

GlassFish v4 ではじめる Java EE 7 ハンズオン・ラボ

Version 1.0

*Yoshio Terada, Java Evangelist
<http://yoshio3.com>, @yoshioterada*

Table of Contents

1.0 はじめに	3
2.0 NetBeans と GlassFish の動作確認	6
3.0 JavaServer Faces アプリケーションの作成	11
4.0 JMS アプリケーションの作成	35
5.0 WebSocket アプリケーションの作成	55
6.0 GlassFish クラスタ環境の構築と動作確認.....	74
参考資料・補足	91

1.0 はじめに

Java EE 7 は Java EE 6 をベースに 3 つの新しいテーマ（開発生産性の向上、HTML 5 対応、エンタープライズ・ニーズへの対応）を提供し、4 つの新機能が追加されました。本ハンズオン・ラボでは新機能の一つである WebSocket にフォーカスをあて、より現実的な WebSocket アプリケーションの構築を行います。本ハンズオン・ラボを終了することで、セキュアでより大規模な WebSocket アプリケーションの構築ができるようになります。

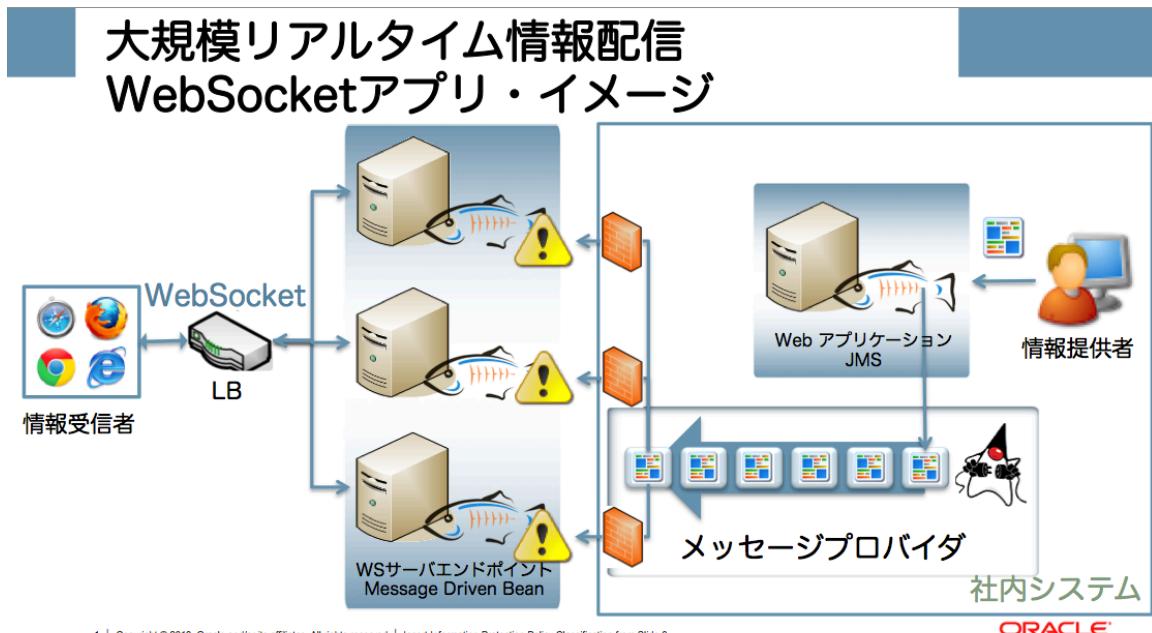


図 1：ハンズ・オン・アプリケーションの概念図

本ハンズオン・ラボは Java EE 7 に含まれる下記の技術を使用します。

- Java Message Service 2.0 (JSR 343)
- JavaServer Faces 2.2 (JSR 344)
- Enterprise Java Beans (JSR 345)
 - Singleton EJB
 - Message-Driven Bean
- Contexts and Dependency Injection 1.1 (JSR 346)
- Java API for WebSocket 1.0 (JSR 356)

これらの API を使用することでより大規模でセキュアな WebSocket アプリケーションの構築ができます。

必須ソフトウェア

- 最新の JDK 7 を入手してください。
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Java Platform, Standard Edition								
<p>Java SE 7u45 This release includes important security fixes. Oracle strongly recommends that all Java SE 7 users upgrade to this release. Learn more →</p> <p>Which Java package do I need?</p> <ul style="list-style-type: none">▪ JDK: (Java Development Kit). For Java Developers. Includes a complete JRE plus tools for developing, debugging, and monitoring Java applications.▪ Server JRE: (Server Java Runtime Environment) For deploying Java applications on servers. Includes tools for JVM monitoring and tools commonly required for server applications, but does not include browser integration (the Java plug-in), auto-update, nor an installer. Learn more →▪ JRE: (Java Runtime Environment). Covers most end-users needs. Contains everything required to run Java applications on your system. <table border="1"><thead><tr><th>JDK DOWNLOAD ↓</th><th>Server JRE DOWNLOAD ↓</th><th>JRE DOWNLOAD ↓</th></tr></thead><tbody><tr><td>JDK 7 Docs<ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations</td><td>Server JRE 7 Docs<ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations</td><td>JRE 7 Docs<ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations</td></tr></tbody></table>			JDK DOWNLOAD ↓	Server JRE DOWNLOAD ↓	JRE DOWNLOAD ↓	JDK 7 Docs <ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations	Server JRE 7 Docs <ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations	JRE 7 Docs <ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations
JDK DOWNLOAD ↓	Server JRE DOWNLOAD ↓	JRE DOWNLOAD ↓						
JDK 7 Docs <ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations	Server JRE 7 Docs <ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations	JRE 7 Docs <ul style="list-style-type: none">▪ Installation Instructions▪ ReadMe▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations						

図 2 : JDK のダウンロード

Java EE 7 Hands-on Lab using GlassFish 4

- NetBeans 7.4 以降を入手してください。NetBeans のダウンロードサイトより「Java EE」もしくは「すべて」のパッケージを選択し「ダウンロード」ボタンを押下してください。「Java EE」、「すべて」を選択した場合、本ハンズ・オンの動作に必要な GlassFish v4 がバンドルされています。

<http://netbeans.org/downloads/>

サポートテクノロジー *	Java SE	Java EE	C/C++	HTML5 & PHP	すべて
④ NetBeansプラットフォームSDK	●	●			●
④ Java SE	●	●			●
④ Java FX	●	●			●
④ Java EE		●			●
④ Java ME					—
④ HTML5		●		●	●
④ Java Card(TM) 3 Connected					—
④ C/C++			●		●
④ Groovy					●
④ PHP				●	●
バンドル サーバー					
④ GlassFish Server Open Source Edition 4.0		●			●
④ Apache Tomcat 7.0.41		●			●

図 3 : NetBeans バンドル版のダウンロード

2.0 NetBeans と GlassFish の動作確認

NetBeans をインストールしたのち、アイコンをダブル・クリックし NetBeans を起動してください。起動すると下記のような画面が表示されます。

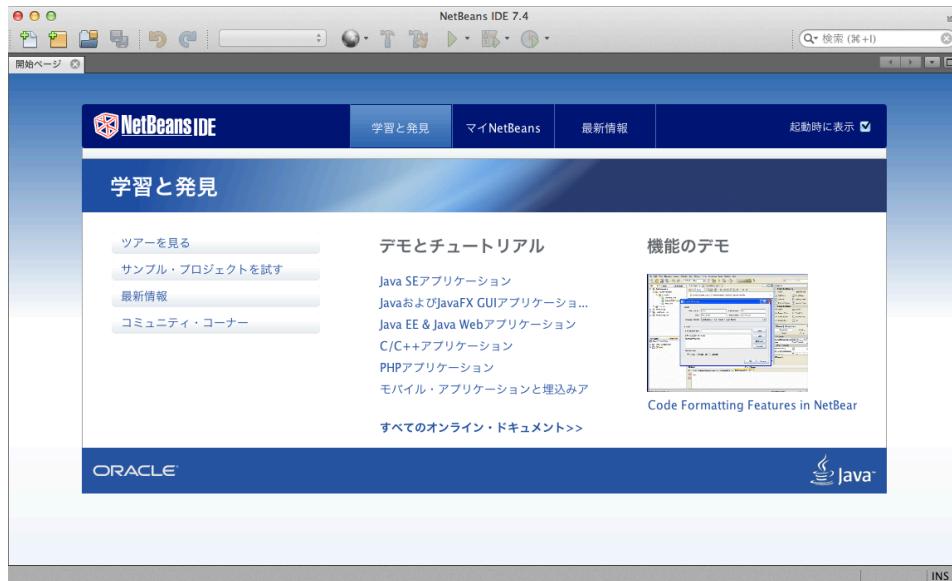


図 4：NetBeans の起動画面

ここで、NetBeans と GlassFish が正しく連携できているかを確認するため、新しくプロジェクトを作成してください。メニューから「ファイル (F)」→「新規プロジェクト(W)...」を選択してください。

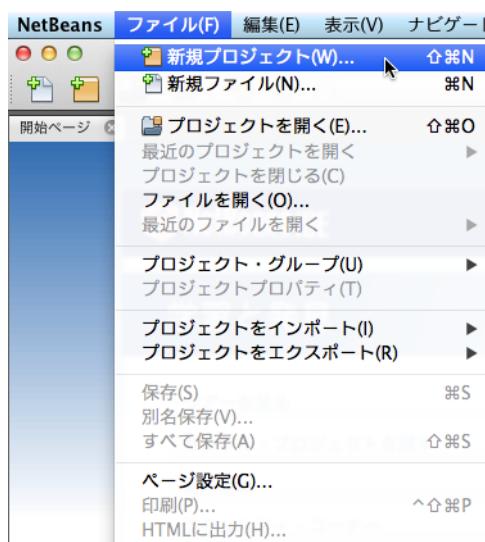


図 5：新規プロジェクト作成

Java EE 7 Hands-on Lab using GlassFish 4

「新規プロジェクト(W)...」を選択すると下記のウィンドウが表示されます。
「カテゴリ(C) :」より「Java Web」を選択し、「プロジェクト(P):」より
「Web アプリケーション」を選択し「次へ >」ボタンを押下してください。

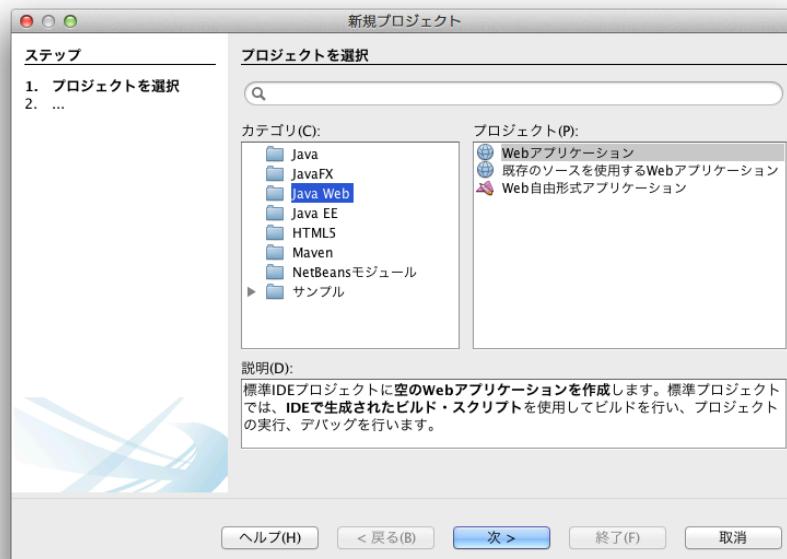


図 6：新規プロジェクト作成

「次へ >」ボタンを押下すると下記のウィンドウが表示されます。「プロジェクト名：」に「WebSocket-HoL」と入力し「次へ >」ボタンを押下してください。

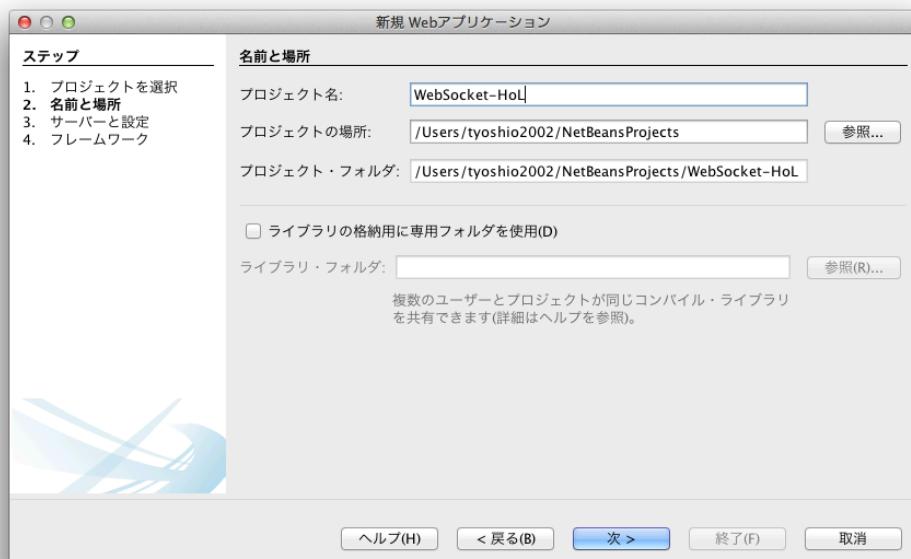


図 7：新規 Web アプリケーション

Java EE 7 Hands-on Lab using GlassFish 4

「次へ >」ボタンを押下すると下記の画面が表示されます。「コンテキスト・パス(P):」の変更は今回不要なので、そのまま「次へ >」ボタンを押下してください。

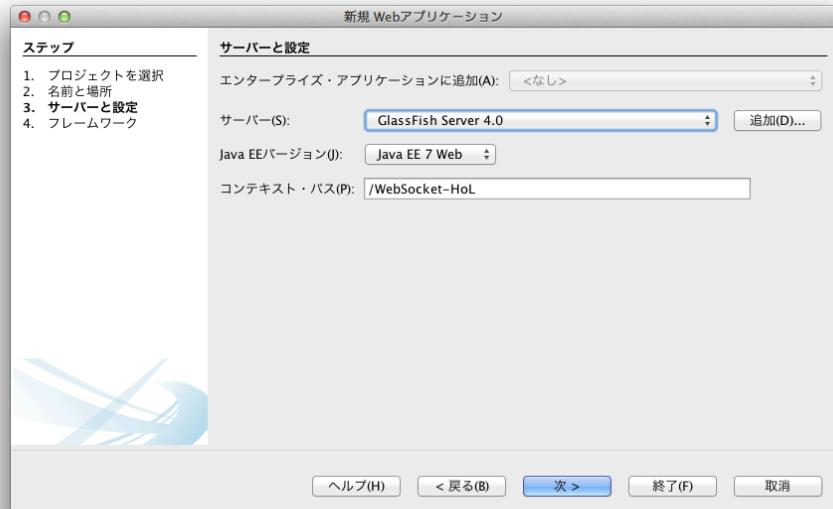


図 8：新規 Web アプリケーション

「次へ >」ボタンを押下すると、下記のウィンドウが表示されます。使用するフレームワークとして「JavaServer Faces」を選択し、最後に「終了(F)」ボタンを押下してください。

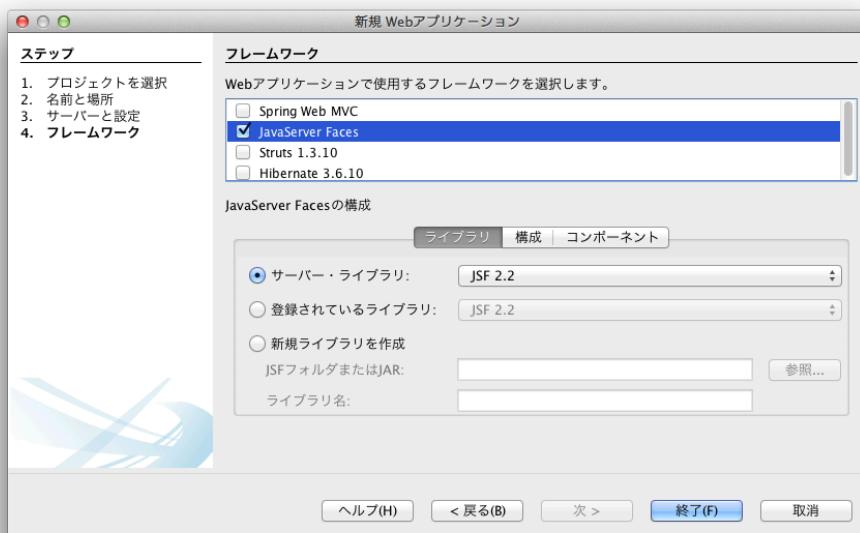


図 9：新規 Web アプリケーション

Java EE 7 Hands-on Lab using GlassFish 4

「終了(F)」ボタンを押下すると下記の画面が表示されます。

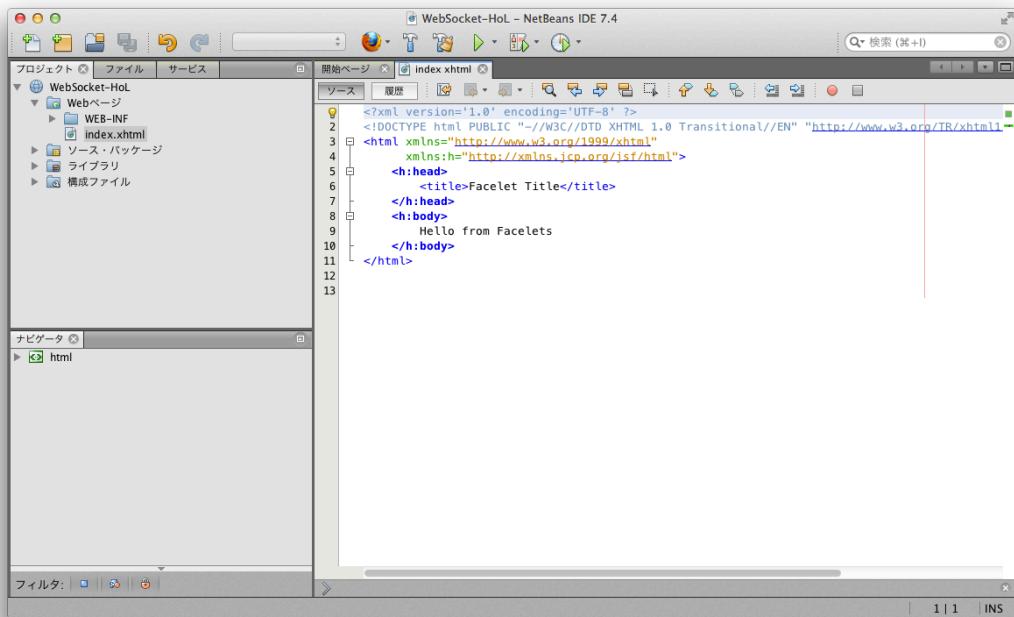


図 10：プロジェクト作成完了画面

新規プロジェクトを作成したので、このプロジェクトを実行します。プロジェクトを実行するために、「図 11：NetBeans ツールバー」に示す NetBeans のツールバーより、「プロジェクトを実行(F6)」ボタンを押してください。



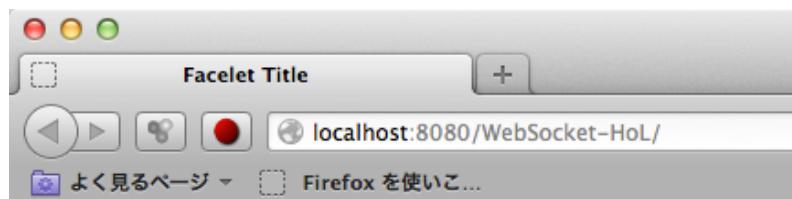
図 11：NetBeans ツールバー

Java EE 7 Hands-on Lab using GlassFish 4

もしくは、NetBeans のメニューから「プロジェクトを実行(R)」を選択し実行してください。



図 12 : NetBeans プロジェクトの実行



Hello from Facelets

図 13 : NetBeans の実行結果

プロジェクトを実行するとブラウザが自動的に起動しデフォルトのページが表示されます。

3.0 JavaServer Faces アプリケーションの作成

本章より実装をはじめていきます。

まず、情報提供者が Web アプリケーションに対して情報を登録する画面を JavaServer Faces (以降 JSF) で実装します。

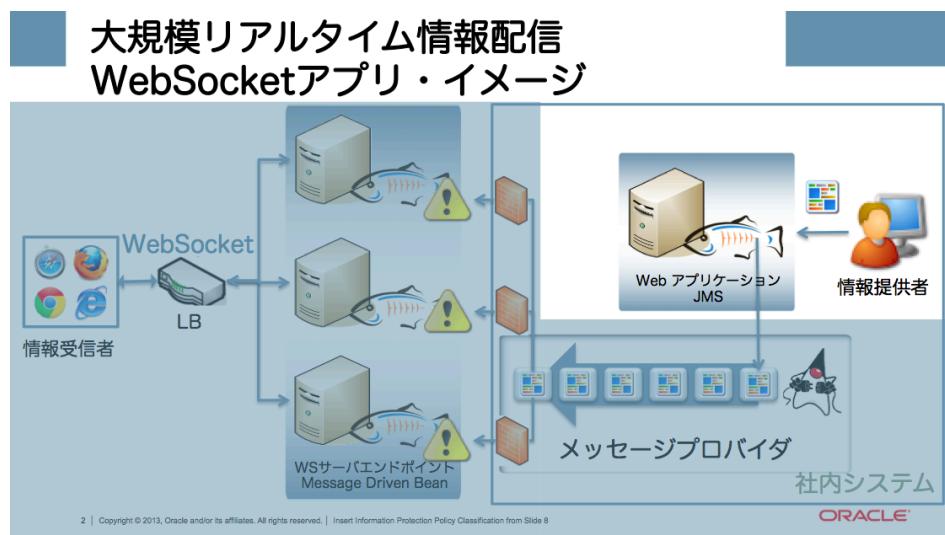


図 14 : JSF アプリケーションの概念図

JSF は Web アプリケーションを構築するための標準 Web フレームワークで、Visual Basic や JavaFX などのリッチクライアント開発に取り入れられているコンポーネント指向開発を取り入れています。

