JBoss Enterprise Application Platform 5

Seam リファレンスガイド

JBoss Enterprise Application Platform 5 のユーザー向け

エディッション 5.1.2

Last Updated: 2017-10-26

JBoss Enterprise Application Platform 5 Seam リファレンスガイド

JBoss Enterprise Application Platform 5 のユーザー向け
エディッション 5.1.2

Gavin King

Pete Muir

Norman Richards

Shane Bryzak

Michael Yuan

Mike Youngstrom

Christian Bauer

Jay Balunas

Dan Allen

Max Andersen

Emmanuel Bernard

Nicklas Karlsson

Daniel Roth

Matt Drees

Jacob Orshalick

Marek Novotny

編集者

Samson Kittoli

Laura Bailey

Elspeth Thorne

With contributions from

James Cobb

Stephen Wadeley
Mark Newton
Steven Ebersole
Michael Coucy
Nicola Benaglia
Stefano Travelli
Francesco Milesi
Jaromír Hradílek

法律上の通知

概要

本ガイドは JBoss Enterprise Application Platform 5 およびその修正リリース用のリファレンスガイドです。

目次

第1章 SEAM チュートリアル

1.1. SEAM サンプルの使用

Seam にはそのさまざまな機能の利用方法をデモ形式で示してくれるサンプルアプリケーションが備わっています。 本チュートリアルでは Seam のユーザーにその使い方を理解していただけるようにいくつかのサンプルを見ていくことにします。Seam のサンプルは Seam ディストリビューションの **examples** サブディレクトリに置かれています。 登録に関する最初の例は**examples/registration** ディレクトリにあります。

各サンプルは同じディレクトリの構造をしています。

- **view** ディレクトリには Web ページテンプレート、イメージ、スタイルシートなどビュー関連のファイルが入っています。
- **resources** ディレクトリにはデプロイメント記述子やその他の構成ファイルが入っています。
- **src** ディレクトリにはアプリケーションソースコードが入っています。

サンプルはすべて Ant **build.xml** よりビルドし実行されるため、始める前に Ant の最新版をインストールしておく必要がある点に注意してください。

1.1.1. JBoss AS でのサンプルの実行

サンプルは JBoss Enterprise Application Platform での使用向けに設定されています。共有ファイル **build.properties** (ご使用の Seam installation のルートフォルダ内) の **jboss.home** を JBoss AS installation の場所に設定する必要があります。

JBoss AS の場所を設定してアプリケーションサーバーを起動したら、 いずれのサンプルもそのディレクトリ内で **ant explode** と入力するとビルドとデプロイを行うことができます。 EAR (Enterprise Archive) としてパッケージ化されたサンプルは **/seam-*example*** のような URL にデプロイします。 *example* はサンプルフォルダの名前です。 これには例外がひとつあります。 サンプルフォルダが「seam」で始まる場合、 プレフィックスの「seam」は省略されます。 たとえば、 JBoss AS がポート 8080 で実行中の場合、 Registration サンプルの URL は **http://localhost:8080/seam-registration/** になるのに対し、 SeamSpace サンプルの URL は **http://localhost:8080/seam-space/** になります。

一方、サンプルが WAR としてパッケージ化されている場合は、**/jboss-seam-*example*** のような URL にデプロイします。

注記

groovybooking、 hibernate、 jpa、 spring など WAR としてしかデプロイできないサンプルがいくつかあります。

1.1.2. サンプルのテストの実行

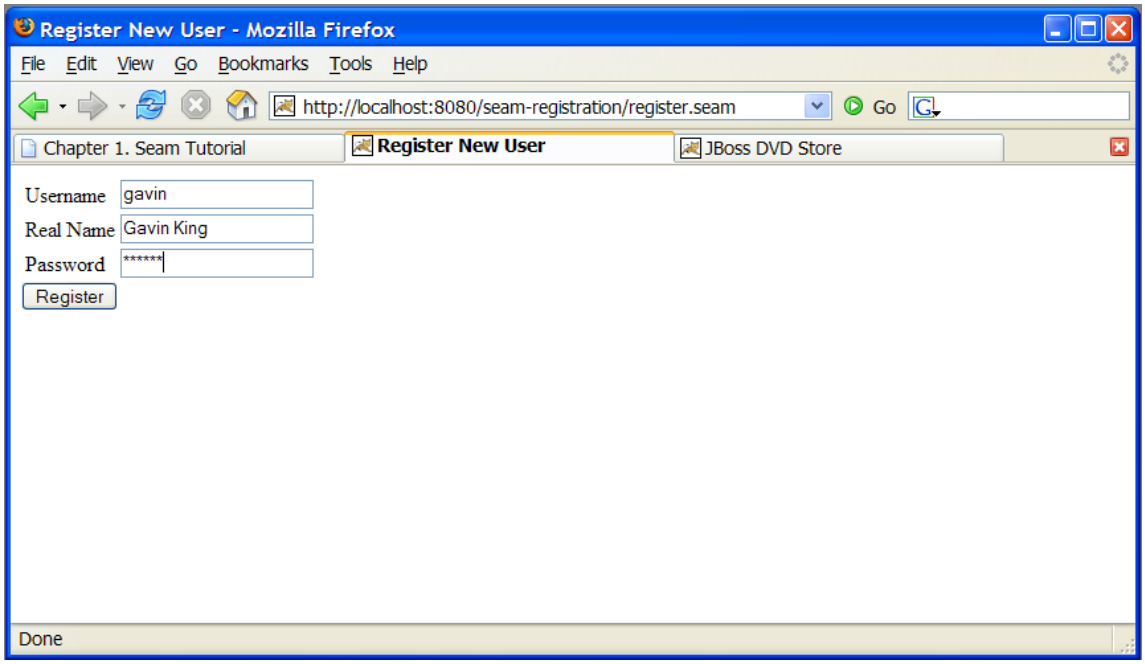
ほとんどのサンプルには TestNG 統合テストスィートが同梱しています。テストを実行する最も簡単な方法は **ant test** を実行することです。

