzlyプロジェクト](#) の成果物を利用しています。またこのサービスによって状態変更に関するイベントを効率的に伝播します。さらに特定のグループ中で変動するインスタンス数を管理し、インスタンスの有効・無効時それぞれにおいて復旧、フェールオーバーを実現します。

GMS はセッション情報の複製がどのインスタンスで保持されているかを把握するため、[コンシステンス・ハッシュアルゴリズム](#) を使用しています。このアルゴリズムはマルチキャスト・ブロードキャストによるネットワークトラフィックを軽減します。初期の GlassFish のバージョンではこれよりも簡単な実装を行っており、特定のインスタンスがグループ内の全インスタンスに対してルックアップ要求をブロードキャストしていました。

GlassFish における高可用性機能の実現方法について理解するため、HTTP のセッションについて検討します。セッションは各リクエストおよびレスポンスで受け渡されるクッキーを使用して特定し、サーバ側で保持されています。パフォーマンスの観点でリクエストは常に同一インスタンスで処理されます(実現方法は後述)。

特定のHTTP セッションに格納されているオブジェクトはクッキーの値(キーとして使用)を用いて取り出します(GlassFish の場合 **JSESSIONID**)。仮にインスタンスが停止した場合、クラスタ中に存在する他のアクティブなインスタンスでリクエストを継続処理でき

るようにフェールオーバーします。

クッキーはフロント・エンドのサーバによって管理され、さらにクッキーのドメインポリシーが正しいか否か適切に管理されています。フェールオーバー後、新しくリクエストを受け付けたインスタンスは、ローカルキャッシュ中に存在するセッション情報を参照するか、もしくは他のインスタンス中に存在するセッション情報を参照し、以前と同様の HTTP セッションデータを利用できます。

GlassFish で高可用性を実現するための技術要件は下記のとおりです。

- インスタンスは同一サブネットワーク内での存在が必須
(同一サブネットに存在しない場合、GMS のマルチキャスト UDP接続が不可)
- eb アプリケーションの **web.xml** ディスクリプタ中に **<distributable>** 要素が必要
- HTTP セッション、ステートフル EJB に保存するオブジェクトは [java.io.Serializable](#) の実装が必要
(実装しない場合、状態は他のインスタンスに対して伝播不可)
- クラスタの全インスタンスとアプリケーションはデプロイ時に高可用性の有効化が必要
- 不整合を避けるため、インスタンスの物理ホストではクロックをできる限り同期させる必要があります。不整合をさけるため、インスタンスが稼働する全物理ホストの時刻同期が必要。
興味深いことに、Java EE アプリケーションの高化用性の実現要件の殆どは、インスタストラクチャに関する内容です。レプリケーションする Java クラスをシリアライズ化したり、**web.xml** ディスクリプタに対して特定のXML要素を含める必要はありますが、アプリケーションの実装コードの観点では、それ以外変更する必要はありません。

リスト1

リスト2

リスト3


```
@Named
@SessionScoped
public class TaskList implements Serializable {

    private List<String> tasks = new ArrayList<String>() {
        {
            add("Wash the dishes");
            add("Invent the next JVM language");
            add("Get a better salary");
        }
    };

    public List<String> getTasks() {
        return unmodifiableList(tasks);
    }

    public void add(String task) {
        tasks.add(task);
    }

    public void remove(String task) {
        tasks.remove(task);
    }
}
```

 [全てのリストをテキストで表示](#)

高化用性に対応するクラスタ構成

クラスタ管理を行うために便利なコマンドがいくつかあります。**リスト3**はクラスタに存在する全インスタンスの稼働・停止状況に関する情報を表示するヘルスチェック用のコマンドです。

リスト 4は、構築したクラスタ環境が同一ネットワークに存在する要件を満たすか否かを確認するコマンドです。各インスタンス間の通信で一部UDPのマルチキャストのソケットを使用するため、セッション情報のレプリケーションを適切に行うためには、各インスタンスが同一のネットワーク上に存在しなければなりません。このコマンドは、全ホスト上で同様に実行し確認する必要があります。**-bindaddress** パラ