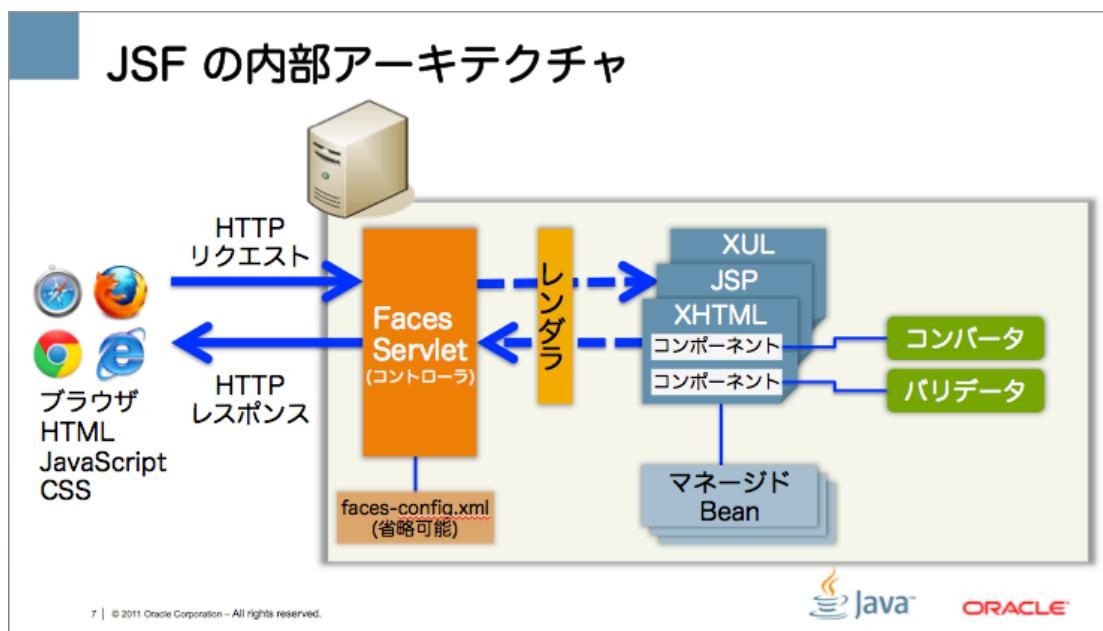


図 15 : JSF の内部アーキテクチャ

JSF は MVC アーキテクチャに基づき実装を行うことが可能で、JSF の実行環境が提供する FacesServlet が MVC におけるコントローラの役割を担います。FacesServlet は事前設定ファイル (faces-config.xml) を読み込み、クライアントからのリクエストに対して適切な処理へマッピングする役割を持っています。開発者は基本的に、コントローラである FacesServlet に対して設定ファイルの修正以外、特に処理を追加実装する必要はありません。開発者が JSF のアプリケーションで実際に実装する箇所はビューとモデル部分になり、ビューは Facelets(xhtml) と EL 式、モデルは Context Dependency Injection (以降 CDI) を用いて実装します。

Facelets では HTML で提供されている HTML タグと 1 対 1 で対応する独自タグを xhtml ファイル中に記載していきます。例えばテキスト・フィールドを表す HTML タグは <INPUT TYPE="TEXT" value=""> ですが、Facelets では <h:inputText id="***" value="" /> で表します。

■ HTMLと対応するJSF (Facelets) タグ

名前		テキストフィールド <code><h:inputText></code> タグ
HTML タグ	JSF タグ	
<code><INPUT TYPE="TEXT" value=""></code>	<code><h:inputText id="username" value="" /></code>	
性別		ラジオボタン <code><h:selectOneRadio></code> タグ
HTML タグ	JSF タグ	
<code><input type="radio" name="sex" value="1"> 男性</code> <code><input type="radio" name="sex" value="2"> 女性</code>	<code><h:selectOneRadio id="sex" value="sex"></code> <code><f:selectItem itemValue="1" itemLabel="男性" /></code> <code><f:selectItem itemValue="2" itemLabel="女性" /></code> <code></h:selectOneRadio></code>	

11 | © 2011 Oracle Corporation - All rights reserved.

Java ORACLE

図 16 : Facelets タグの利用例

ご参考：JSF 2.2 で利用可能な Facelets タグの一覧は下記に記載されています。

<http://docs.oracle.com/javaee/7/javaserverfaces/2.2/vdldocs/facelets/>

Facelets タグを用いて画面デザインを作成し、画面デザイン完了後、画面の入力データ等をプログラム内で扱えるように致します。

Facelets+EL式によるCDIとのバインディング

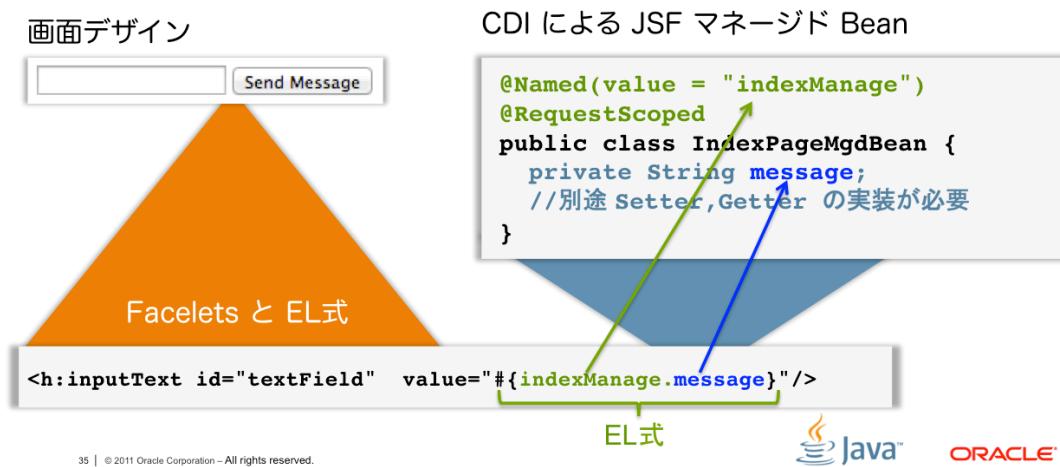


図 17 : Facelets + EL 式と CDI のバインディング

テキスト・フィールド内に入力された値をプログラム内で取得するためには、Facelets 内に定義する EL 式（上記の図では #{indexManage.message} と記載されている箇所）を用いて、CDI で実装したクラス（上記の図では IndexPageMgdBean クラス）にバインディング（結びつけ）します。

画面デザインに示すような HTML のテキスト・フィールドを表現するために Facelets では <h:inputText /> タグを定義します。ここでタグ内の value に記載する箇所 "#{indexManage.message}" が EL 式で、@Named のアノテーションを付加した CDI のクラス（IndexPageMgdBean）にバインドし、ドットで連結し、フィールド（message）にバインドしています。

バインドされた値を取得するためには CDI で別途定義するゲッター・メソッド（上記の場合 getMessage() メソッド）を呼び出し取得することが可能です。

このように JSF は HTML コンポーネントの値を、POJO の Java クラスに対してバインドして簡単にデータを取り出す仕組みを提供し Web アプリケーションの簡易化をはかっています。JSF は、実装内部ロジックは多少複雑ですが、Web アプリケーションの開発者にその内部アーキテクチャの複雑さを隠蔽し、かんたんに開発ができる仕組みを提供しています。

本ハンズオンでは JSF の Facelets タグと EL 式、CDI を利用して簡単な JSF アプリケーションの作成を行います。このハンズ・オンではかんたんに実装するために、画面遷移、入力データ検証、Ajax 化などは行いません。ユーザから入力されたデータをサーバ側で受けて処理をする JSF アプリケーションを作成します。

まず、JSF の管理対象 Bean を Context Dependency Injection (以降 CDI) で実装するため CDI を有効にします。次に JSF のパラメータで日本語データを扱うことができるよう、GlassFish で UTF-8 の文字コードを扱う設定を行います。

はじめにプロジェクトをマウスで選択してください。選択したのち、右クリックし「新規...」→「その他...」を選択してください。

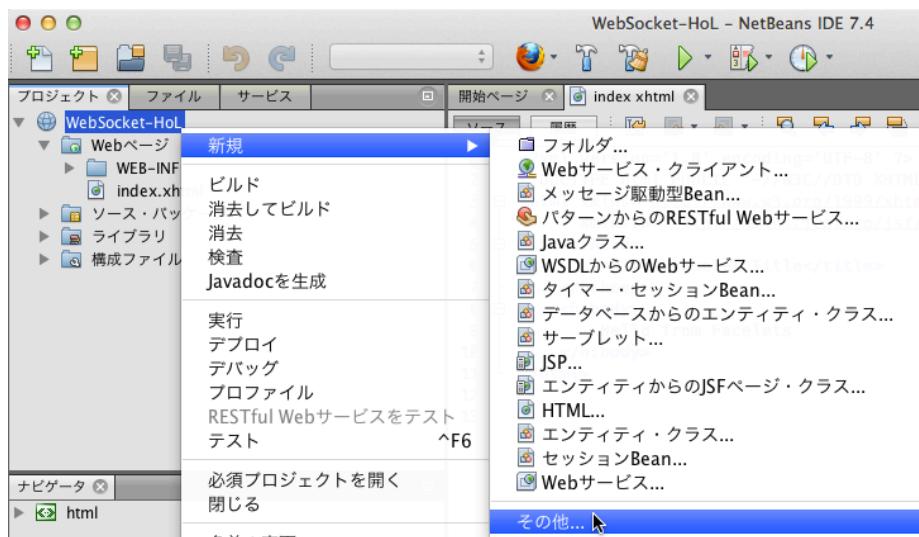


図 12 : CDI の有効化

選択すると下記のウィンドウが表示されます。ここで「カテゴリ(C):」より「コンテキストと依存性の注入」を選択し、「ファイル・タイプ(F):」より「beans.xml (CDI 構成ファイル)」を選択し、「次へ >」ボタンを押下してください。

Java EE 7 Hands-on Lab using GlassFish 4

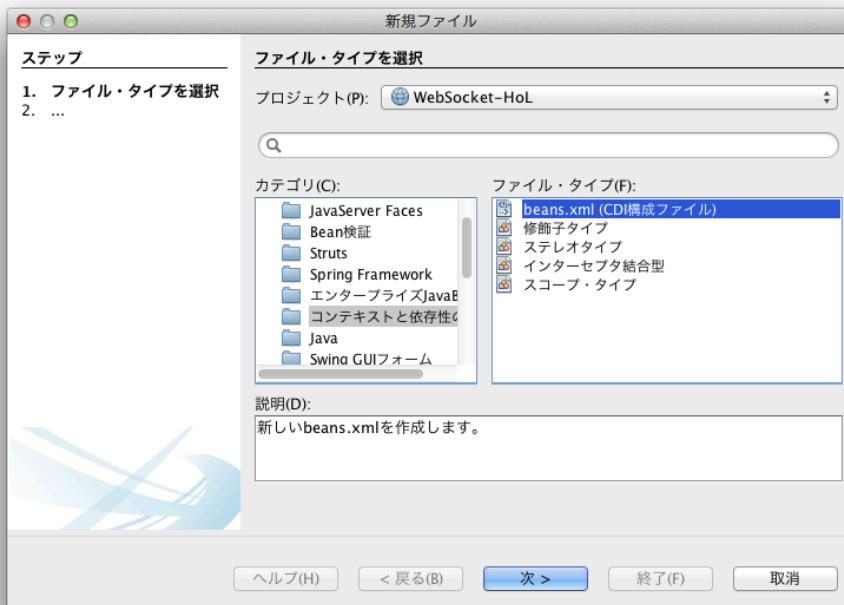


図 19：新規ファイル

ボタンを押下すると下記のウィンドウが表示されます。ここではデフォルトの設定のまま「終了(F)」ボタンを押下してください。



図 20：New beans.xml (CDI 構成ファイル) ウィンドウ

ボタンを押下すると「構成ファイル」配下に下記の beans.xml ファイルが自動的に生成されます。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="annotated">
</beans>
```

ここで、デフォルトで記載されている「**bean-discovery-mode="annotated"**」の箇所を **bean-discovery-mode="all"**¹ に修正してください。修正後は下記になります。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">
</beans>
```

次に、GlassFish で UTF-8 の文字エンコードを扱う設定を行います。プロジェクトをマウスで選択し右クリックし「新規」→「その他...」を選択してください。

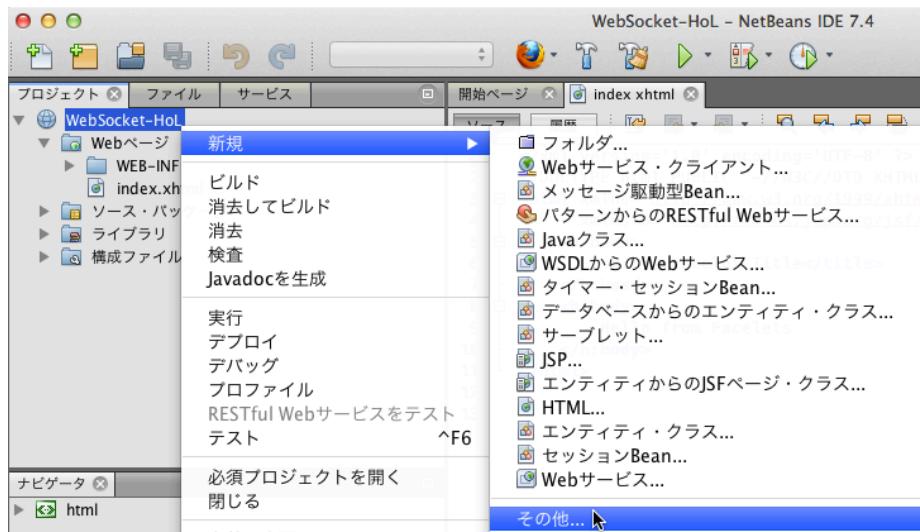


図 21：新規 GlassFish ディスクリプタの作成

¹ bean-discovery-mode は “all”, “annotated”, “none” のいずれかを指定できます。annotated を指定した場合、アノテーションが付加されたクラスに対してのみインジェクション可能です。all を指定した場合、全てのクラスに対してインジェクション可能となります。

次に「カテゴリ(C):」より「GlassFish」を選択し、「ファイル・タイプ(F):」より「GlassFish ディスクリプタ」を選択し「次へ >」ボタンを押下してください。

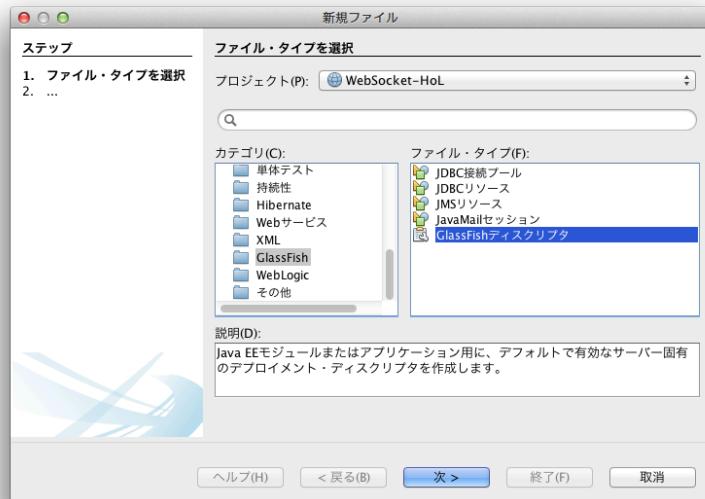


図 22：新規 GlassFish ディスクリプタの作成

ボタンを押下すると下記のウィンドウが表示されます。ここではデフォルトの設定のまま「終了(F)」ボタンを押下してください。

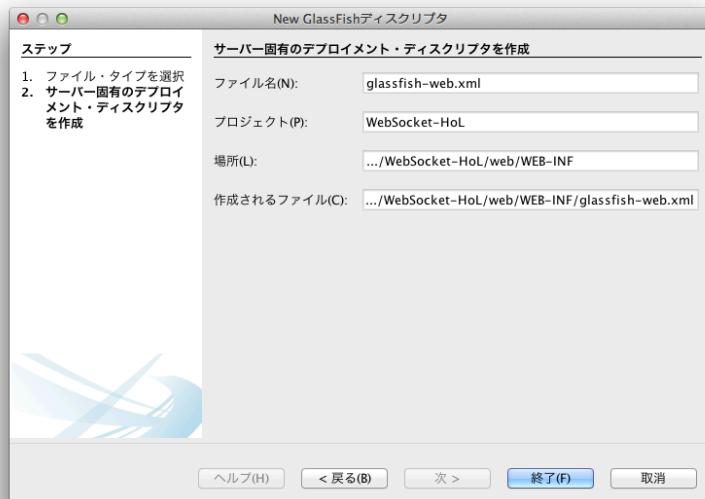


図 23：新規 GlassFish ディスクリプタの作成

Java EE 7 Hands-on Lab using GlassFish 4

ボタンを押下すると下記の画面が表示されます。ここで「XML」ボタンを押下してください。



図 24 : glassfish-web.xml の編集画面

押下すると下記の XML ファイルが表示されます。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC
  "-//GlassFish.org//DTD GlassFish Application Server 3.1
  Servlet 3.0//EN" "http://glassfish.org/dtds/glassfish-
  web-app_3_0-1.dtd">
<glassfish-web-app error-url="">
  <class-loader delegate="true"/>
  <jsp-config>
    <property name="keepgenerated" value="true">
      <description>
        Keep a copy of the generated servlet class' java code.
      </description>
    </property>
  </jsp-config>
</glassfish-web-app>
```

ここで、GlassFish でデフォルトで UTF-8 文字エンコードを扱うために、
`<parameter-encoding default-charset="UTF-8" />` の 1 行を追加してください。追加すると下記になります。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC
  "-//GlassFish.org//DTD GlassFish Application Server 3.1
  Servlet 3.0//EN" "http://glassfish.org/dtds/glassfish-
  web-app_3_0-1.dtd">
<glassfish-web-app error-url="">
  <class-loader delegate="true"/>
  <jsp-config>
    <property name="keepgenerated" value="true">
      <description>
        Keep a copy of the generated servlet class' java code.
      </description>
    </property>
  </jsp-config>
  <parameter-encoding default-charset="UTF-8" />
</glassfish-web-app>
```

以上で、CDI と UTF-8 の文字エンコードが利用可能になりました。

プロジェクトの「構成ファイル」ディレクトリ配下を確認すると下記のファイルが含まれてますので確認してください。

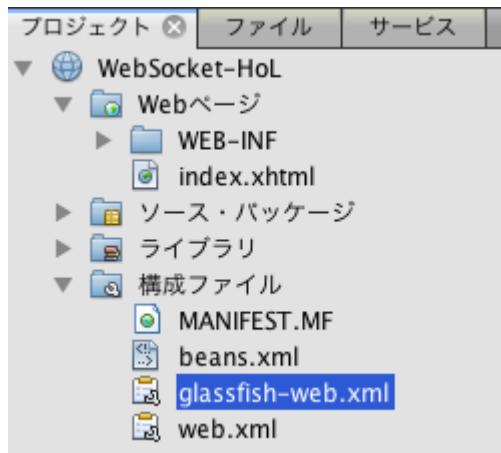


図 25：構成ファイルの確認

Java EE 7 Hands-on Lab using GlassFish 4

次に、JSF ページのバックエンド処理を行う、「JSF 管理対象 Bean」を CDI として作成します。プロジェクトを選択し、右クリックした後、「新規」→「その他...」を選択してください。

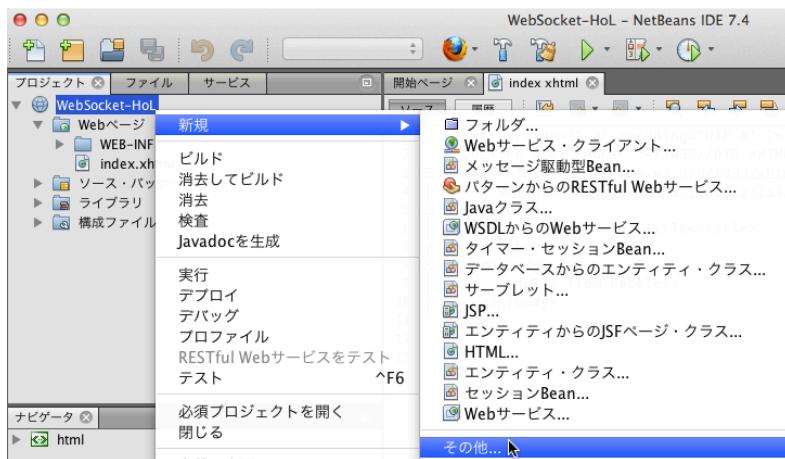


図 26：新規 JSF バッキングビーンの作成

選択すると、下記のウィンドウが表示されます。ここで「カテゴリ(C):」より「JavaServer Faces」を選択し、「ファイル・タイプ(F):」より「JSF 管理対象 Bean」を選択した後、「次へ >」ボタンを押下してください。

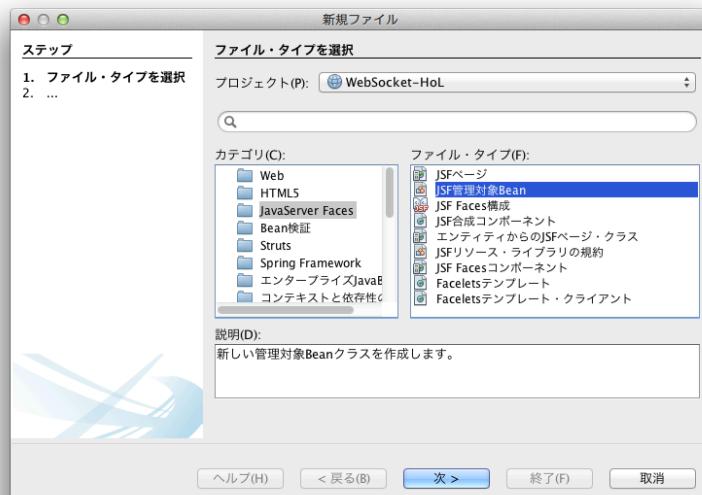


図 27：新規 JSF 管理対象ビーン

Java EE 7 Hands-on Lab using GlassFish 4

ボタンを押下すると下記のウィンドウが表示されます。ここで、「クラス名(N):」に「IndexPageMgdBean」を入力し、「パッケージ(K):」に「jp.co.oracle.cdis」、「スコープ:」を「dependent」から「request」に変更し、最後に「終了(F)」ボタンを押下してください。

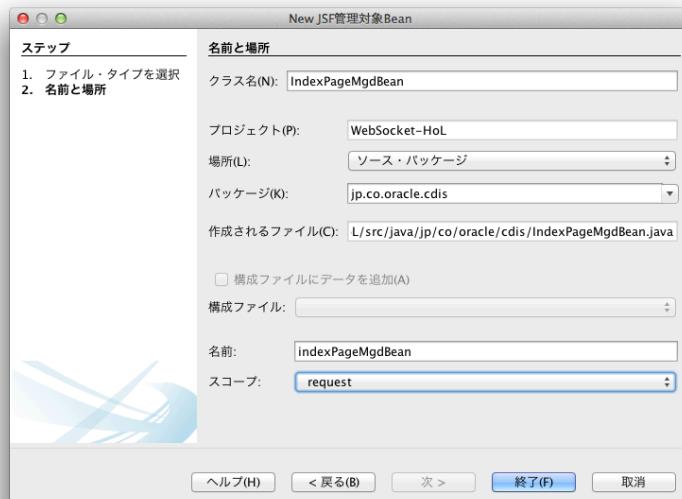


図 28 : New JSF 管理対象 Bean

ボタンを押下すると下記のコードが自動的に生成されます。

```
/*
 * To change this license header, choose License Headers in
 * Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package jp.co.oracle.cdis;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

/**
 *
 * @author *****
 */
@Named(value = "indexPageMgdBean")
@RequestScoped
public class IndexPageMgdBean {

    /**
     * Creates a new instance of IndexPageMgdBean
     */
    public IndexPageMgdBean() {
    }
}
```

ここで、CDI のデフォルト名を変更します。@Named(value = “**indexPageMgdBean**”)と記載されている箇所を、@Named(value = “**indexManage**”)に修正してください。また、情報提供者から入力された文字をプログラムにバインド（結びつけ）する変数を定義します。「private String message;」を定義してください。

```
/*
 * To change this license header, choose License Headers in
 * Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package jp.co.oracle.cdis;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

/**
 *
 * @author *****
 */
@Named(value = "indexManage")
@RequestScoped
public class IndexPageMgdBean {

    private String message;

    /**
     * Creates a new instance of IndexPageMgdBean
     */
    public IndexPageMgdBean() {
    }
}
```

次に、NetBeans のメニューより「リファクタリ...」→「フィールドをカプセル化...」を選択してください。

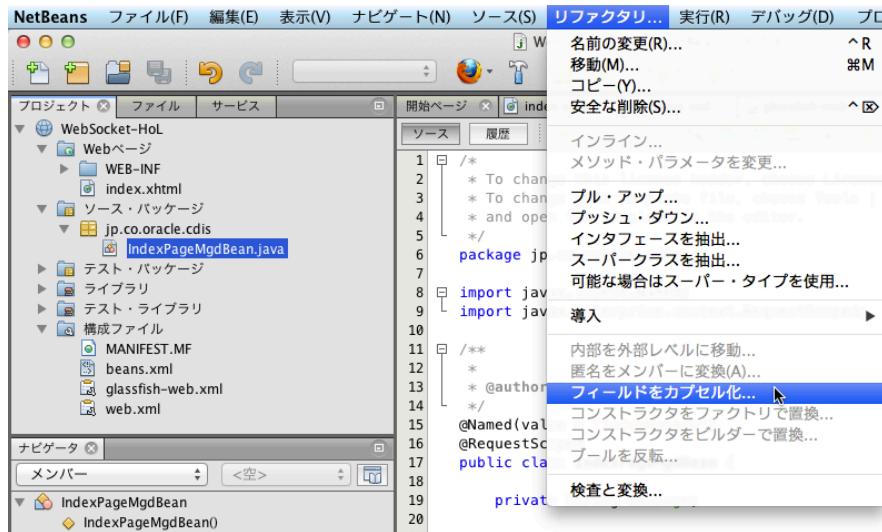


図 29：フィールドのカプセル化

選択すると下記のウィンドウが表示されます。ここで、「カプセル化するフィールド一覧(L):」から「取得メソッドを作成」、「設定メソッドを作成」のそれぞれに含まれる「getMessage」、「setMessage」にチェックし「リファクタリング(R)」ボタンを押下してください。



図 30：フィールドをカプセル化

ボタン押下すると下記のコードが自動的に生成されます。

```
/*
 * To change this license header, choose License Headers in
 * Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package jp.co.oracle.cdis;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

/**
 *
 * @author *****
 */
@Named(value = "indexManage")
@RequestScoped
public class IndexPageMgdBean {

    private String message;

    /**
     * Creates a new instance of IndexPageMgdBean
     */
    public IndexPageMgdBean() {
    }

    /**
     * @return the message
     */
    public String getMessage() {
        return message;
    }

    /**
     * @param message the message to set
     */
    public void setMessage(String message) {
        this.message = message;
    }
}
```

上記のバインド変数を定義する事により、Expression Language(EL) 式を通じて、JSF のページ内からデータ参照、データ設定ができるようになります。²

次に、データが入力された際に行う実際の処理を実装します。ここではページ内でボタンを一つ用意しボタンが押下された際に処理する内容を実装します。今回は、JSF の振る舞いを簡単に理解するために、ボタンを押下された

² JSF の Facelets でデータ・バインドに EL 式 #{indexManage.message} を定義できるようになります。例えば、テキスト・フィールドで message を参照、設定するためには、JSF タグ <h:inputText id="textField" value="#{indexManage.message}" /> を記入します。

際に pushSendButton() メソッドを呼び出し、入力された文字をそのまま標準出力に表示する処理を実装します。クラスに対して下記のメソッドを追加してください。³

```
public String pushSendButton() {
    System.out.println(getMessage());
    return "";
}
```

実装した全ソースコードは下記となります。

```
package jp.co.oracle.cdis;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

/**
 *
 * @author *****
 */
@Named(value = "indexManage")
@RequestScoped
public class IndexPageMgdBean {

    private String message;

    /**
     * Creates a new instance of IndexPageMgdBean
     */
    public IndexPageMgdBean() {
    }

    /**
     * @return the message
     */
    public String getMessage() {
        return message;
    }

    /**
     * @param message the message to set
     */
    public void setMessage(String message) {
        this.message = message;
    }
    public String pushSendButton() {
        System.out.println(getMessage());
        return "";
    }
}
```

³ JSF のページからこのメソッドを実行するためには、ボタンタグで action に EL 式を定義します。<h:commandButton value="Send Message" action="#{indexManage.pushSendButton()}" /> ボタンが押下された際に pushSendButton() メソッドが実行されます。またアクションを実行した際、そのメソッドの戻り値が画面遷移先を表します。メソッドの返り値を空文字「""」としているのは本処理を実行した後、画面遷移を行わないと空文字「""」としています。

次に、管理者が接続するページを JSF の Facelets で作成します、プロジェクトを選択したのち、右クリックし「新規」→「その他...」を選択してください。

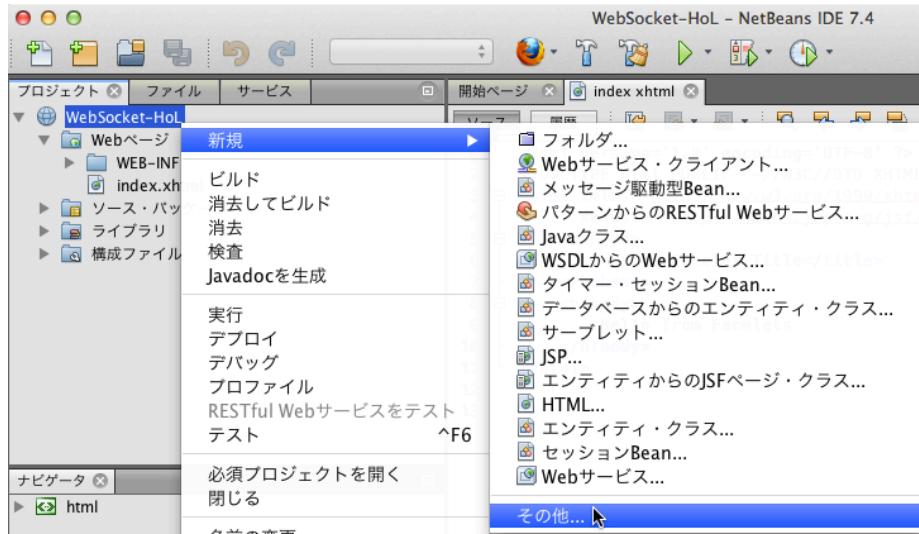


図 31：新規 Web ページの作成

選択すると下記のウィンドウが表示されます。ここで「カテゴリ(C):」より「その他」を選択し、「ファイル・タイプ(F):」から「フォルダ」を選択して「次へ >」ボタンを押下してください。

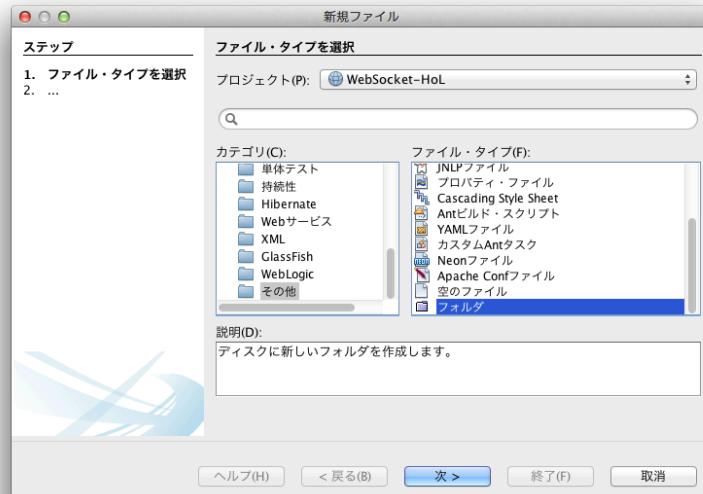


図 32：新規フォルダの作成

Java EE 7 Hands-on Lab using GlassFish 4

ボタンを押下すると下記のウィンドウが表示されます。ここで「フォルダ名(N):」に「admin」、「親フォルダ(R):」に「web」を記載し、最後に「終了(F)」ボタンを押下してください。



図 33 : New フォルダ作成

ボタンを押下すると、プロジェクトの「Web ページ」ディレクトリ配下に「admin」ディレクトリが作成されます。

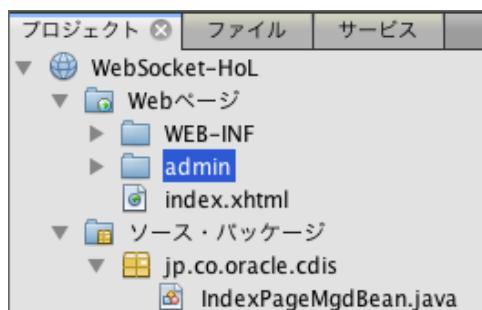


図 34 : ディレクトリ構成

次に、作成した「admin」ディレクトリをマウスで選択した後、右クリックしてください。右クリックした後「新規」→「その他...」を選択してください。

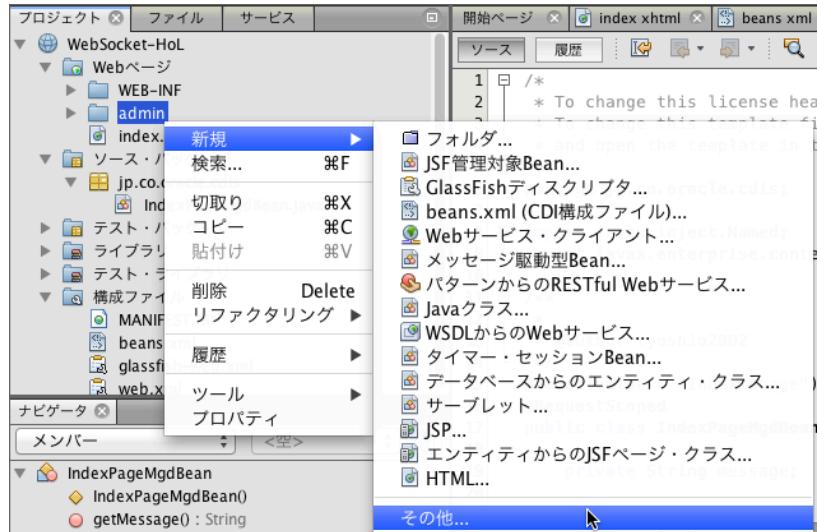


図 35：新規 Web ページ作成

選択すると下記のウィンドウが表示されます。ここで「カテゴリ(C) :」より「JavaServer Faces」を選択し、「ファイル・タイプ(F) :」から「JSF ページ」を選択し「次へ」ボタンを押下してください。

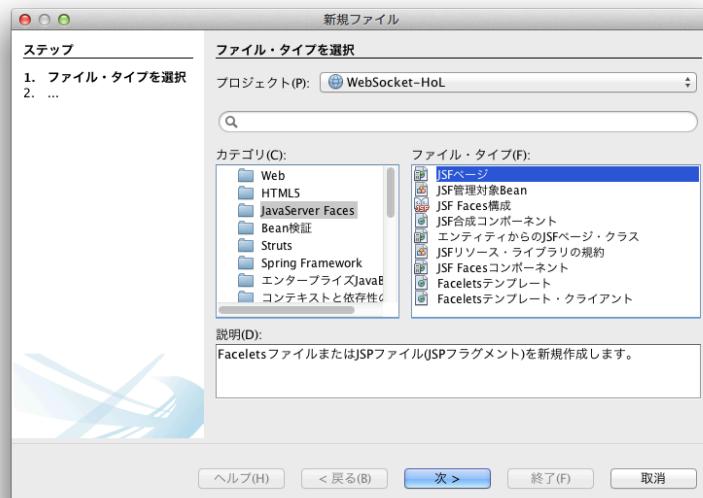


図 36：新規 JSF ページの作成

Java EE 7 Hands-on Lab using GlassFish 4

ボタンを押下すると下記のウィンドウが表示されます。ここで、「ファイル名：」に「index」を入力し、最後に「終了(F)」ボタンを押下してください

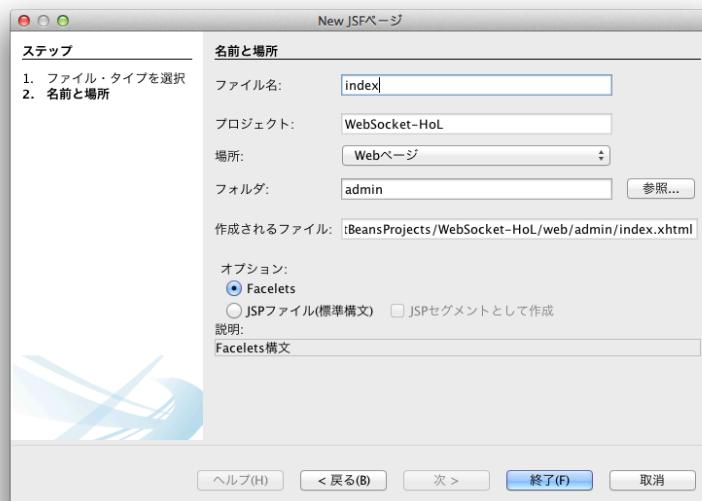


図 37 : New JSF ページ

ボタンを押下すると自動的に下記のファイルが生成されます。

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Facelet Title</title>
  </h:head>
  <h:body>
    Hello from Facelets
  </h:body>
</html>
```

次に、プロジェクトを実行した際に、この作成したページがデフォルトで表示されるようにプロジェクトの設定を修正します。プロジェクトの「構成ファイル」ディレクトリ配下に存在する「web.xml」ファイルをダブル・クリックしてください。

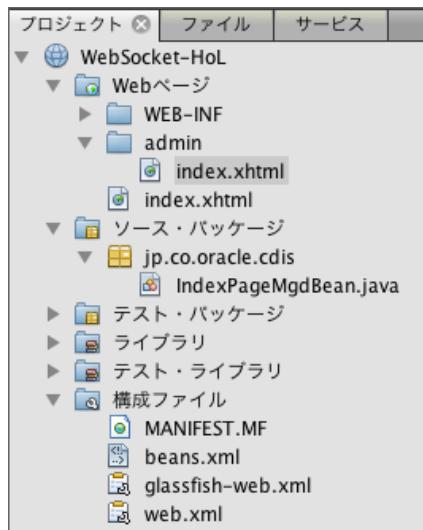


図 38：構成ファイルの修正

ダブル・クリックすると「web.xml」ファイルの設定ウィンドウが表示されます。ここで「ソース」ボタンを押下すると下記のような画面が表示されます。

```

    ソース 一般 サーブレット フィルタ ページ 参照 セキュリティ 検索
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
3      <context-param>
4          <param-name>javax.faces.PROJECT_STAGE</param-name>
5          <param-value>Development</param-value>
6      </context-param>
7      <servlet>
8          <servlet-name>Faces Servlet</servlet-name>
9          <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10         <load-on-startup>1</load-on-startup>
11     </servlet>
12     <servlet-mapping>
13         <servlet-name>Faces Servlet</servlet-name>
14         <url-pattern>/faces/*</url-pattern>
15     </servlet-mapping>
16     <session-config>
17         <session-timeout>
18             30
19         </session-timeout>
20     </session-config>
21     <welcome-file-list>
22         <welcome-file>faces/index.xhtml</welcome-file>
23     </welcome-file-list>
24 </web-app>
25

```

図 39：web.xml の設定画面

Java EE 7 Hands-on Lab using GlassFish 4

ここで、XML 要素 <welcome-file> の項目を、下記のように修正し admin 配下の index.xhtml が呼び出されるように修正してください。

```
<welcome-file-list>
    <welcome-file>faces/admin/index.xhtml</welcome-file>
</welcome-file-list>
```

上記設定完了後、プロジェクトを実行すると、デフォルトで admin ディレクトリ配下の index.xhtml が呼び出されるようになります。

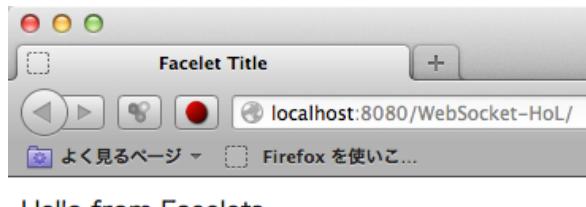


図 40：設定完了後の実行画面

それでは、実際に Web ページを作成していきます。今回は、JSF の Facelets で作成する画面のイメージは下記になります。

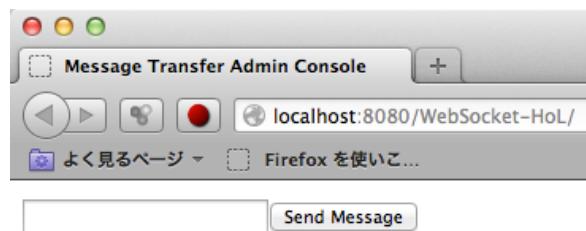


図 41：画面完成イメージ

上記の画面で、「Send Message」のボタンが押下された際、テキスト・フィールドに入力された文字列をサーバ側に送信するコードを実装します。自動生成された XHTML ファイルを下記のように修正してください。

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Message Transfer Admin Console</title>
    </h:head>
    <h:body>
        <h:form>
            <h:inputText id="textField"
                         value="#{indexManage.message}" />
            <h:commandButton value="Send Message"
                             action="#{indexManage.pushSendButton()}" /><br/>
        </h:form>
    </h:body>
</html>
```

実装した後、NetBeans のプロジェクトを実行してください。



図 42 : NetBeans プロジェクトの実行

Java EE 7 Hands-on Lab using GlassFish 4

プロジェクトを実行すると下記の画面が表示されます。

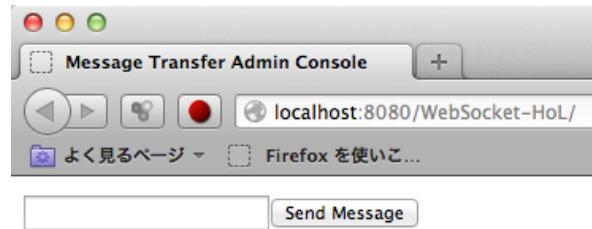


図 43：実行画面

テキスト・フィールドに「こんにちは」と入力し「Enter」キー、もしくは
「Send Message」ボタンを押下してください。

Java EE 7 Hands-on Lab using GlassFish 4

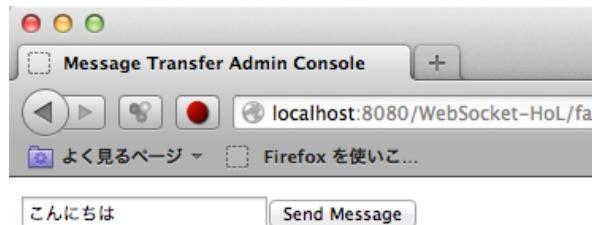


図 44：アプリケーションの実行

実行後、NetBeans の「GlassFish 4.0」のコンソールを確認すると「こんにちは」の文字列が表示されていることを確認できます。

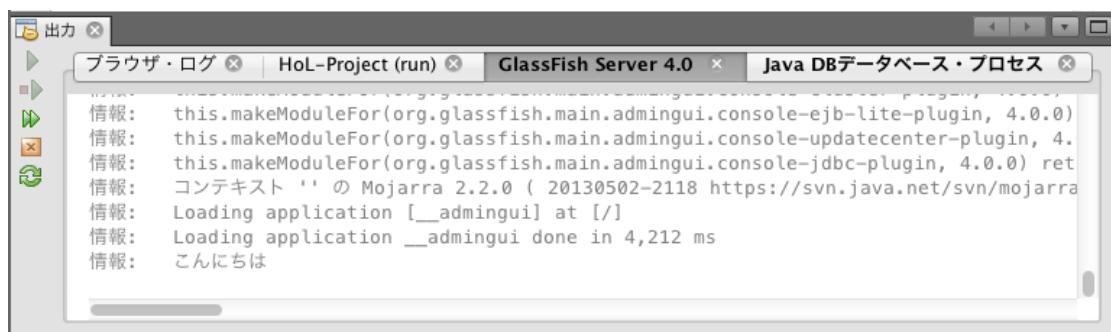


図 45：GlassFish コンソールの確認

以上で JSF の Facelets によるアプリケーションの作成は完了です。JSF では上記のように、CDI と EL 式を組み合わせることで、かんたんにユーザの入力をサーバ側で取り扱えます。

4.0 JMS アプリケーションの作成

次に、情報提供者が JSF のアプリケーションで入力した文字をメッセージ・プロバイダに送信し、送信された文字列を取り出すアプリケーションを実装します。今回、メッセージ・プロバイダとして GlassFish v4.0 にバンドルされている OpenMQ を利用します。⁴そこで別途メッセージ・プロバイダをインストールする必要はありません。