メータは省略可能ですが、同一ホスト上で複数のネットワーク・インタフェースを持つ場合に、どの NIC を使用するかを指定するような場合に有効です。コマンドを実行すると、クラスタに所属する全インスタンスが稼働するホストから応答が得られます。仮に特定のホストから応答が得られない場合、未応答の理由を調査する必要があります。実際に GMS 通信が正常に動作しないために、セッション情報のレプリケーションができない場合があります。

クラスタ環境を構築するため、最後にグローバルで高可用性を有効にするプロパティの設定を行います。この設定は個々のインスタンス・レベル、もしくはコンテナ・レベルで設

準備万端

Java EEアプリケーションを高可用性に対応させるための要件は、ほとんどがインフラストラクチャに関するものです

定をカスタマイズできます。例えば Web コンテナでは有効にし、EJB コンテナでは無効にするような設定も可能です。ほとんどの場合、全インスタンスおよび全コンテナで高可用性をサポートしますが、この場合クラスターレベルで設定を有効にします。プロパティ名は、**<cluster name>-config.availability-service.availability-enabled** のような文字列となります。本サンプルにおける設定例は**リスト5**のようになります。

TaskEEの高可用性の有効化

以上で、セッション情報をレプリケーションによって高可用性を実現するクラスタ環境が適切に設定されました。次に**リスト6**に示すように TaskEE のアプリケーションをデプロイします。

デプロイのコマンドは、単一インスタンスに対するデプロイと同様ですが、通常の引数に加え **--availabilityenabled** フラグを追加し true に設定して実行します。このフラグは名前からもわかるとおり、DAS に対してセッション・レプリケーションとフェールオーバーのサポートを設定し、設定後、GMS、EJB コンテナ、Web コンテナに対して設定情報を通知します。

スモークテスト(予備の簡易テスト)として**図1**に示すように、クラスタ内の各インスタンスに対して、TaskEE のアプリケーションに関連付けられたルートコンテキストの URL に対して、HTTP 接続を行い、いくつかのタスクの追加・削除を行ってください。同じホストまたはノードで稼働しているインスタンスでは、セッション情報が同期される一方、別のインスタンスに対して接続すると、異なるタスクリストを持つ新しいセッションが作成されるはずですが、しかし今までの設定で HTTP セッションレプリケーションは正しく設定したのではないのでしょうか。もちろん正しく設定されています、ホストによってセッションが異なる振る舞いをするのはバグではなく正常な動作です。実際、HTTP セッションの情報はサーバ側で保持されている事を思い

だしてください。そして各 HTTP クライアントのリクエスト内にクッキーの値をキーとして含め、サーバとクライアント間で交換しています。クッキーは非常に重要な情報で、サーバ・プロバイダの管理下でないサーバとのやり取りは許されていません。そのためクッキーは特定のドメインやサブドメインに制限されます。たとえば、**localhost**、**stardust**、**caramoustra** 上で稼働する、それぞれの TaskEE アプリケーションに接続した場合、各ホスト名が異なるためセッション情報は異なります。

ロード・バランサの追加

アプリケーションを利用するユーザは直感的に、一つのホストもしくは URL に対して接続します。実際にはクラスタ内の各インスタンス上でアプリケーションを実行していますが、その事実は利用者であるユーザに対しては透過的に見せる必要があります。これを実現するためロードバランサを使用します。簡単にいうと、ロードバランサはフロント・エンドで稼働する HTTP サーバであり、稼働中のインスタンスに対してリクエストをディスパッチする役割を持っています。また各リクエストに対するレスポンスを HTTP クライアントに対して応答する役割も持っています。

クッキーは単一ホストと交換しますが、統一的なセッション情報の格納領域が共有されるため、セッション・レプリケーションが可能となります。同様のことがステートフル EJBにも適用されますが、EJB のリクエストは通常、HTTP セッション自身の結果と関連づけられます。ロードバランサのリクエスト・ディスパッチ方法は様々な手法があります。簡単な手法としてはランダムな振り分け方法、もしくはラウンド・ロビン方式などです。より複雑な手法としては、特定の HTTP クライアントからのリクエストを常に同じインスタンスに対してディスパッチする方法があります。この方法ではディスパッチされていたインスタンスが停止した場合にのみ、フェー