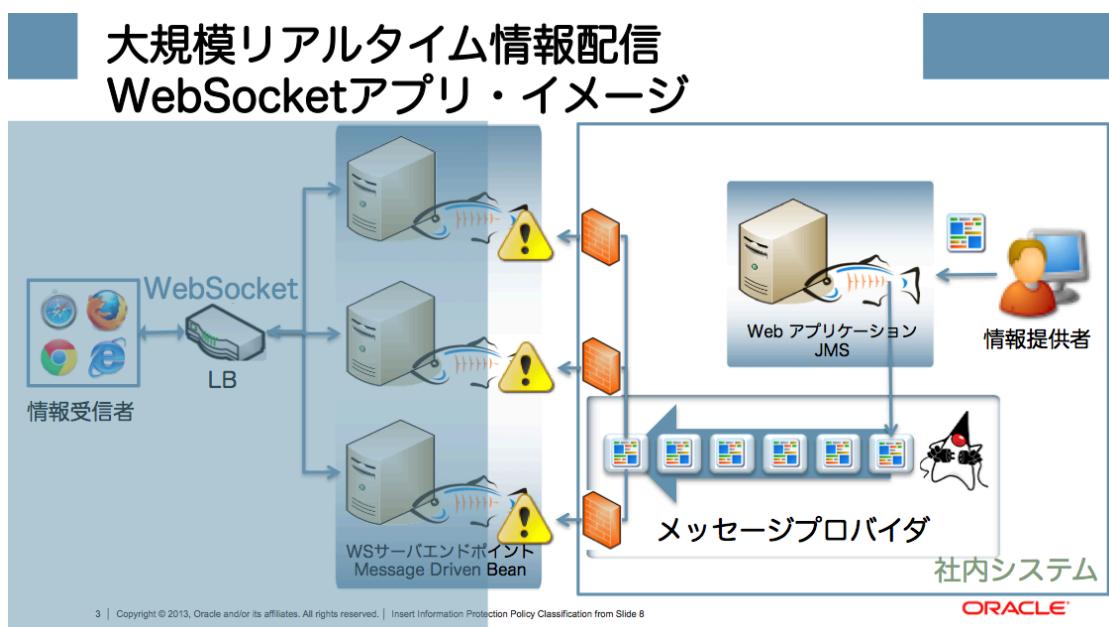


図 46 : JMS アプリケーションの概念図

JMS のアプリケーションを実際に作成していく前に JMS の仕組みについて簡単に紹介します。JMS は Java 用の Message Oriented Middleware(MOM) の API を提供し、メッセージを送受信するための機能を提供します。JMS を利用する事でシステム間の疎結合を実現することができます。

JMS では下記に示す 2 種類の実装モデル(Point-to-Point, Publish/Subscriber)を提供しています。

⁴ GlassFish v4.0 は「Full Java EE Platform」をインストールした場合、Open MQ (<https://mq.java.net/>) が GlassFish にバンドルされています。今回は GlassFish にバンドルされている Open MQ をメッセージ・プロバイダとして利用します。Open MQ 5.0.1 より、Open MQ 単体で WebSocket をサポートしていますが、今回は Java EE アプリケーションとして実装するため GlassFish 経由で OpenMQ を使用します。

Point To Point (1対1)

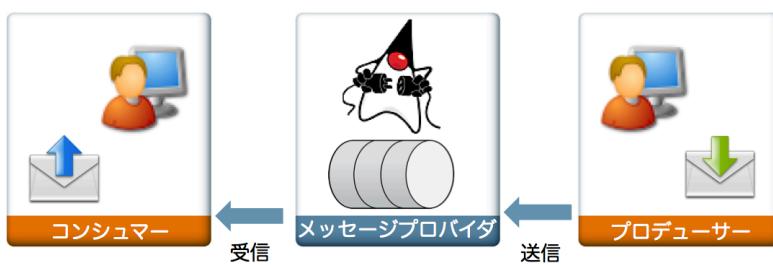


図 47 : Point to Point モデル

Point To Point では、送信側と受信側で 1 対 1 の関係を持ち、メッセージ・プロバイダ上に存在する、Queue を通じてメッセージをコンシュマーにメッセージを届けます。1 対 1 と記載していますが、実際には送信側は複数のプロデューサーがメッセージを送信する事も可能です。一方で受信側は必ず 1 になります。このモデルではメッセージがコンシュマー（受信側）に届くまで、もしくはメッセージ自身の有効期限が切れるまでメッセージは Queue に残り続けます。

Publish / Subscribe (1対多)

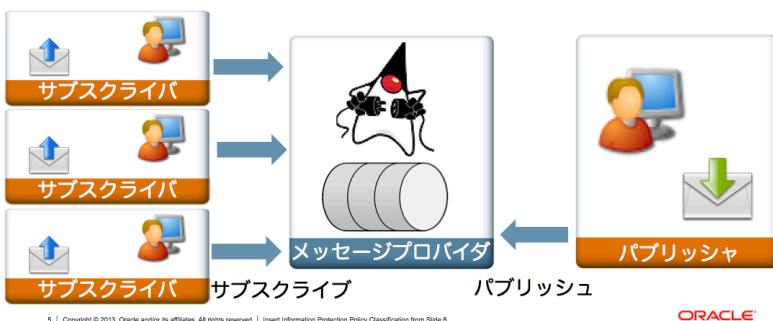


図 48 : Publish/Subscriber モデル

一方で、Publish/Subscriber モデルでは送信側と受信側で 1 対多の関係を持つ事ができます。メッセージの送信者であるパブリッシャはメッセージ・プロバイダに存在する Topic に対してメッセージを送信します。Topic に対してパブリッシュされたメッセージは複数のクライアントがメッセージを受信する事ができるようになります。送信側は受信側の台数やシステム構成等を意図する事なく Topic に配信することができ、また受信側は、実際のパブリッシャ（送信側）を意識する事なく興味のあるメッセージを受信できるます。

今回のアプリケーションは、大規模環境を想定しクラスタ環境における複数のインスタンスで同一メッセージを受信できるようにするために、Publish/Subscriber のモデルを使用して実装します。このモデルを使用する事でシステムに対する負荷が増大した場合もソースコードに一切手を加えることなく、クラスタのインスタンスを追加するだけシステムを柔軟に拡張できるようになります。

次に、JMS アプリケーションの作成方法について説明します。JMS のアプリケーションは、Java EE 環境だけでなく Java SE 環境でも実装する事ができます。しかし Java EE 環境では、Java SE 環境で実装するよりもかんたんに実装する事ができます。

まず、アプリケーション・サーバ側でメッセージ・プロバイダ(OpenMQ)に対する接続ファクトリと宛先の設定を行います。

メッセージ・プロバイダ設定

- アプリケーション・サーバの Topic 設定
 - 接続ファクトリ
 - 宛先名

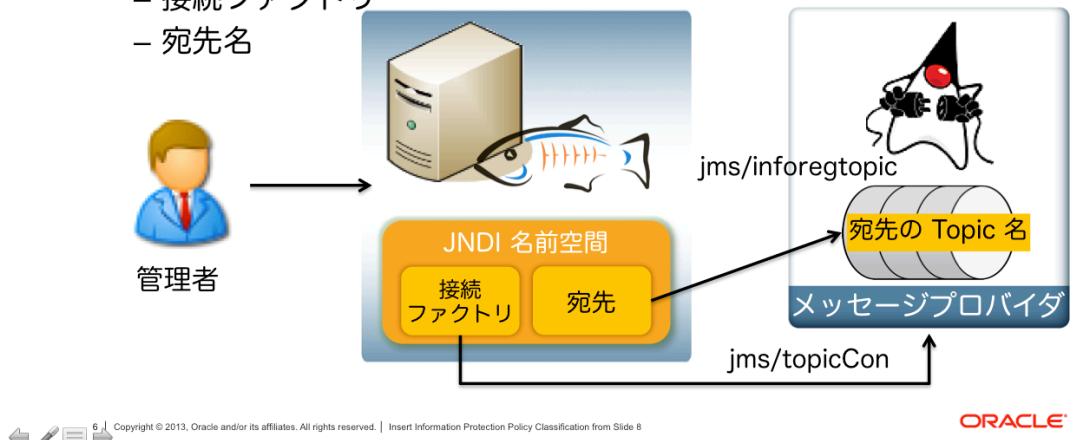


図 49：メッセージ・プロバイダの設定

GlassFish v4.0 で JMS リソースで Topic 用の「接続ファクトリ」を作成するためには下記のコマンドを実行します。⁵

```
> asadmin create-jms-resource --restype
    javax.jms.TopicConnectionFactory jms/topicCon
コネクタ・リソース jms/topicCon が作成されました。
Command create-jms-resource executed successfully.
```

つづいて、Topic の物理的な「宛先リソース」を作成します。下記のコマンドを実行してください。

```
> asadmin create-jms-resource --restype javax.jms.Topic --property
Name=physicaltopic jms/inforegtopic
管理対象オブジェクト jms/inforegtopic が作成されました。
Command create-jms-resource executed successfully.
```

⁵ Windows 環境では GlassFish の管理者パスワードの入力を求められます。Windows 環境の GlassFish におけるデフォルト管理者パスワードの確認方法は次ページで紹介します。

[Windows, Linux 環境]

Windows, Linux 環境で、GlassFish の asadmin コマンド、もしくは GUI の管理コンソールにアクセスする際に、管理者パスワードの入力を促された場合、デフォルトで設定されている管理者パスワードは下記の方法で取得することができます。まず、「サービス」タブより「サーバ」→「GlassFish Server 4.0」を選択してください。選択した後右クリックし「プロパティ」を選択してください。

※ Mac OS/X の環境ではデフォルトで管理者パスワードの入力は求められません。

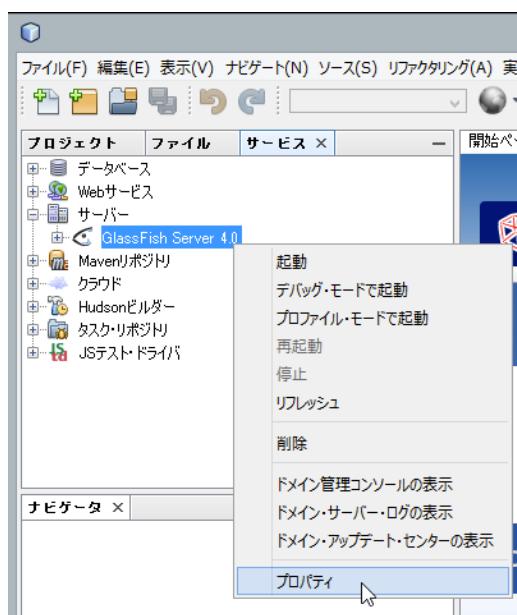


図 50：Windows 環境におけるデフォルト
管理者パスワードの取得

[Windows, Linux 環境]

選択すると下記のウィンドウが表示されます。ここで「パスワード(W) :」と記載されている箇所を確認します。ウィンドウを表示した時点では「●●●●●」と表示されています。ここで「表示」ボタンを押下するとデフォルトのパスワードが表示されます。ここに記入されている内容をメモ帳などに控えておいてください。

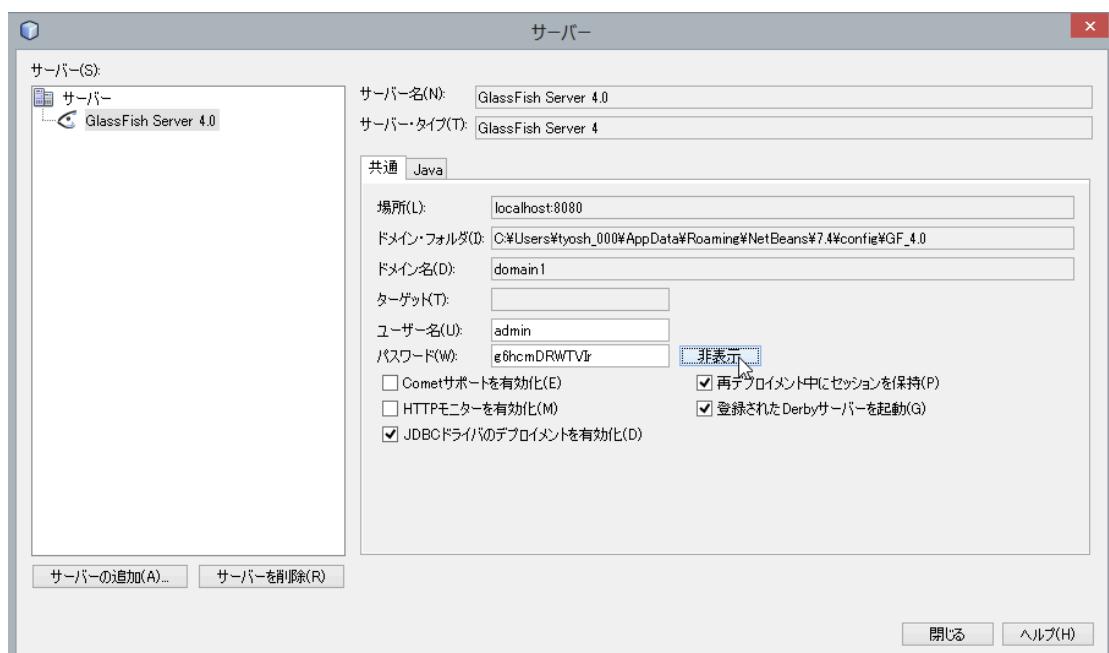


図 51：Windows 環境におけるデフォルト管理者パスワードの取得

GlassFish は GUI の管理コンソールも用意されているため、コマンドを実行する代わりに下記のように GUI で設定を行うことも可能です。ブラウザより下記 URL を入力し管理コンソールに接続してください。

<http://localhost:4848>

[Windows, Linux 環境]

Windows 環境では、ログイン画面が表示されますので、ユーザ名に「admin」、パスワードに先ほど NetBeans より取得した管理者パスワードを入力し「ログイン」ボタンを押下してください。

Java EE 7 Hands-on Lab using GlassFish 4

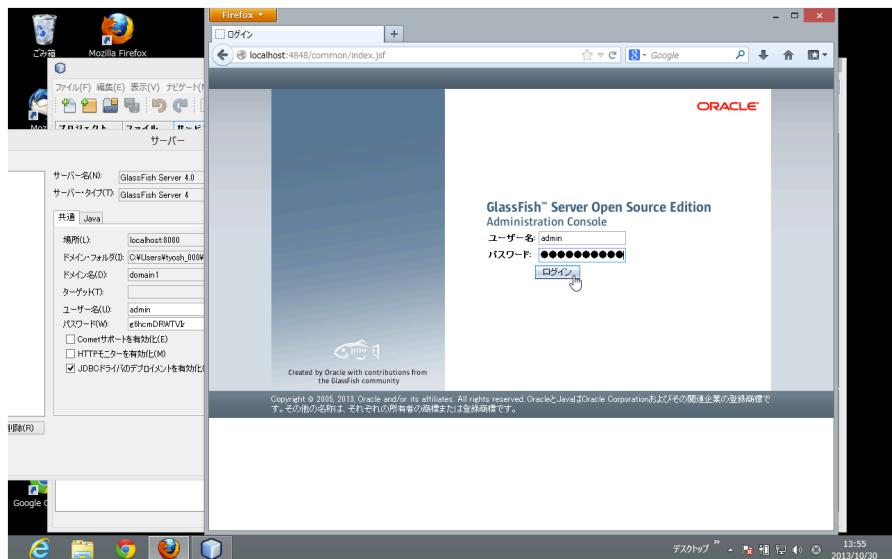


図 52 : Windows 環境における管理コンソールへのログイン

ログインが完了すると下記のウィンドウが表示されます。



図 53 : GlassFish 管理コンソール

Java EE 7 Hands-on Lab using GlassFish 4

デフォルトの GlassFish の管理者パスワードは左ペインより「ドメイン」を選択した後、右ペインより「管理者パスワード」のタブを選択します。選択すると下記の画面が表示されますの「新規パスワード」を設定できます。

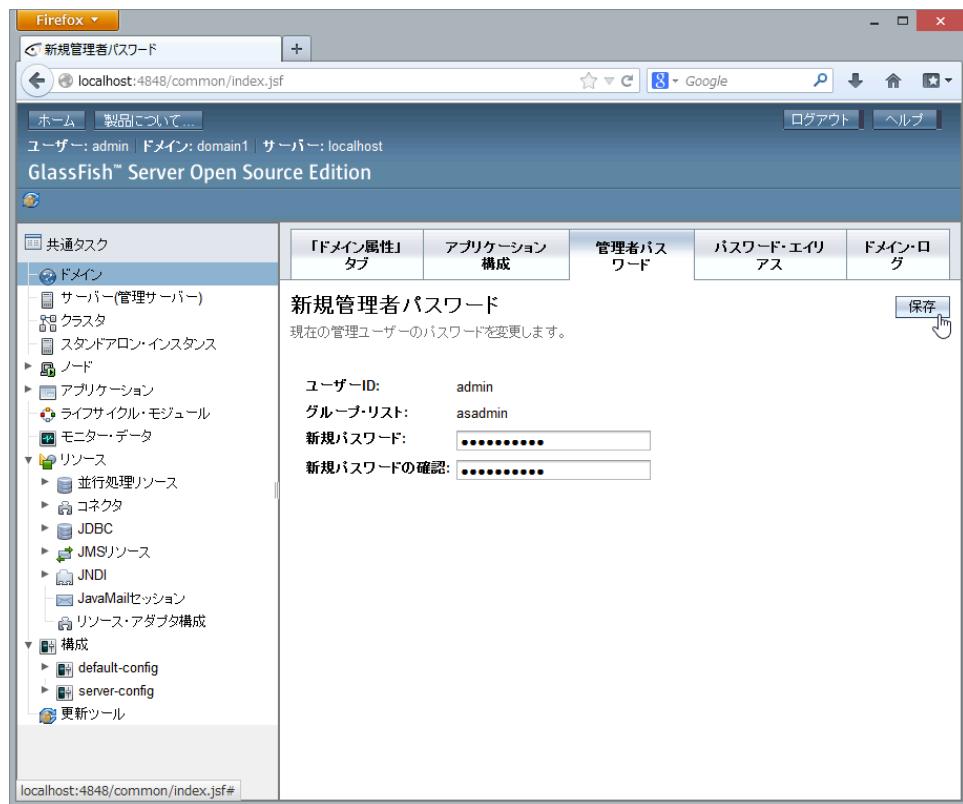


図 54 : GlassFish 管理者パスワードの変更

管理者パスワードを変更した後、NetBeans でワーニング・ウィンドウが表示されますので、GlassFish で変更した管理者パスワードを NetBeans にも反映させてください。

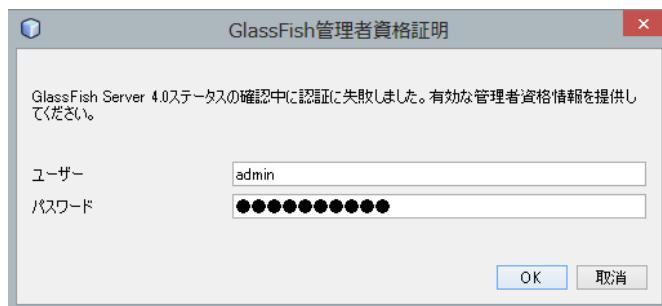


図 55 : NetBeans の GlassFish 管理者パスワードの変更

以降に示す GUI の設定は上記でコマンドを実行した場合は必要ありません。参考のために GUI での設定手順を紹介します。

次に左ペインより、「JMS リソース」のツリーを展開し「接続ファクトリ」を選択します。選択すると下記の画面が表示されますので「新規...」ボタンを押下してください。

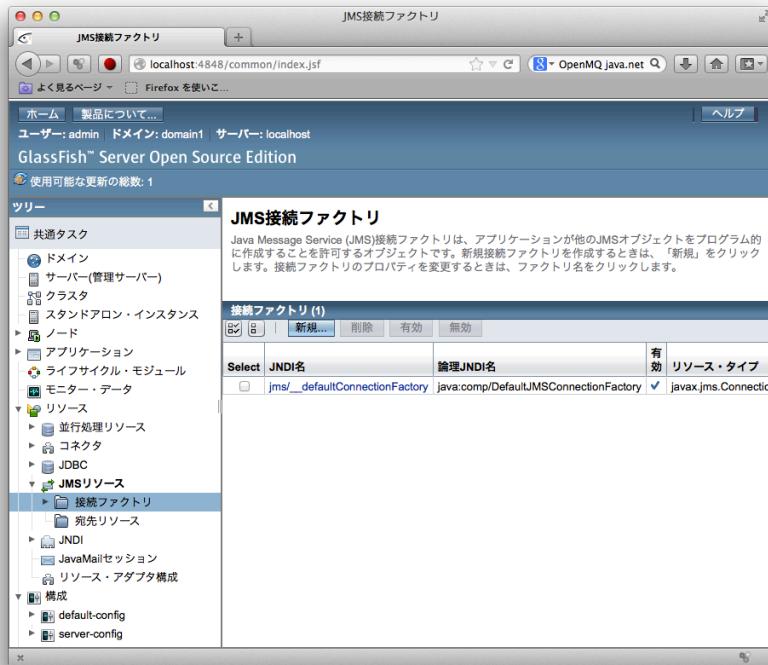


図 56：JMS 接続ファクトリ

Java EE 7 Hands-on Lab using GlassFish 4

ボタンを押下すると右ペインに下記の画面が表示されます。ここで「JNDI名：」に「jms/topicCon」と入力し「OK」ボタンを押下してください。



図 57 : 新規 JMS 接続ファクトリ

次に、JMS の「宛先リソース」を作成します。「JMS リソース」のツリーを展開し「宛先リソース」を選択します。選択すると下記の画面が表示されますので「新規...」ボタンを押下してください。

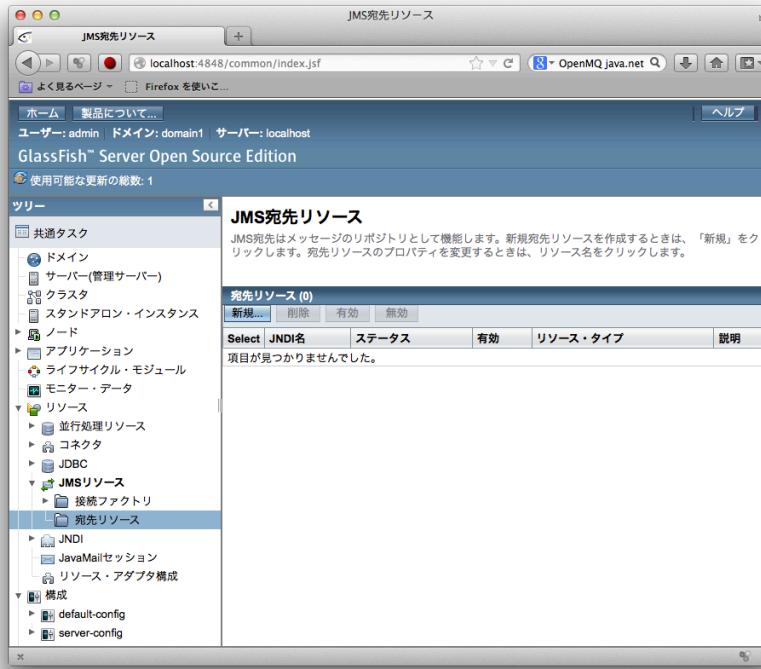


図 58：JMS 宛先リソース

ボタンを押下すると右ペインに下記の画面が表示されます。ここで「JNDI 名：」に「jms/inforegtopic」、「物理宛先名」に「phisicaltopic」と入力し「OK」ボタンを押下してください。



図 59：新規 JMS 宛先リソース

以上でアプリケーション・サーバ側の設定は完了です。

アプリケーション・サーバの設定が完了したので、アプリケーションの実装を行います。

Java EE コンテナ上で JMS のアプリケーションを実装する場合、アプリケーション・サーバのリソースをインジェクトして実装を行います。

JMS アプリケーション on EJB コンテナ

- アプリケーション・サーバの設定を注入

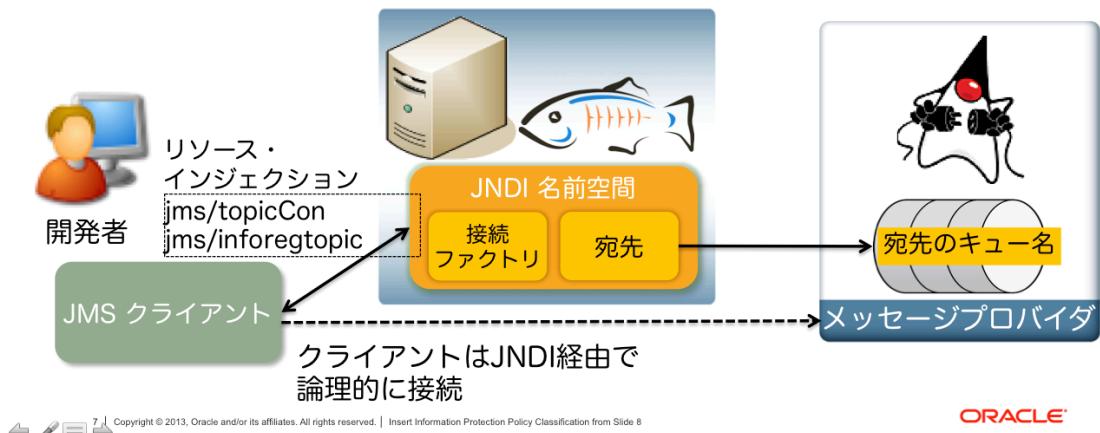


図 60 : JMS アプリケーションの構築概念図

現在、アプリケーション・サーバ上で設定されている内容は下記です。

設定項目	設定値
JMS 接続ファクトリ	Jms/topicCon
JMS 宛先	Jms/inforegtopic

上記設定内容を元にプログラム側からリソースをインジェクトしてメッセージ・プロバイダの Topic に対してメッセージをパブリッシュする実装を行います。JSF のマネージド Bean として実装した IndexPageMgdBean クラスに対して下記のコードを追加してください。

```
package jp.co.oracle.cdis;

import javax.annotation.Resource;
import javax.inject.Named;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.jms.JMSConnectionFactory;
import javax.jms.JMSContext;
import javax.jms.Topic;

@Named(value = "indexManage")
@RequestScoped
public class IndexPageMgdBean {

    @Inject
    @JMSConnectionFactory("jms/topicCon")
```

```
JMSContext context;
@Resource(mappedName = "jms/inforegtopic")
Topic topic;

private String message;

public IndexPageMgdBean() {
}

public String getMessage() {
    return message;
}

public void setMessage(String message) {
    this.message = message;
}

public String pushSendButton() {
    context.createProducer().send(topic, getMessage());
    return "";
}
}
```

メッセージのパブリッシュ（送信用）のコードは以上です。

JMSContext は Java EE 7 から追加されたクラスで、このクラスを利用することで JMS アプリケーションの実装がとてもかんたんになります。JMSContext を利用できるようにするために、アプリケーション・サーバで設定したリソース名を使用して@JMSSConnectionFactory と@Inject アノテーションでインジェクトします。また Topic に対する「JMS 宛先リソース」を@Resource アノテーションでインジェクトします。メッセージを送信するためには、JMSContext#createProducer() メソッドを呼び出して JMSProducer クラスのオブジェクトを生成し、メッセージを送信します。

参考：

Java EE 6 までは、JMS の Topic に対するメッセージのパブリッシュ（送信用）を行うためには、下記の手順が必要な他、JMSEception のハンドリングもしなければならなかったため、多くの冗長的な実装コードが必要でした。

- @Resource アノテーションで Topic 用の接続ファクトリ TopicConnectionFactory をインジェクト
- TopicConnectionFactory よりコネクション TopicConnection を生成
- TopicConnection より TopicSession を生成
- TopicSession より TopicPublisher を生成
- メッセージ送信用の Message オブジェクトを生成
- publish() メソッドを呼び出し

Java EE 7 の JMS 2.0 では特にメッセージを送信するため (Queue, Topic 共に) に必要な実装コードがとても短く、またとてもかんたんに実装できることが上記より分かります。

今まで、JMS アプリケーションの実装に対して敷居が高いと感じられていた方は Java EE 7 から JMS アプリケーションをお試しください。

Java EE 7 Hands-on Lab using GlassFish 4

続いてメッセージ受信用のコードを Message-Driven Bean (MDB) として実装します。プロジェクトを選択したのち、右クリックし「新規」→「その他...」を選択してください。

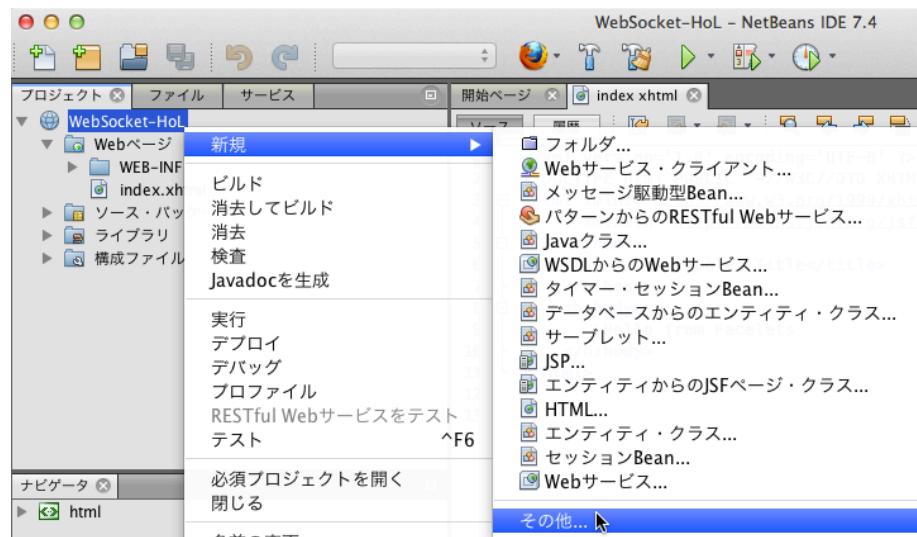


図 61：新規メッセージ駆動型 Bean の作成

選択すると下記のウィンドウが表示されます。ここで「カテゴリ(C):」より「エンタープライズ JavaBeans」を選択し、「ファイル・タイプ(F):」より「メッセージ駆動型 Bean」を選択し「次 >」ボタンを押下してください。

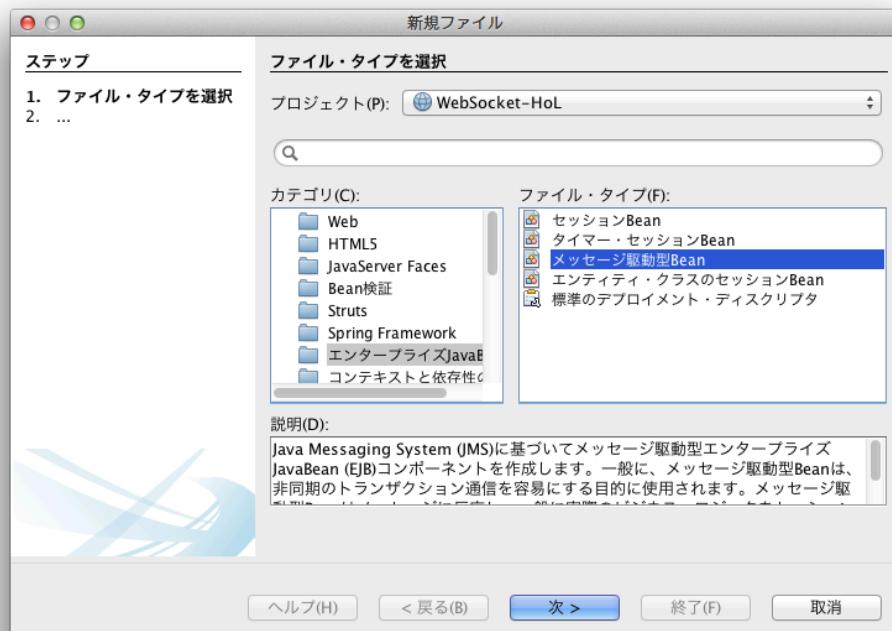


図 62：新規メッセージ駆動型 Bean の作成

ボタンを押下すると下記のウィンドウが表示されます。ここで「EJB 名(N):」に「MessageListenerMDBImpl」を入力し、「パッケージ(K) :」に「jp.co.oracle.ejbs」を選択してください、また「サーバの送信先(S):」の部分で「jms/inforegtopic」を選択し最後に「次 >」ボタンを押下してください。

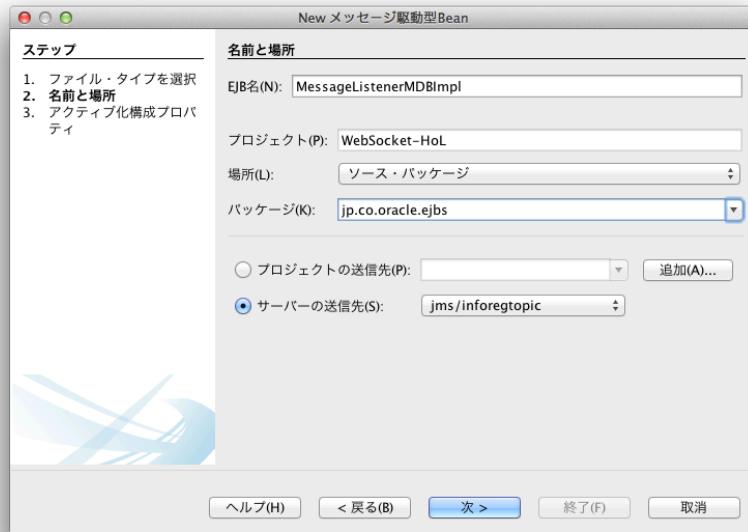


図 63：New メッセージ駆動型 Bean の作成

ボタンを押下すると下記のウィンドウが表示されます。ここではデフォルトの設定のまま「終了(F):」ボタンを押下してください。

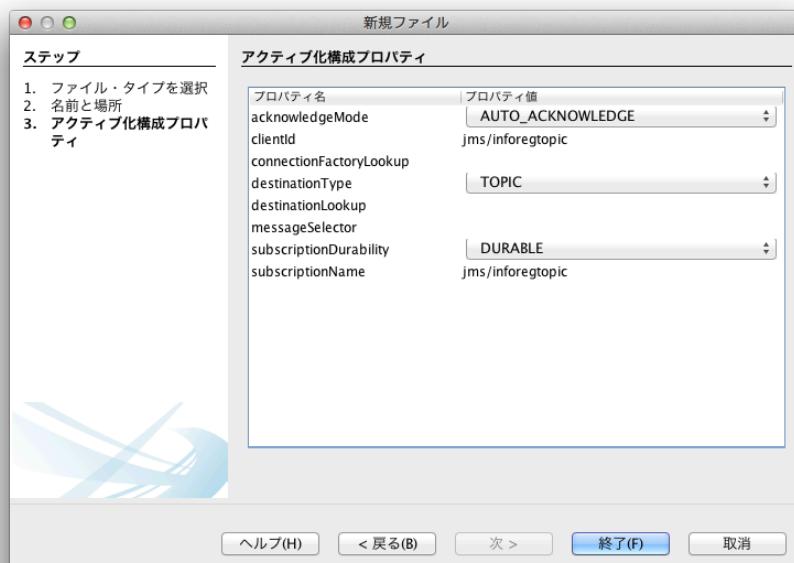


図 64：New メッセージ駆動型 Bean の作成

ボタンを押下すると下記のコードが自動生成されます。

```
/*
 * To change this license header, choose License Headers in Project
 * Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package jp.co.oracle.ejbs;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;

/**
 *
 * @author *****
 */
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "destinationLookup",
        propertyValue = "jms/inforegtopic"),
    @ActivationConfigProperty(propertyName =
        "subscriptionDurability", propertyValue = "durable"),
    @ActivationConfigProperty(propertyName = "clientId",
        propertyValue = "jms/inforegtopic"),
    @ActivationConfigProperty(propertyName = "subscriptionName",
        propertyValue = "jms/inforegtopic")
})
public class MessageListenerMDBImpl implements MessageListener {

    public MessageListenerMDBImpl() {
    }

    @Override
    public void onMessage(Message message) {
    }
}
```

上記のコードを下記のように修正してください。MDB はクラスに対して、
`@MessageDriven` のアノテーションを付加し、
`@ActivationConfigProperty` のアノテーションで設定を定義します。また
MDB を実装するクラスは `MessageListener` インタフェースを実装し、
`onMessage()` メソッドをオーバーライドします。下記の `onMessage()` メソッドの実装では、メッセージ・プロバイダより受け取ったメッセージを標準出力に出力しています。

```
package jp.co.oracle.ejbs;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
```

```

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "destinationLookup",
        propertyValue = "jms/inforegtopic"),
    @ActivationConfigProperty(propertyName =
        "subscriptionDurability", propertyValue = "Durable"),
    @ActivationConfigProperty(propertyName = "clientId",
        propertyValue = "${com.sun.aas.instanceName}"),
    @ActivationConfigProperty(propertyName = "subscriptionName",
        propertyValue = "jms/inforegtopic")
})
public class MessageListenerMDBImpl implements MessageListener {

    private static final Logger logger =
        Logger.getLogger(
            MessageListenerMDBImpl.class.getPackage().getName());

    public MessageListenerMDBImpl() {
    }

    @Override
    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage) message;
        try {
            String text = textMessage.getText();
            System.out.println(text);
        } catch (JMSException ex) {
            logger.log(Level.SEVERE, "recieve message failed :", ex);
        }
    }
}

```

※ソースコードの補足⁶

上記でメッセージ受信の実装は完了です。コードを修正した後、NetBean のプロジェクトを実行してください。

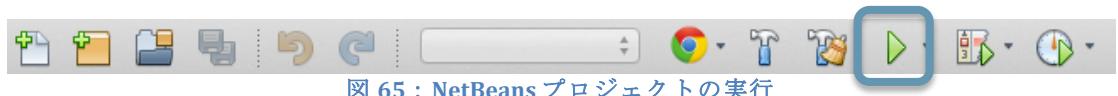


図 65 : NetBeans プロジェクトの実行

⁶ MDB の activationConfig の設定で **clientId**, **propertyValue = "\${com.sun.aas.instanceName}"** の項目があります。これはクラスタ環境で、複数台のマシンで運用する場合、**clientId** に対して固有の名前を割り当てる必要があるため GlassFish 固有の設定を使用しています。実際にはインスタンス名がここに代入されます。また、**subscriptionDurability** の値がデフォルトで **durable** となっており NetBeans の自動生成コードのバグと思われます。実際には **Durable** ですので変更してください。

実行すると、下記の画面が表示されます。テキスト・フィールドに文字列を入力し「Enter」キーを押下するか、もしくは「Send Message」ボタンを押下してください。

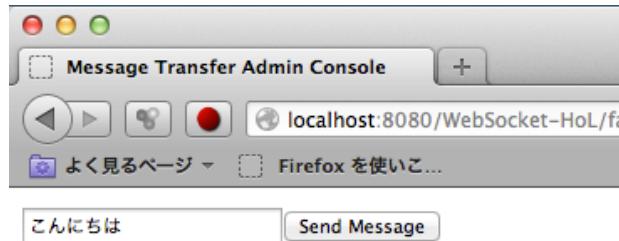


図 66：Web ページに文字列を入力

NetBeans の「出力」ウィンドウの「GlassFish Server 4.0」のタブを選択すると GlassFish のログが表示されています。ボタンを押下すると、入力した文字列と同じ文字列がログに表示されます。同じ文字が表示されていればメッセージ・プロバイダを通じて、メッセージの送受信が正常に行われています。

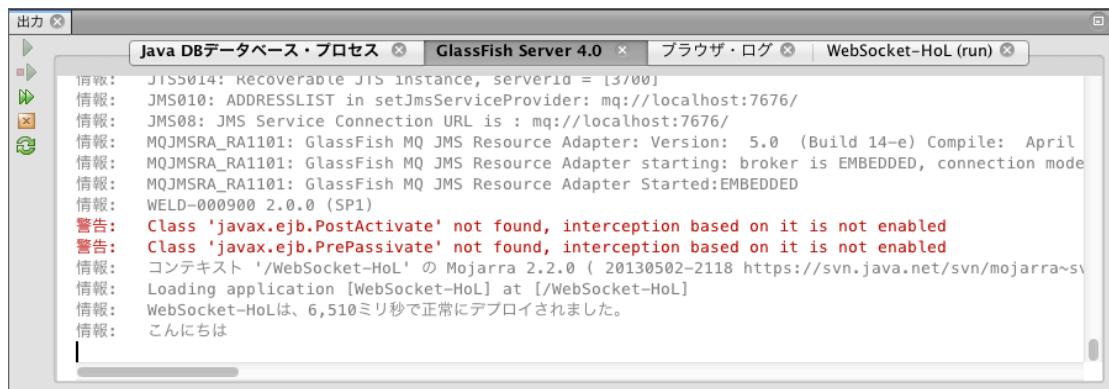


図 67：GlassFish のログ出力

以上で、JSF と JMS のアプリケーションの連携は完了です。

5.0 WebSocket アプリケーションの作成

最後に、WebSocket アプリケーションを作成します。

WebSocket は HTTP をアップグレードした TCP ベースのプロトコルで、双方向・全二重の通信ができます。また WebSocket プロトコルの仕様は RFC 6455 で定義され、API は W3C によって定義されています。



図 68 : WebSocket について

WebSocket はライフサイクル（コネクションの接続・切断、メッセージ受信、エラー発生）を持ち、それぞれのサイクルに対応した実装を行います。



図 69 : WebSocket のライフサイクル

Java EE 7 では JSR 356 として「Java™ API for WebSocket」が新たに追加され、下記に示すアノテーションを使ってかんたんに WebSocket の工

ンドポイントの実装ができます。アノテーションは、クライアント側の実装かサーバ側の実装化を識別するために用意された、クラスに付加する(@ServerEndpoint, @ClientEndpoint⁷) アノテーションと、WebSocket の各ライフサイクルに対応するメソッドに付加する(@OnOpen, @OnClose, @OnMessage, @OnError)アノテーションが用意されています。
@PathParam のアノテーションは特別で、RESTful Web サービスのように、WebSocket のリクエスト URI パスの一部をパラメータとして扱う為に指定するアノテーションです。

WebSocket アノテーション

アノテーション	レベル	内容
@ServerEndpoint	クラス	サーバ側のエンドポイントを示す
@ClientEndpoint	クラス	クライアント側のエンドポイントを示す
@OnOpen	メソッド	接続確立時に呼び出すメソッドを指定
@OnClose	メソッド	接続切断時に呼び出すメソッドを指定
@OnMessage	メソッド	メッセージを受信した時に呼び出すメソッドを指定
@PathParam	メソッド パラメータ	エンドポイントのURI 引数に指定されたパラメータを取得する際に仕様
@OnError	メソッド	エラー発生時に呼び出すメソッドを指定

50 | Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

 Java™  ORACLE®

図 70 : WebSocket のアノテーション

今回のアプリケーションはサーバ・エンドポイント（サーバ側）の実装行います。JSR 356 の API を使用して簡単に WebSocket のサーバ・エンドポイントが実装できることを確認してください。

⁷ Java EE はサーバ・サイドのテクノロジーを取り扱いますが、JSR 356 準拠の実行環境はクライアント側の実装も提供します。クライアントの API を利用することで、JavaFX 等の Java クライアントから WebSocket サーバ・エンドポイントへ接続するアプリケーションも実装できるようになります。

この WebSocket アプリケーションは、まず「情報受信者（クライアント・エンドポイント）」が WebSocket のサーバ・エンドポイントに接続し、サーバ側で「情報受信者」の接続・切断情報を管理します。次に、前章で実装した MDB がメッセージ・プロバイダの Topic を監視し、メッセージを受信した際に接続されている全情報受信者に対して情報を発信します。