リスト4

リスト5

リスト6

```
$ asadmin validate-multicast --bindaddress 192.168.56.1
Will use port 2048
Will use address 228.9.3.1
Will use bind interface 192.168.56.1
Will use wait period 2,000 (in milliseconds)
```

```
Listening for data...
Sending message with content "infinity.home" every 2,000 milliseconds
Received data from infinity.home (loopback)
Received data from ubuntu-vm.home
Received data from infinity.home
Exiting after 20 seconds. To change this timeout, use the --timeout command line option.
Command validate-multicast executed successfully.
```

 [全てのリストをテキストで表示](#)

ルオーバーで振り先を変更します。これはスティッキー・ロードバランスと呼ばれています。スティッキー・ロードバランスは同じインスタンスをターゲットとして接続するため最大限セッション情報を特定のローカルに保持することができる利点があります。これに対してラウンド・ロビン方式やランダムによるディスパッチ方式では、より多くのレプリケーション処理が必要になるためクラスタ中に存在するインスタンス間のネットワークトラフィックが増加します。

GlassFish サーバとフロント・エンドに存在する HTTP サーバの接続設定はとても簡単です。一般的には Apache HTTPD と **mod_jk** と呼ぶ特別なモジュールで AJP プロトコルを使用して接続します。AJP を利用すると静的コンテンツの提供といった特定のタスクは直接 Apache HTTPD で処理し、その他の処理を GlassFish のインスタンスで処理することができます。

また、Oracle GlassFish Server のライセンスを購入し商用サポートを受けている利用者は、[専用のロードバランサ・プラグイン](#)を入手

する事ができるようになります。このプラグインは、IzPack ベースのインストーラで、簡単に Apache HTTPD、Microsoft IIS、Oracle iPlanet Web server と接続するための設定ができます。また、全体的に導入設定手順が簡単で、フロント・エンドの HTTP サーバと GlassFish クラスタ間をセキュアに統合する事も可能です。

GlassFish Open Source Edition の利用者は通常、[Apache HTTPD](#) と **mod_jk** の組み合わせによる連携設定を推奨します。一方で Apache HTTPD 以外にもロードバランサとして振る舞う事が可能な HTTP サーバは数多く存在します。**mod_jk** を使用した設定は世の中に数多く存在するため、この記事では Apache HTTPD の代わりに Lighttpd を使用した設定方法を紹介します。Lighttpd は堅牢で軽量な HTTP サーバで、人気のある Web サイトでの稼働実績もあり、設定ファイルの構文も単純でわかりやすいため非常に魅力的です。

テスト用に [lighttpd.conf](#) に対して設定した内容を**リスト7**に示します。このロードバランサの設定は 1981 番ポートでリクエストを待

ち受け、クラスタ中に含まれる3つのインスタンスに対して振り分けを行っています。この設定ファイルは簡単なため説明の必要はないでしょう。

このハッシュ・ロードバランサの方法は、特定のHTTPクライアントからのリクエストを継続して同一インスタンスに対してディスパッチします(インスタンスが稼働している間)。

Lighttpdがリクエストをディスパッチ中にインスタンスに対して接続できない事を検知した場合、Lighttpd は自動的に別のインスタンスを選択し、セッション・フェールオーバを引き起こします。

下記のコマンドを実行して Lighttpd をロードバランサとして起動します。

```
$ lighttpd -D -f lighttpd.conf
2011-07-01 15:23:59: (log.c.166)
```

フェールオーバの動作

上記でロードバランサの設定が完了したため、**図2**に示すように、クライアントのブラウザから1981 番ポートに接続する事でアプリケーションの動作確認ができます。また、クライアントからのリクエストを各インスタンスが正常