大規模リアルタイム情報配信 WebSocketアプリ・イメージ

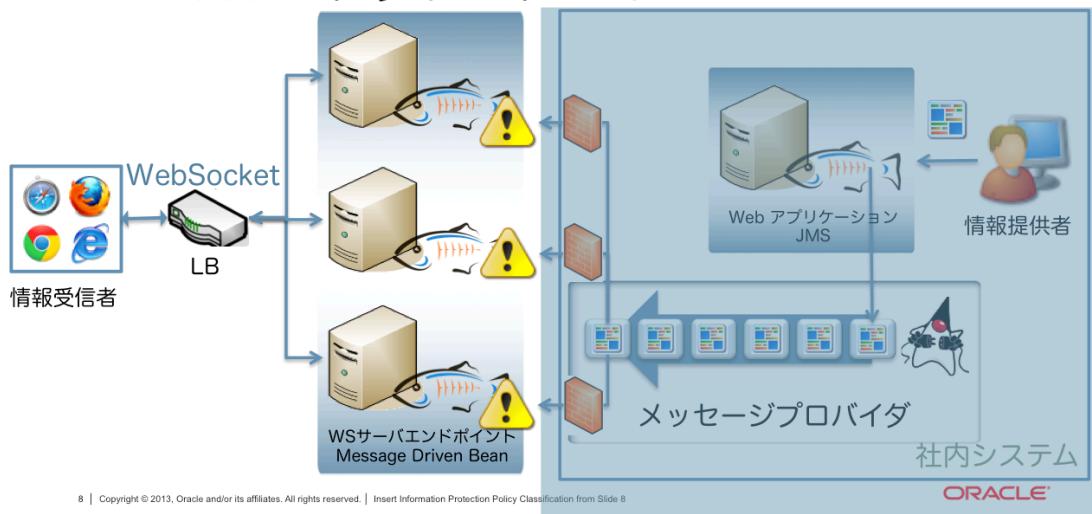


図 71 : WebSocket アプリケーションの概念図

まず、WebSocket のサーバ・エンドポイントを作成します。プロジェクトをマウスで選択し右クリックしてください。次に「新規」→「その他...」を選択してください。

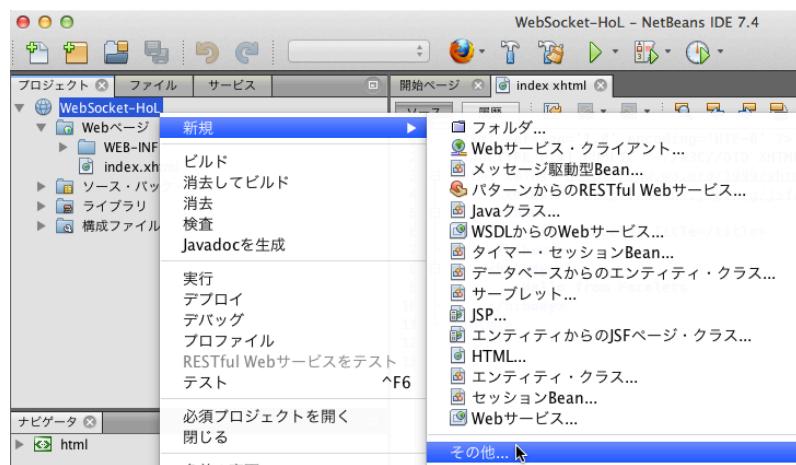


図 72 : WebSocket サーバ・エンドポイントの作成

選択すると下記のウィンドウが表示されます。「カテゴリ(C):」より「Web」を選択し、「ファイル・タイプ(F):」より「WebSocket エンドポイント」を選択して「次 >」ボタンを押下してください。

Java EE 7 Hands-on Lab using GlassFish 4

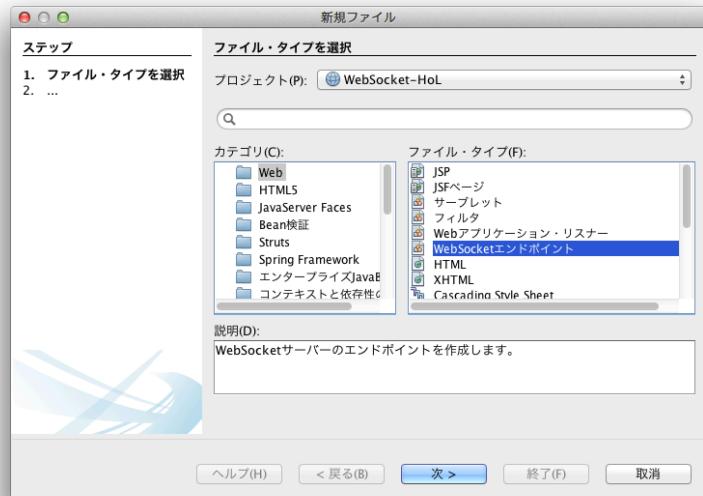


図 73：WebSocket エンドポイントの作成

ボタンを押下すると下記のウィンドウが表示されます。ここで「クラス名(N):」に「InfoTransServerEndpoint」を入力し、「パッケージ(K):」に「jp.co.oracle.websockets」、「WebSocket URI(U):」に「/infotrans」を入力した後、最後に「終了(F)」ボタンを押下してください。

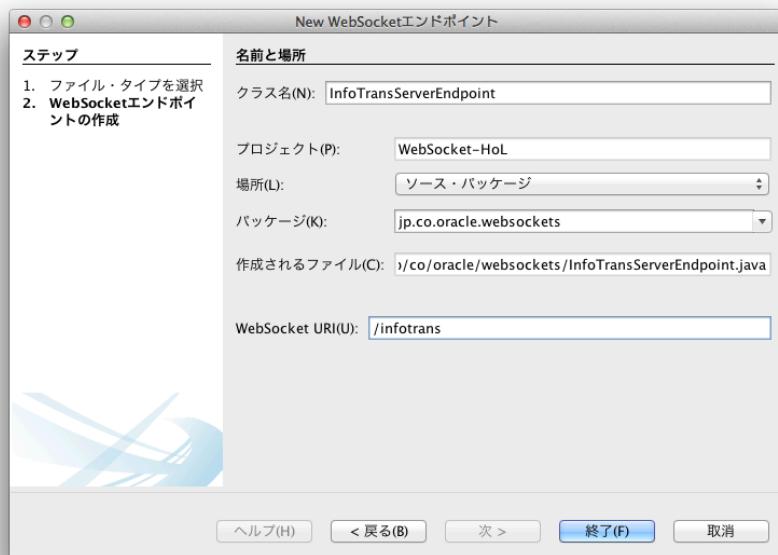


図 74：New WebSocket エンドポイント

ボタンを押下すると下記のコードが自動的に生成されます。

```
/*
 * To change this license header, choose License Headers in Project
 * Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package jp.co.oracle.websockets;

import javax.websocket.OnMessage;
import javax.websocket.server.ServerEndpoint;

/**
 *
 * @author *****
 */
@ServerEndpoint("/infotrans")
public class InfoTransServerEndpoint {

    @OnMessage
    public String onMessage(String message) {
        return null;
    }

}
```

今回のアプリケーションでは WebSocket サーバ・エンドポイントでは接続・切断の管理だけをおこない、クライアントからメッセージの受信は行わないため@OnMessage のアノテーションが付加されたコードは不要で削除してください。その代わりに@OnOpen, @OnClose のメソッドを実装してください。

```
package jp.co.oracle.websockets;

import javax.websocket.OnClose;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/infotrans")
public class InfoTransServerEndpoint {

    @OnOpen
    public void initOpen(Session session) {
        System.out.println("接続");
    }

    @OnClose
    public void closeWebSocket(Session session) {
        System.out.println("切断");
    }
}
```

次にこのサーバ・エンドポイントに接続する HTML ファイルを作成します。まず、プロジェクト作成時に自動生成された「index.xhtml」ファイルを削除します。対象のファイルを選択し右クリックしてください。メニューより「削除」を選択してください。

※ ここで削除するファイルは、admin/index.xhtml ではありません。



図 75：既存ファイルの削除

削除を選択すると下記のダイアログ・ウィンドウが表示されます。ここで「はい」ボタンを押下してください。



図 76：オブジェクト削除の確認

次に、HTML ファイルを作成します。プロジェクトを選択し右クリックしてください。次に「新規」→「その他...」を選択してください。

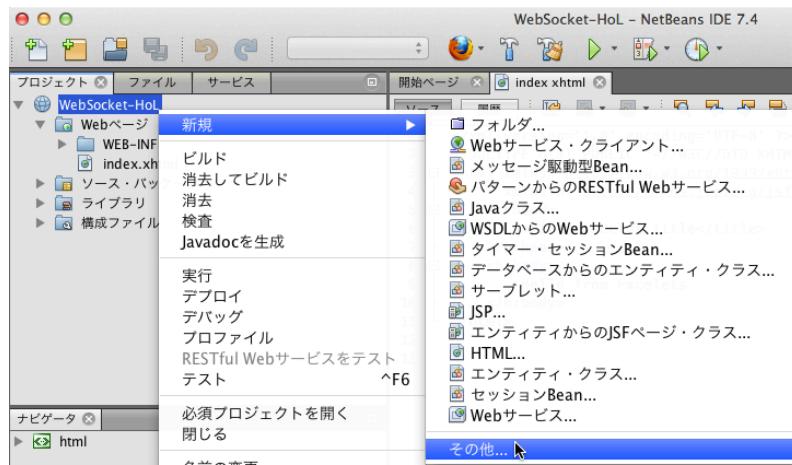


図 77：新規 HTML ファイルの作成

選択すると下記のウィンドウが表示されます。「カテゴリ(C):」より「HTML5」を選択し、「ファイル・タイプ(F):」より「HTML ファイル」を選択し「次 >」ボタンを押下してください。

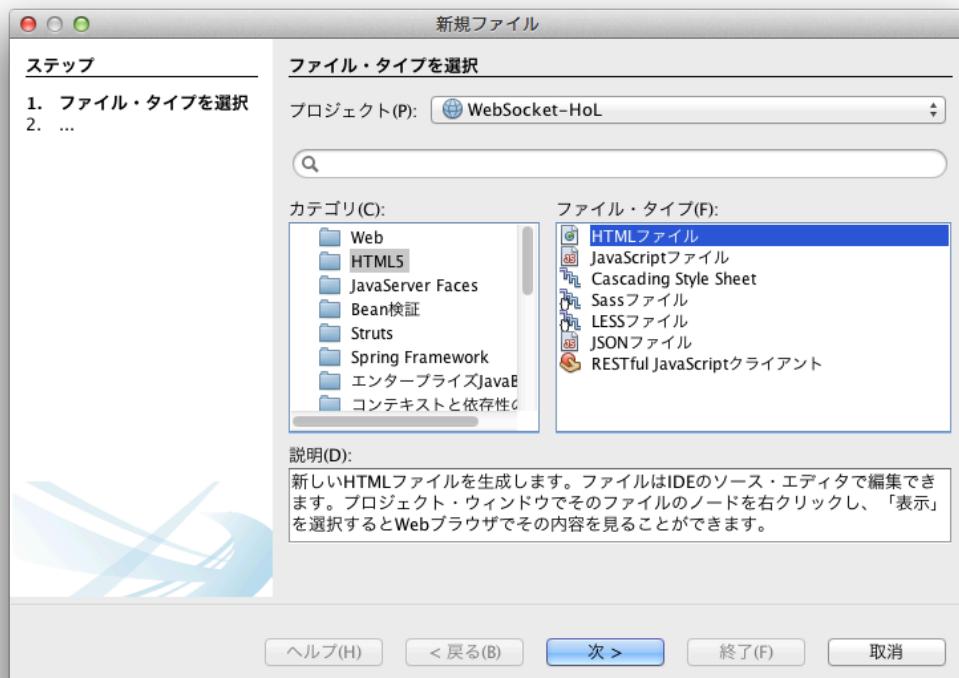


図 78：新規 HTML ファイルの作成

ボタンを押下すると下記のウィンドウが表示されます。「ファイル名(N):」に「client-endpoint」を入力し、「フォルダ(L):」に「web」を入力した後最後に「終了(F)」ボタンを押下してください。



図 79：新規 HTML ファイルの作成

ボタンを押下すると下記のコードが自動生成されます。

```
<!DOCTYPE html>
<!--
To change this license header, choose License Headers in Project
Properties.
To change this template file, choose Tools | Templates
and open the template in the editor.
-->
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width">
  </head>
  <body>
    <div>TODO write content</div>
  </body>
</html>
```

上記コードを WebSocket サーバ・エンドポイントに接続するコードに修正します。HTML に下記のコードを実装してください。下記のコードは WebSocket のサーバ・エンドポイントを示す下記の URL に接続し、「ws://localhost:8080/WebSocket-HoL/infotrans」 WebSocket の各ライフサイクルの実装を JavaScript で実装しています。また WebSocket のサーバ・エンドポイントに接続されている時は、「Connect」ボタンを非表示にし、「DisConnect」のボタンを表示します。逆に切断されている時は、「DisConnect」ボタンを非表示にし、「Connect」ボタンを表示します。

```
<!DOCTYPE html>

<html>
  <head>
    <title>WebSocket RealTime Infomation Transfer</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <style type="text/css">
      table,td,th {
        width: 700px;
        font-size: medium;
        border-collapse: collapse;
        border: 1px black solid;
      }
    </style>
  <script language="javascript" type="text/javascript">
    var websocket = null;
    var numberOfMessage;

    function init() {
      numberOfMessage = 0;
      document.getElementById("close").style.display = "none";
    }

    function closeServerEndpoint() {
      websocket.close(4001, "Close connection from client");
      document.getElementById("connect").style.display = "block";
      document.getElementById("close").style.display = "none";
      document.getElementById("server-port").disabled = false;
    }

    function connectServerEndpoint() {
      var host = document.getElementById("server-port").value;
      var wsUri = "ws://" + host + "/WebSocket-HoL/infotrans";
      if ("WebSocket" in window) {
        websocket = new WebSocket(wsUri);
      } else if ("MozWebSocket" in window) {
        websocket = new MozWebSocket(wsUri);
      } else {
        websocket = new Websocket(wsUri);
      }
      websocket.onopen = function(evt) {
        onOpen(evt);
      };
      websocket.onmessage = function(evt) {
        onMessage(evt);
      };
      websocket.onerror = function(evt) {
        onError(evt);
      };
    }

    function onOpen(evt) {
      var message = "Connected to server";
      var data = document.createElement("p");
      data.innerHTML = message;
      document.getElementById("log").appendChild(data);
      numberOfMessage++;
      var message = "Number of messages: " + numberOfMessage;
      var data = document.createElement("p");
      data.innerHTML = message;
      document.getElementById("log").appendChild(data);
      var message = "Message: " + evt.data;
      var data = document.createElement("p");
      data.innerHTML = message;
      document.getElementById("log").appendChild(data);
    }

    function onMessage(evt) {
      var message = "Received message: " + evt.data;
      var data = document.createElement("p");
      data.innerHTML = message;
      document.getElementById("log").appendChild(data);
    }

    function onError(evt) {
      var message = "Error: " + evt.message;
      var data = document.createElement("p");
      data.innerHTML = message;
      document.getElementById("log").appendChild(data);
    }

    function onDisconnect() {
      var message = "Disconnected from server";
      var data = document.createElement("p");
      data.innerHTML = message;
      document.getElementById("log").appendChild(data);
    }

    function onConnect() {
      var message = "Connected to server";
      var data = document.createElement("p");
      data.innerHTML = message;
      document.getElementById("log").appendChild(data);
    }
  </script>
</head>
<body>
  <table border="1">
    <tr>
      <td>Server Port</td>
      <td><input type="text" id="server-port" value="localhost:8080" /></td>
    </tr>
    <tr>
      <td>Connect</td>
      <td><input type="button" id="connect" value="Connect" /></td>
    </tr>
    <tr>
      <td>Close</td>
      <td><input type="button" id="close" value="Close" /></td>
    </tr>
  </table>
  <p>Log:</p>
  <div id="log"></div>
</body>

```

```

        onError(evt);
    };
    websocket.onclose = function(evt) {
        closeServerEndpoint();
    };
    document.getElementById("connect").style.display = "none";
    document.getElementById("close").style.display = "block";
    document.getElementById("server-port").disabled = true;
}

function onOpen(evt) {
    ;
}

function onMessage(evt) {
    writeToScreen(evt.data);
    numberOfMessage++;
}

function onError(evt) {
    writeToScreen("ERROR: " + evt.data);
}

function writeToScreen(messages) {
    var table = document.getElementById("TBL");
    var row = table.insertRow(0);
    var cell1 = row.insertCell(0);
    cell1.style.color = "WHITE";

    var textNode = document.createTextNode(messages);
    var z = numberOfMessage % 2;
    if (z == 1) {
        cell1.style.backgroundColor = "#669900";
    } else {
        cell1.style.backgroundColor = "#ED9B09";
    }
    cell1.appendChild(textNode);
}

    window.addEventListener("load", init, false);
</script>
</head>
<body>
    <h2>WebSocket RealTime Infomation Transfer Sample Application!</h2>
    サーバ接続ポート番号:<input id="server-port" type="text" value="" />
        <input id="connect" type="button" value="Connect" onClick="connectServerEndpoint();">
        <input id="close" type="button" value="DisConnect" onClick="closeServerEndpoint();">
        <br/>
        <TABLE BORDER="1" ID="TBL">
        </TABLE>
    </body>
</html>

```

上記のコードを修正した後、NetBean のプロジェクトを実行してください。

Java EE 7 Hands-on Lab using GlassFish 4



NetBeans 実行したのち、ブラウザより下記の URL にアクセスしてください。アクセスすると下記の画面が表示されます。

<http://localhost:8080/WebSocket-HoL/client-endpoint.html>



WebSocket RealTime Information Transfer

サーバ接続ポート番号 : Connect

図 81 : WebSocket クライアント・エンドポイント

次に、「サーバ接続ポート番号」に「localhost:8080」と入力し、

「Connect」、「DisConnect」のボタンを数度押下してください。ボタンを押下した際、GlassFish Server 4.0 のログを確認するとボタンの押下の度に「接続」、「切断」メッセージが繰り返し出力されている事が確認できます。

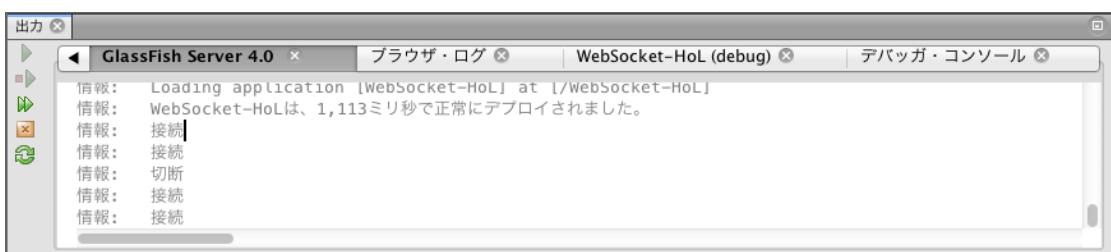


図 82 : GlassFish のログ確認

Java EE 7 Hands-on Lab using GlassFish 4

次に、WebSocket のクライアント・エンドポイントの情報をアプリケーション内で一元管理する Singleton EJB を作成します。プロジェクトを選択し右クリックしてください。次に「新規」→「その他...」を選択します。

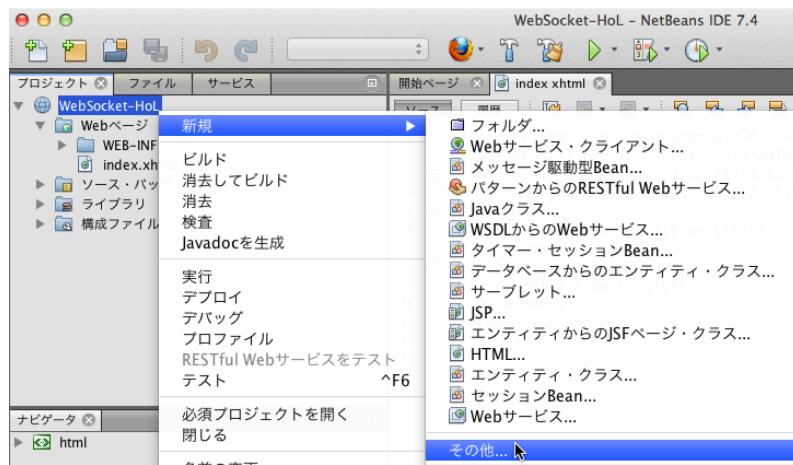


図 83：新規 Singleton EJB の作成

選択すると下記のウィンドウが表示されます。ここで「カテゴリ(C):」より「エンタープライズ JavaBeans」を選択し、「ファイル・タイプ(F):」より「セッション Bean」を選択し「次 >」ボタンを押下してください。

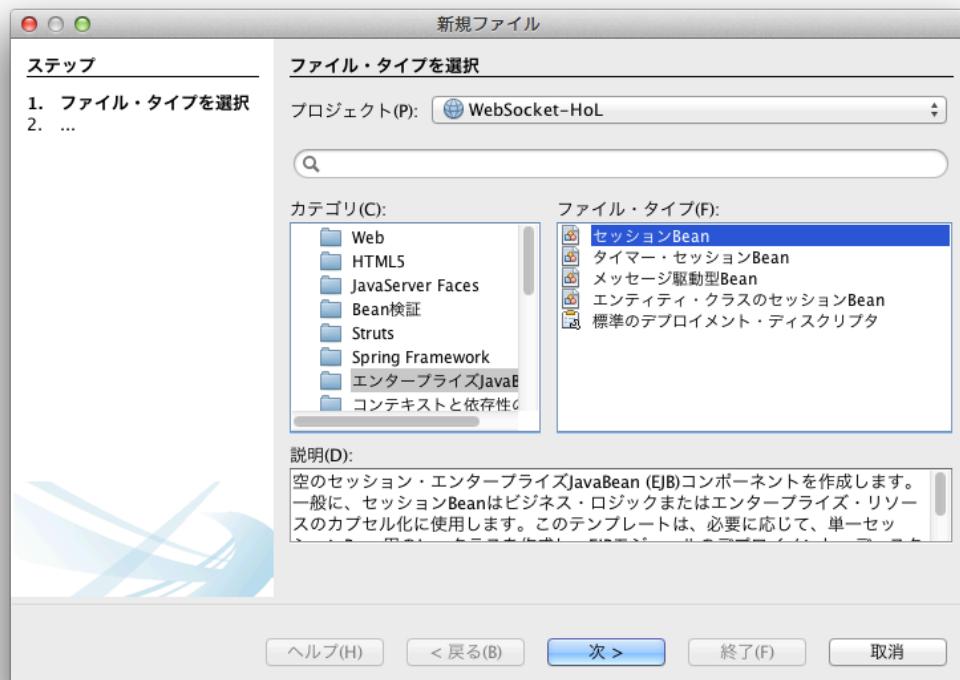


図 84：新規 Singleton EJB の作成

Java EE 7 Hands-on Lab using GlassFish 4

ボタンを押下すると下記のウィンドウが表示されます。ここで「EJB名(N):」に「ClientManageSingleEJB」、「パッケージ(K):」に「jp.co.oracle.ejbs」が記入されている事を確認し、「セッションのタイプ:」のラジオボタンに「シングルトン」を選択し、最後に「終了(F)」ボタンを押下してください。

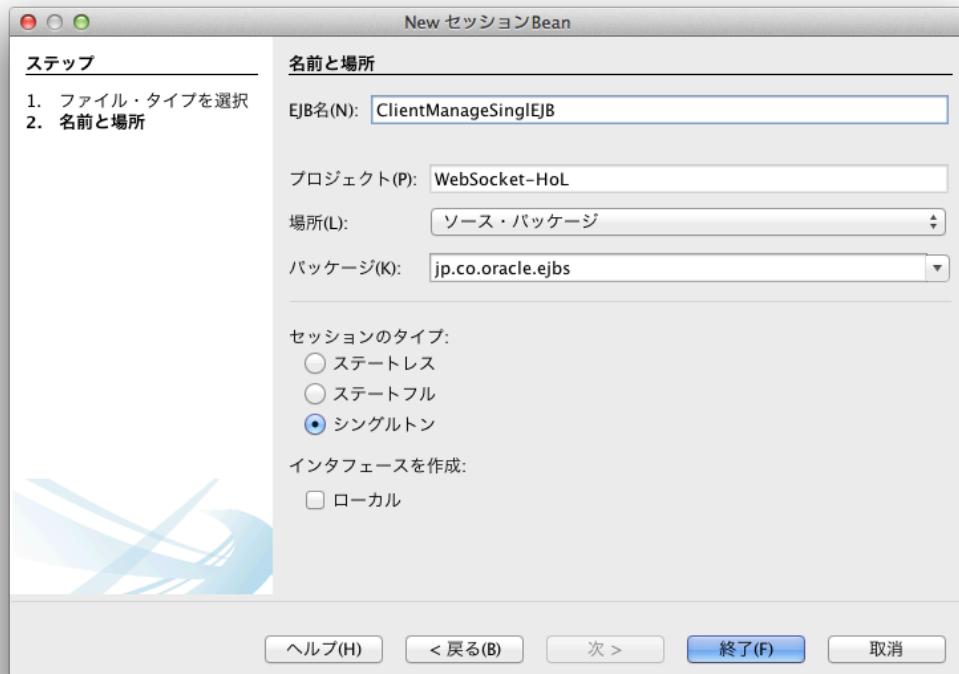


図 85 : New セッション Bean

ボタンを押下すると自動的に下記のコードが生成されます。

```
/*
 * To change this license header, choose License Headers in Project
 * Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

package jp.co.oracle.ejbs;

import javax.ejb.Singleton;
import javax.ejb.LocalBean;

/**
 *
 * @author *****
 */
@Singleton
@LocalBean
public class ClientManageSingleEJB {

    // Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Business Method")
}
```

今回、このシングルトンの EJB ではアプリケーションの起動時に EJB を初期化するため、クラスに対して @Startup を付加しています。この EJB では WebSocket のクライアント・エンドポイントから接続された際にクライアントの Session 情報をコレクションに追加し(addClient)、切断された際にコレクションから削除(removeClient)します。全クライアント情報は Set<Session> peers に含まれ、接続済みの全 WebSocket クライアント・エンドポイントに対してメッセージを同期で送信するために sendMessage() メソッドを実装しています。

```
package jp.co.oracle.ejbs;

import java.io.IOException;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.Singleton;
import javax.ejb.LocalBean;
import javax.ejb.Startup;
import javax.websocket.Session;

@Singleton
@LocalBean
@Startup
public class ClientManageSingleEJB {

    private static final Logger logger =
        Logger.getLogger(
            ClientManageSingleEJB.class.getPackage().getName());

    public ClientManageSingleEJB(){}

    private final Set<Session> peers =
        Collections.synchronizedSet(new HashSet<Session>());

    public void addClient(Session session) {
        peers.add(session);
    }

    public void removeClient(Session session) {
        peers.remove(session);
    }

    public void sendMessage(String message){
        for(Session session : peers){
            try {
                session.getBasicRemote().sendText(message);
            } catch (IOException ex) {
                logger.log(Level.SEVERE,
                           "Failed to send the message to Client :", ex);
            }
        }
    }
}
```

WebSocket のクライアント・エンドポイントの管理を行う Singleton EJB を作成したので、WebSocket のサーバ・エンドポイント側のコードも修正

します。WebSocket のサーバ・エンドポイント側の実装を下記のように修正してください。

下記では@EJB のアノテーションで ClientManageSingleEJB をインジェクトしています。クライアントと接続した場合、ClientManageSingleEJB #addClient() メソッドを呼び出し、切断時に ClientManageSingleEJB #removeClient() メソッドを呼び出し、クライアント・エンドポイントを EJB で一元的に管理しています。

```
package jp.co.oracle.websockets;

import javax.ejb.EJB;
import javax.websocket.OnClose;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import jp.co.oracle.ejbs.ClientManageSingleEJB;

@ServerEndpoint("/infotrans")
public class InfoTransServerEndpoint {
    @EJB
    ClientManageSingleEJB clManager;

    @OnOpen
    public void initOpen(Session session) {
        clManager.addClient(session);
    }

    @OnClose
    public void closeWebSocket(Session session) {
        clManager.removeClient(session);
    }
}
```

また、MDB でメッセージを受信した際に、接続済みの全 WebSocket のクライアント・エンドポイントに対してメッセージを配信するために、MDB の実装を下記のように修正してください。

下記では、@EJB のアノテーションで ClientManageSingleEJB をインジェクトしています。メッセージ・プロバイダの Topic よりメッセージを受信した際に、ClientManageSingleEJB#sendMessage() メソッドを呼び出しています。

```
package jp.co.oracle.ejbs;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.EJB;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(mappedName = "jms/inforegtopic", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
    ...}
}
```

Java EE 7 Hands-on Lab using GlassFish 4

```
@ActivationConfigProperty(propertyName = "subscriptionDurability",
    propertyValue = "Durable"),
@ActivationConfigProperty(propertyName = "clientID",
    propertyValue ="${com.sun.aas.instanceName}"),
@ActivationConfigProperty(propertyName = "subscriptionName",
    propertyValue="TESTSubSCRIPTION")
})
public class MessageListenerMDBImpl implements MessageListener {

    private static final Logger logger = Logger.getLogger(
        MessageListenerMDBImpl.class.getPackage().getName());

    @EJB
    ClientManageSingleEJB clManager;

    @Override
    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage) message;
        try {
            String text = textMessage.getText();
            clManager.sendMessage(text);
        } catch (JMSException ex) {
            logger.log(Level.SEVERE, "onMessage() failed", ex);
        }
    }
}
```

上記のコードを修正した後、NetBean のプロジェクトを実行してください。



図 86 : NetBeans プロジェクトの実行

Java EE 7 Hands-on Lab using GlassFish 4

プロジェクトを実行すると下記の画面が表示されます。ここではそのまま何もせずにこの画面を保持したまま、別のブラウザもしくはブラウザの新規タブを開いてください。



図 87：プロジェクトの実行画面

Java EE 7 Hands-on Lab using GlassFish 4

別のブラウザもしくは別のタブを開き下記の URL にアクセスしてください。

<http://localhost:8080/WebSocket-HoL/client-endpoint.html>



WebSocket RealTime Information Transfer

サーバ接続ポート番号 :

図 88：クライアント・エンドポイントへ接続

ブラウザで URL にアクセスした後、「Connect」ボタンを押下してください。ボタンを押下した後、JSF の画面より“こんにちは”と入力し「Send Message」ボタンを押下するか「Enter Key」を押下してください。メッセージを送信すると WebSocket のクライアント・エンドポイント側で入力した文字が表示されます。

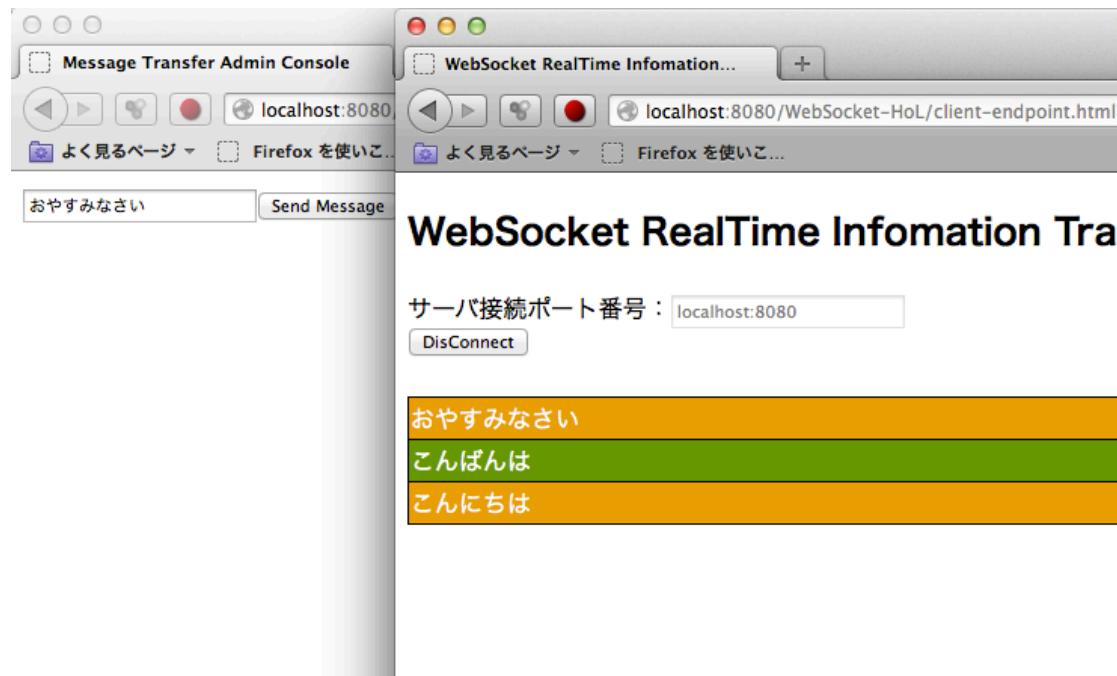


図 89：アプリケーションの実行

以上で全アプリケーションの実装と動作確認は完了です。

6.0 GlassFish クラスタ環境の構築と動作確認

最後に、WebSocket アプリケーションを大規模環境で運用するためにクラスタ環境を構築しクラスタ環境でアプリケーションを動作させます。

今回はハンズ・オンとしてかんたんに実装・検証できることを目的としてプロジェクトを作成したため、情報提供者側の JSF アプリケーションと、情報受信者側の WebSocket アプリケーションを同一アプリケーションとして作成しました。

下記のハンズ・オンでは、1つの GlassFish クラスタを作成し、クラスタ内に複数のインスタンスを作成し、クラスタ環境でアプリケーションの動作確認を行います。

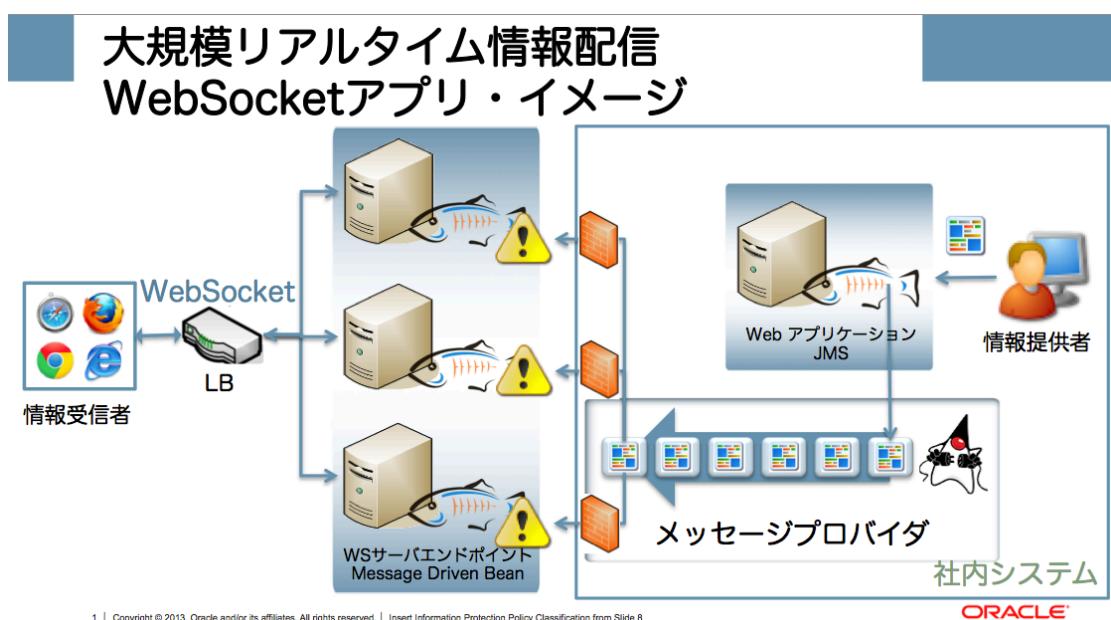


図 90：大規模クラスタ環境の構築概念図

補足

本アプリケーションは MQ を介してメッセージの送受信を行うため、情報提供者側、情報受信者側それぞれを別のアプリケーションとして作成し、全く別の GlassFish サーバ・インスタンスに対して配備することも可能です。情報提供者側のアプリケーションは DMZ 内のセキュリティに保護された環境に存在する GlassFish サーバ・インスタンスに配備し、情報受信者側のアプリケーションは外部ネットワークから接続可能な GlassFish のサーバ・インスタンスに配備する事でよりセキュアなメッセージ配信システムを構築することもできます。下記に示すハンズ・オンを正しく理解したのち、応用してセキュアな環境を構築してください。

まず、ブラウザで GlassFish の管理コンソールに接続⁸してください。

<http://localhost:4848>

接続したのち、左ペインより「クラスタ」を選択してください。選択すると下記の画面が表示されます。ここで「新規...」ボタンを押下してください。



図 91 : GlassFish クラスタの作成

⁸ asadmin のコマンドを使って構築する事もできますが、ここでは、かんたんに設定するために、GUI の管理コンソールを使用して環境構築を行います。

ボタンを押下すると下記の画面が右ペインに表示されます。ここで「クラスタ名:」に「my-cluster」を入力し、「作成するサーバ・インスタンス(0)」の「新規...」ボタンを2度押下してください。ボタンを押下すると「インスタンス名」、「重み」の入力項目、「ノード」のコンボボックスが2行表示されますので、1行目に「instance1」、「100」、「localhost-domain1」を2行目に「instance2」、「100」、「localhost-domain1」をそれぞれ入力し、最後に「OK」ボタンを押下してください。

新規クラスタ

構成を共有しないクラスタを作成するには、default-config構成と「選択している構成のコピーを作成します」オプションを選択します。構成を共有するクラスタを作成するには、別の構成と「選択している構成を参照する」オプションを選択します。ノードが存在しない場合は、ノードを作成してからクラスタにインスタンスを追加します。ノードを作成するには、「ノード」ページを使用します。

クラスタ名: *	my-cluster	OK 取消												
構成:	default-config	default-config構成は、コピーのみが可能で、参照することはできません。												
	<input checked="" type="radio"/> 選択している構成のコピーを作成する	<input type="radio"/> 選択している構成を参照する												
Message Queueクラスタ構成タイプ:	<input checked="" type="radio"/> デフォルト: マスター・プローカが設定された組込みの通常のクラスタ <input type="radio"/> カスタム													
作成するサーバー・インスタンス (2) <table border="1"> <thead> <tr> <th>Select</th> <th>インスタンス名</th> <th>重み</th> <th>ノード</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/></td> <td>instance1</td> <td>100</td> <td>localhost-domain1</td> </tr> <tr> <td><input type="checkbox"/></td> <td>instance2</td> <td>100</td> <td>localhost-domain1</td> </tr> </tbody> </table>			Select	インスタンス名	重み	ノード	<input type="checkbox"/>	instance1	100	localhost-domain1	<input type="checkbox"/>	instance2	100	localhost-domain1
Select	インスタンス名	重み	ノード											
<input type="checkbox"/>	instance1	100	localhost-domain1											
<input type="checkbox"/>	instance2	100	localhost-domain1											

図 92 : GlassFish 新規クラスタ

ボタンを押下すると、右ペインに下記の画面が表示されます。

クラスタ

GlassFish Serverクラスタを作成および管理します。クラスタとは、高可用性、スケーラビリティ、ロード・バランシングおよび障害保護を実現するGlassFish Serverインスタンスの集合体です。

クラスタ (1)			
	名前	構成	インスタンス
<input checked="" type="checkbox"/>	my-cluster	my-cluster-config	instance1 停止 instance2 停止

図 93 : GlassFish クラスタの作成完了

ここで「Select」にチェックし「クラスタの起動」ボタンを押下してください。ボタンを押下すると下記の確認ダイアログ・ウィンドウが表示されますので、「OK」ボタンを押下してください。



図 94 : クラスタ起動確認
ダイアログ・ウィンドウ

クラスタが正常に起動すると右ペインに下記の画面が表示されます。ここで「instance1」、「instance2」がそれぞれ「稼働中」と表示されていることを確認してください。

クラスタ

GlassFish Serverクラスタを作成および管理します。クラスタとは、高可用性、スケーラビリティ、ロード・バランシングおよび障害保護を実現するGlassFish Serverインスタンスの集合体です。

クラスタ (1)							
	名前	構成	インスタンス				
<input type="checkbox"/>	my-cluster	my-cluster-config	<table border="1"><tr><td>instance1</td><td><input checked="" type="checkbox"/> 稼働中</td></tr><tr><td>instance2</td><td><input checked="" type="checkbox"/> 稼働中</td></tr></table>	instance1	<input checked="" type="checkbox"/> 稼働中	instance2	<input checked="" type="checkbox"/> 稼働中
instance1	<input checked="" type="checkbox"/> 稼働中						
instance2	<input checked="" type="checkbox"/> 稼働中						

図 95 : GlassFish クラスタ起動完了

次に、既存の JMS リソースをクラスタ環境でも利用できるように設定変更してください。

左ペインの「JMS リソース」ツリーを展開し、「接続ファクトリ」から「jms/topicCon」を選択してください。選択すると下記の画面が表示されます。

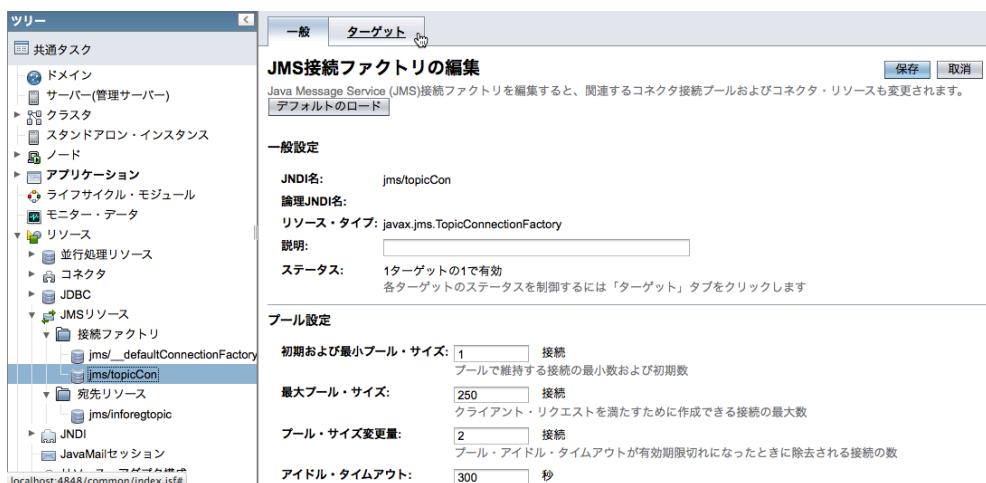


図 96 : JMS 接続ファクトリの編集

次に、右ペインより「ターゲット」タブを選択してください。選択すると右ペインに下記の画面が表示されます。

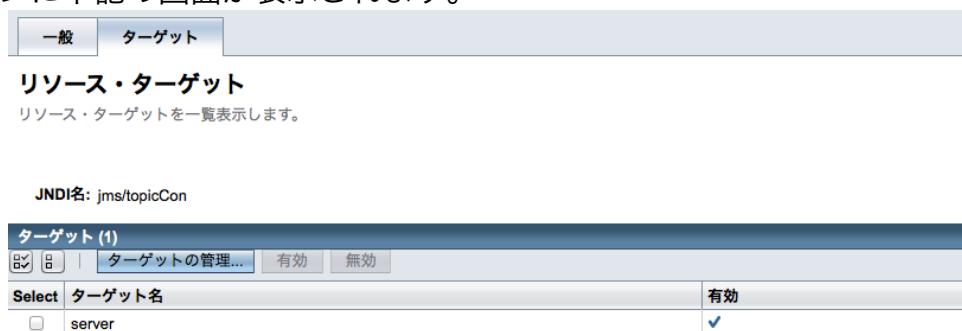


図 97 : JMS リソース・ターゲットの設定

ここで「ターゲットの管理...」ボタンを押下してください。押下すると下記の画面が表示されます。ここで「使用可能なターゲット：」より「my-cluster」を選択し、「追加 >」ボタン、もしくは「すべてを追加 >>」ボタンを押下してください。「選択したターゲット:」に「my-cluster」が含まれていることを確認し、最後に「保存」ボタンを押下してください。



図 98 : JMS リソース・ターゲットの管理

ボタンを押下すると下記の画面が表示されます。



図 99 : JMS 接続ファクトリ・リソースの設定完了

Java EE 7 Hands-on Lab using GlassFish 4

次に、左ペインの「JMS リソース」ツリーを展開し、「宛先リソース」から「jms/inforegtopic」を選択してください。選択すると下記の画面が表示されます。



図 100 : JMS 宛先リソースの設定

次に、右ペインより「ターゲット」タブを選択してください。選択すると下記の画面が表示されます。



図 101 : JMS 宛先リソースのターゲット

ここで「ターゲットの管理...」ボタンを押下してください。押下すると下記の画面が表示されます。ここで「使用可能なターゲット：」より「my-cluster」を選択し、「追加 >」ボタン、もしくは「すべてを追加 >>」ボタンを押下してください。「選択したターゲット:」に「my-cluster」が含まれていることを確認し、最後に「保存」ボタンを押下してください。