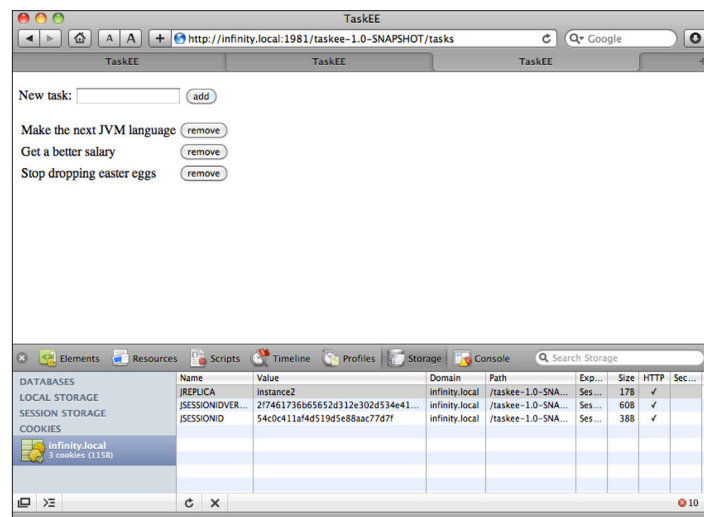


図2

に処理する事も確認できます。一般的な Web ブラウザ中で使用可能な Web インспекタを使用して、ロードバランサがどのインスタンスに対して最初にディスパッチしているかヒントを得る事ができます。クッキー名 JREPLICA にインスタンス名が記載されています。この例では **instance2** が記載されています(**図2**参照)。この状態で instance2 を停止させ、ページの再読み込みを行うか、もしくはタスクの追加・削除といった操作を行う事でフェールオーバを動作させることができます。そして残る2つのインスタンスがセッション情報を失う事なくリクエストに対する処理を継続します。さらに、インスタンスの起動・停止を繰り返し、フェールオーバ・メカニズムやその効果を確認する事ができます。

トラブルシューティングに関する注意:

稼働中のインスタンスが停止して別のホスト上でインスタンスが稼働するようになった場合に、セッション情報が引き継がれない問題が発生する場合があります。ほとんどの場合は、ネットワーク、ロードバランサ、GlassFish サーバの特定の構成で、異なるドメインのクッキーを発行してしまうことによって引き起こされています。これを避けるため、HTTP ロードバランサのパブリック URL として期待するクッキーのドメイン名を GlassFish サーバ上で明示的に指定する事ができます。これを実現するためには、アプリケーション中で、**WEB-INF/glassfish-web.xml** を用意し**リスト8**に示すようなコードを記述する必要があります(接続例:<http://my.wonderfulapp.com>)。

まとめ

この記事で GlassFish のクラスタリングに関するシリーズは終了です。最初の記事はクラスタの一元的なプ

リスト7

リスト8

```
server.document-root = "."
server.port = 1981
server.modules += ( "mod_proxy" )
proxy.balance = "hash"
proxy.server = ( "" =>
(
(
"host" => "127.0.0.1",
"port" => 28080
),
(
"host" => "127.0.0.1",
"port" => 28081
),
(
"host" => "192.168.56.101",
"port" => 28080
)
)
)
```

 [全てのリストをテキストで表示](#)

ロビジョニングと管理について説明しました。そしてこの記事ではアプリケーションの高可用性とセッション・フェールオーバの有効化について説明しました。また GlassFish クラスタのフロント・エンドで稼働する HTTP ロードバランサの設定についても若干触れました。ここで説明した Lighttpd による設定は、既存ドキュメントは少ないですが、テストが容易で一般性を損なわない便利な方法です。また今回の記事ではステートフル EJB のセッション・レプリケーションについては取り扱いませんでしたが、全てにおいてこの記事で記載した内容と同様に適用可能です。GlassFish サーバは開発用途で便利なサーバであるだけでなく、パフォーマンスと高可用性の両方が求められる、水平スケーラビリティが必要な本番環境にも十分適用可能な本格的なサーバです。またク

ラスタのインスタンスに対する管理や一元的なプロビジョニングを行うための洗練された扱いやすいツールも備わっています。</article>

LEARN MORE

- [“GlassFish Server Load Balancing Plug-in Installation and Setup”](#)
- [Beginning Java EE 6 with GlassFish 3 by Antonio Goncalves \(Apress, 2010\)](#)
- [The Java EE 6 Tutorial: Basic Concepts, fourth edition \(Prentice Hall, 2010\)](#)