図 102 : JMS 宛先リソース・ターゲットの管理

ボタンを押下すると下記の画面が表示されます。

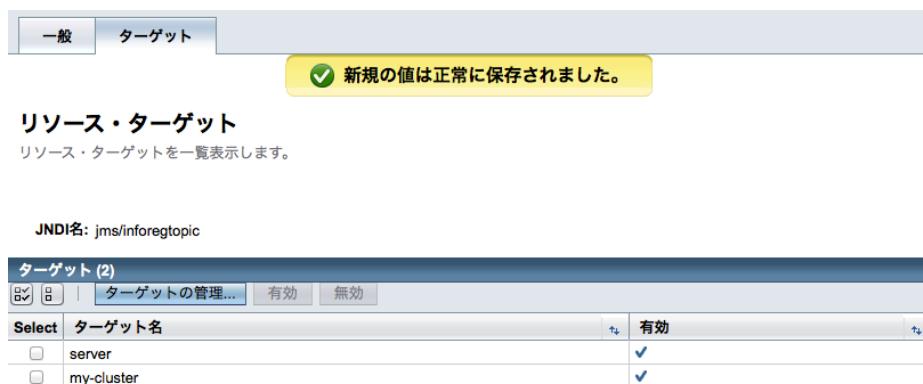


図 103 : JMS 宛先リソース設定完了

Java EE 7 Hands-on Lab using GlassFish 4

最後に、既存のデプロイ済の「WebSocket-HoL」アプリケーションをクラスタ環境でも利用できるように設定変更してください。

左ペインの「アプリケーション」ツリーを展開し、「WebSocket-HoL」を選択してください。選択すると下記の画面が表示されます。

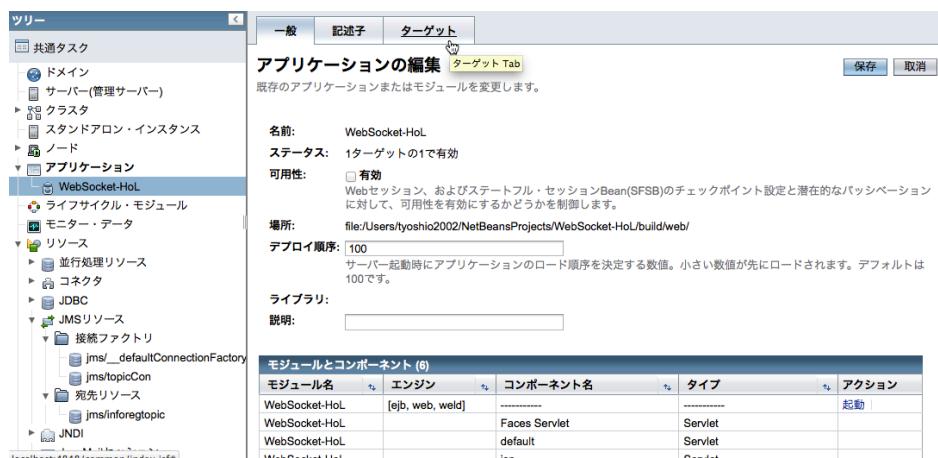


図 104：クラスタ環境におけるアプリケーションの有効化

次に、右ペインより「ターゲット」タブを選択してください。選択すると下記の画面が表示されます。



図 105：アプリケーション・ターゲット管理

ここで「ターゲットの管理...」ボタンを押下してください。押下すると下記の画面が表示されます。ここで「使用可能なターゲット：」より「my-cluster」を選択し、「追加 >」ボタン、もしくは「すべてを追加 >>」ボタンを押下してください。「選択したターゲット:」に「my-cluster」が含まれていることを確認し、最後に「保存」ボタンを押下してください。



図 106：アプリケーション・ターゲットの管理

ボタンを押下すると下記の画面が表示されます。

新規の値は正常に保存されました。

アプリケーション・ターゲット

アプリケーション・ターゲットを一覧表示します。

ターゲット (2)

Select	ターゲット名	有効	LB有効	仮想サーバー
<input type="checkbox"/>	my-cluster	true	true	仮想サーバーの管理
<input type="checkbox"/>	server	true	true	仮想サーバーの管理

図 107：クラスタ環境におけるアプリケーションの有効化

アプリケーションがクラスタ環境で有効になりましたので、クラスタの各インスタンスに接続しアプリケーションが正常に動作しているか確認を行ってください。動作確認を行うために、まずクラスタ内の各インスタンスがどのポート番号で HTTP リクエストを待ち受けているかを確認します。左ペインの「クラスタ」ツリーを展開し、「my-cluster」を選択してください。選択すると下記の画面が表示されます。

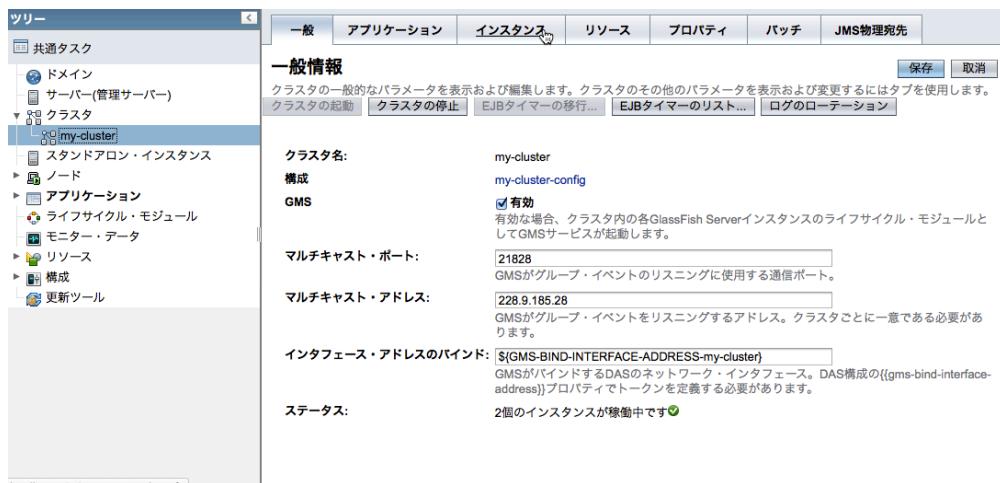


図 108 : GlassFish クラスタの各インスタンスの詳細確認

ここで「インスタンス」タブを選択してください。選択すると下記の画面が表示されます。ここで「instance1」のリンクを押下してください。

一般	アプリケーション	インスタンス	リソース	プロパティ	パッチ	JMS物理宛先
クラスタ化されたサーバー・インスタンス						
現在のクラスタのクラスタ化されたGlassFish Serverインスタンスを作成および管理します。						
クラスタ名: my-cluster						
サーバー・インスタンス (2)						
<input type="button" value="新規..."/> <input type="button" value="削除"/> <input type="button" value="起動"/> <input type="button" value="停止"/> <input type="button" value="...他の操作..."/>						
Select	名前	LBの重み	構成	ノード	ステータス	
<input type="checkbox"/>	instance1	100	my-cluster-config	localhost-domain1	稼働中	
<input type="checkbox"/>	instance2	100	my-cluster-config	localhost-domain1	稼働中	

図 109 : クラスタ化されたサーバ・インスタンス一覧

リンクを押下すると下記の画面が表示されます。ここで「プロパティ」タブを選択してください。



図 110：インスタンスの一般情報

「プロパティ」タブを選択すると下記の画面が表示されます。ここで「HTTP_LISTENER_PORT」と記載されている変数名の「現在の値」を確認してください。



図 111：インスタンスのシステム・プロパティ

同様に、「instance2」のHTTPリスナーポート番号を調べてください。
上記のシステムでは、「instance1」に「28080」が「instance2」に
「28081」が設定されてました。

ブラウザで、各インスタンスに接続し稼働していることを確認してください。

http://localhost:28080
http://localhost:28081

それぞれのインスタンスで、下記のように正常に表示されていることを確認してください。

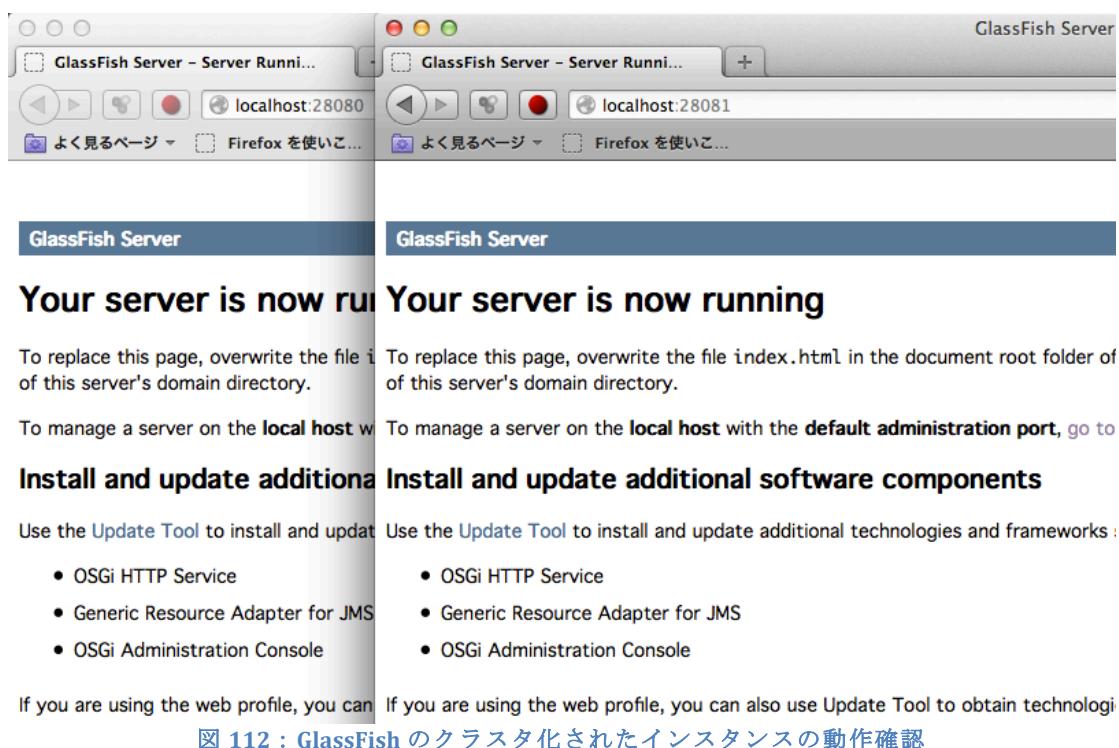


図 112 : GlassFish のクラスタ化されたインスタンスの動作確認

各インスタンスの HTTP ポート番号が分かりましたので、WebSocket アプリケーションに接続してください。ここで、「サーバ接続ポート番号」に一つのブラウザでは「localhost:28080」を、そしてもう一つのブラウザには「localhost:28081」と入力しそれぞれ「Connect」ボタンを押下してください。

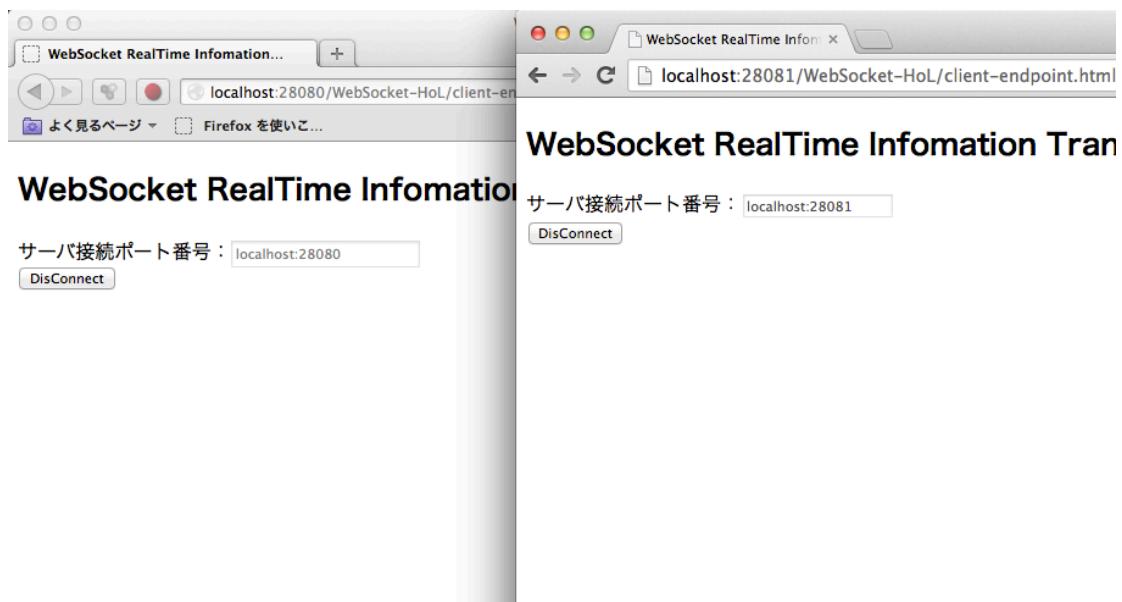


図 113：情報受信者用の画面の表示

Java EE 7 Hands-on Lab using GlassFish 4

次に、情報提供者側の JSF のアプリケーションを別のブラウザ、もしくはタブで表示してください。

`http://localhost:28080/WebSocket-HoL/`
もしくは、
`http://localhost:28081/WebSocket-HoL/`

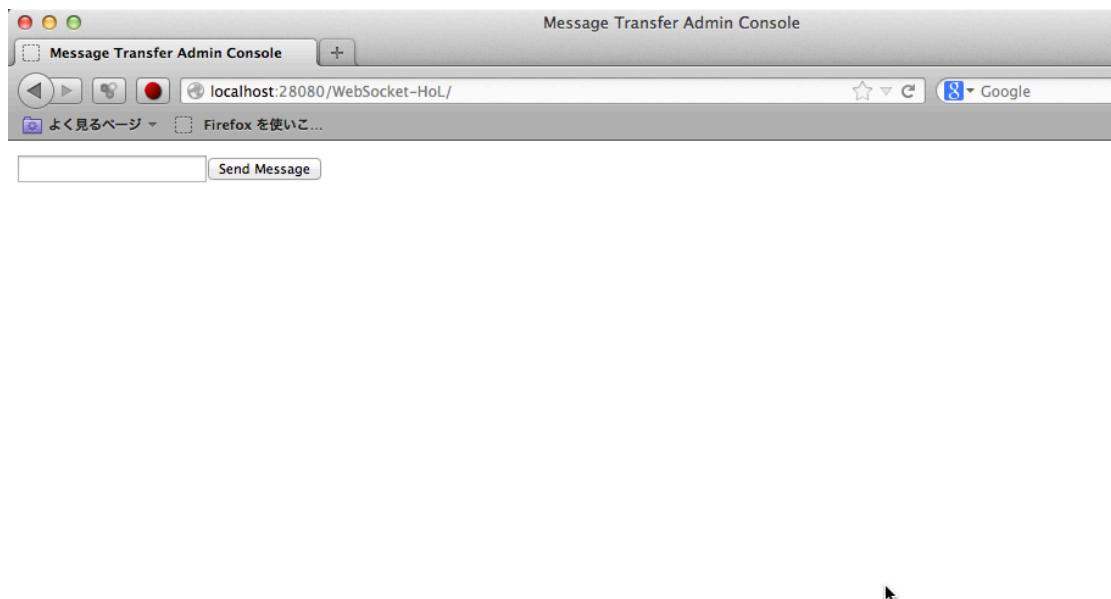


図 114：情報提供者用の画面表示

全ての画面を表示したのち、情報提供者用の画面から文字を入力してください。情報受信者用の画面に同じ文字が表示されることを確認できます。

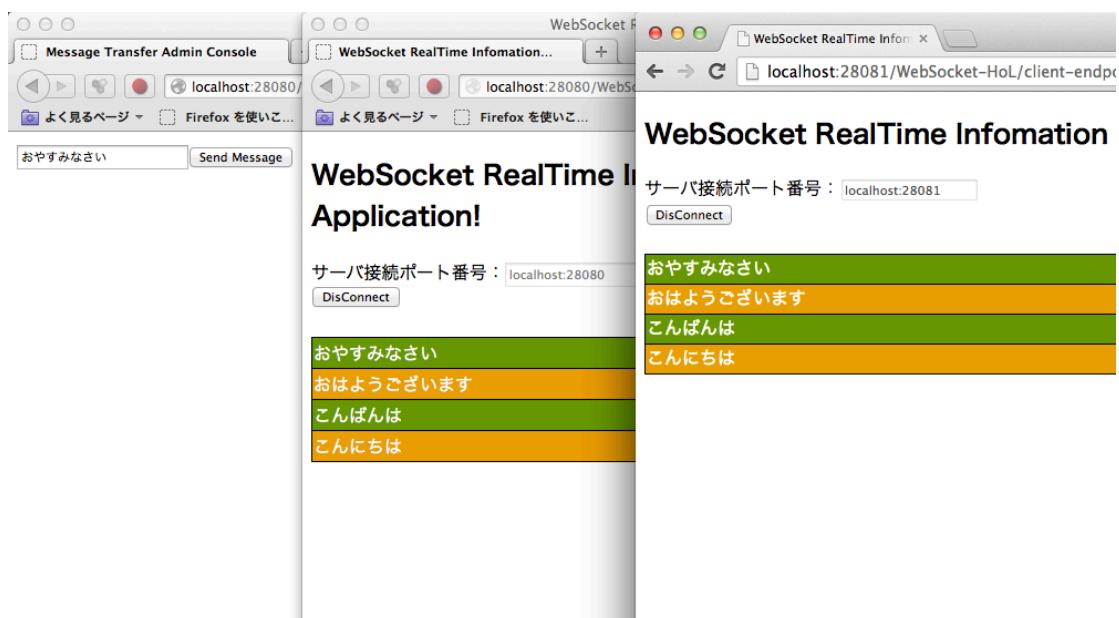


図 114：クラスタ環境でのアプリケーション実行

今回、GlassFish のクラスタ環境で WebSocket アプリケーションを作成しました。ステップバイステップで HoL 資料を作成しているためページ数は多くなっていますが、実際のコード記述量は決して多くありません。

JMS と組み合わせることでクラスタ環境でもアプリケーションの変更は一切なく WebSocket アプリケーションを大規模に展開できることがわかりました。今回は、リアルタイムのメッセージ配信システムを構築しましたが、今回のアプリケーションを応用し、またアイディア次第で様々な大規模 WebSocket アプリケーションを構築できるようになるかと思います。是非 Java EE 7 の機能を色々とお試しください。

参考資料・補足

本ハンズオンラボの NetBeans プロジェクトの全ソースコード
<https://github.com/yoshioterada/JavaEE7-HoL/>

Java EE SDK Download
<http://www.oracle.com/technetwork/java/javaee/downloads/index.html>

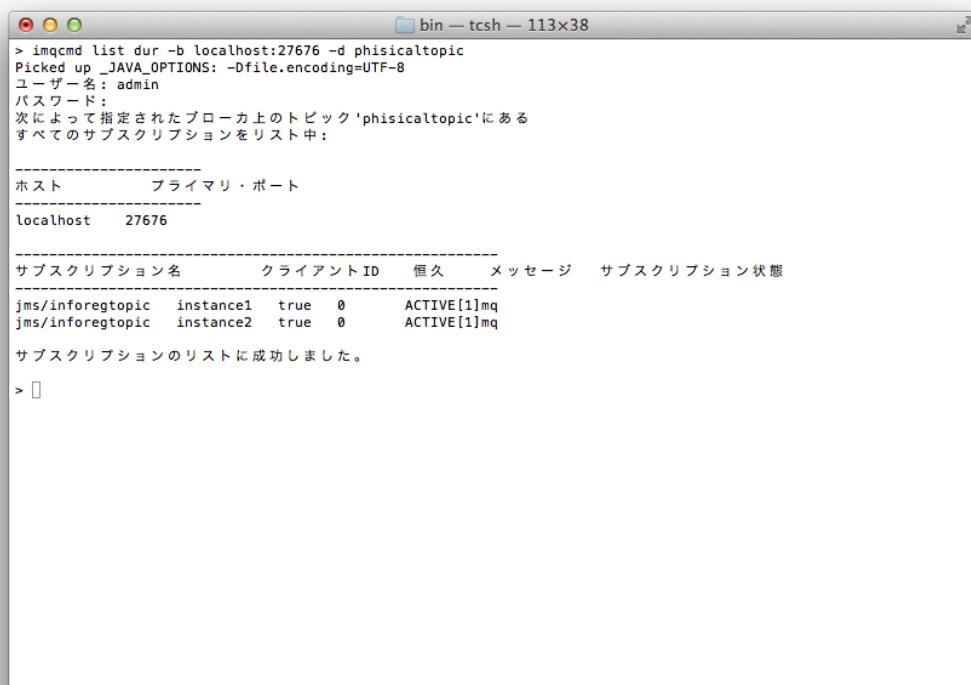
GlassFish/OpenMQ 関連ドキュメント
<https://glassfish.java.net/documentation.html>

Java EE 7 詳細説明資料
<http://www.slideshare.net/OracleMiddleJP/java-ee-7-detail>

The Java EE 7 Tutorial
<http://docs.oracle.com/javaee/7/tutorial/doc/>

OpenMQ の imqcmd を利用して、本アプリケーション用に作成した Topic : phisicaltopic のサブスクリーブを確認してみます。下記の imqcmd を実行し、クライアント ID にインスタンス名(instance1, instance2)が表示され ACTIVE になっていることを確認できます。OpenMQ の状態を確認するためには、imqcmd を使用し確認できます。詳しくはヘルプをご参照ください。

※ ユーザ名 : admin, パスワード : admin で imqcmd を実行できます。



The screenshot shows a terminal window titled "bin — tcsh — 113x38". The command "imqcmd list dur -b localhost:27676 -d phisicaltopic" is run, followed by prompts for Java options, user name (admin), and password. The output lists two subscribers (instance1 and instance2) for the topic "jms/inforegtopic" with status ACTIVE[1]mq.

```
> imqcmd list dur -b localhost:27676 -d phisicaltopic
Picked up _JAVA_OPTIONS: -Dfile.encoding=UTF-8
ユーザー名: admin
パスワード:
次によって指定されたプローカ上のトピック 'phisicaltopic' にある
すべてのサブスクリプションをリスト中:

-----
ホスト      プライマリ・ポート
-----
localhost    27676

-----
サブスクリプション名      クライアントID      恒久      メッセージ      サブスクリプション状態
-----
jms/inforegtopic    instance1    true    0      ACTIVE[1]mq
jms/inforegtopic    instance2    true    0      ACTIVE[1]mq

サブスクリプションのリストに成功しました。
> █
```

図 13 : imqcmd の実行例

以上