TestNG プラグインを使って IDE 内でテストを実行することも可能です。そのためには、Eclipse IDE で Seam テストケースを実行またはデバッグする前に Ant テストを実行する必要があります。詳細は、Seam ディストリビューションのサンプルディレクトリにある readme.txt を確認してください。

1.2. 最初の SEAM アプリケーション: 登録サンプル

登録サンプルは新しいユーザーがそのユーザー名や実名、パスワードをデータベースに保存できるシンプルなアプリケーションです。このサンプルでは基本的機能のみを使用し、JSF アクションリスナーとしての EJB3 セッション Bean の使用や、Seam の基本設定を示しています。

最初のページは 3 つの入力フィールドを持つ基本的なフォームを表示します。試しに、項目を入力してフォームをサブミットしてください。ユーザーオブジェクトがデータベースに保存されます。



1.2.1. コードの理解

このサンプルは 2 つの Facelets テンプレートが実装されています。エンティティ Bean がひとつとステートレスセッション Bean がひとつです。本項ではコードを詳細に見ていきます。まずベースレベルから見てみましょう。

1.2.1.1. エンティティ Bean: `User.java`

ユーザーデータ用に EJB エンティティ Bean が必要です。このクラスはアノテーションによって **永続性** と **データ妥当性検証** を宣言的に定義しています。また、Seam コンポーネントとしてのクラスを定義するために別にいくつかのアノテーションも必要です。

例1.1 User.java

```
@Entity                          (1)
@Name("user")                    (2)
@Scope(SESSION)                  (3)
```

```
@Table(name="users")
public class User implements Serializable
{
    private static final long serialVersionUID =
        1881413500711441951L;

    private String username;
    private String password;
    private String name;

    public User(String name,
                String password,
                String username)
    {
        this.name = name;
        this.password = password;
        this.username = username;
    }

    public User() {}

    @NotNull @Length(min=5, max=15)
    public String getPassword()
    {
        return password;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }

    @NotNull
    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    @Id @NotNull @Length(min=5, max=15)
    public String getUsername()
    {
        return username;
    }
```

```
    public void setUsername(String username)
    {
        this.username = username;
    }

}
```

❶	EJB3 標準 **@Entity** アノテーションは、**User** クラスがエンティティ Bean であることを示しています。
❷	Seam コンポーネントには **@Name** アノテーションで指定される**コンポーネント名**が必要です。この名前は Seam アプリケーション内で一意である必要があります。JSF が Seam に Seam コンポーネント名と同じ名前を持つコンテキスト変数を解決するよう指示し、コンテキスト変数が現在未定義 (null) である場合、Seam はそのコンポーネントをインスタンス化し、新しいインスタンスをコンテキスト変数にバインドします。この場合、JSF が **user** という変数に初めて出会うときに Seam が **User** をインスタンス化します。
❸	Seam がコンポーネントをインスタンス化すると必ず、新しいインスタンスをコンポーネントの **default context** のコンテキスト変数にバインドします。デフォルトのコンテキストは **@Scope** アノテーションを使用して指定されます。**User** Bean はセッションスコープのコンポーネントです。
❹	EJB 標準 **@Table** アノテーションは、**User** クラスが **users** テーブルにマッピングされることを示しています。
❺	**name**、**password**、**username** はエンティティ Bean の永続属性です。すべての永続属性は、アクセサメソッドを定義します。これはレスポンス出力フェーズおよびモデル値の更新フェーズで JSF によりこのコンポーネントが使用されるときに必要です。
❻	空のコンストラクタは、EJB 仕様と Seam の両方で必要です。
❼	**@NotNull** と **@Length** アノテーションは、 Hibernate Validator フレームワークの一部です。Seam は Hibernate Validator を統合するため、 データの妥当性検証にこれを使用することができます (永続性に Hiberenate を使用していない場合でも使用できます)。
❽	EJB 標準 **@Id** アノテーションは、 エンティティ Bean の主キーであることを示しています。

このサンプルで、最も注目してほしい重要なものは **@Name** と **@Scope** アノテーションです。 このアノテーションは、このクラスが Seam コンポーネントであることを規定しています。

次項では、 **User** クラスのプロパティが直接 JSF コンポーネントにバインドされ、モデル値の更新フェーズで JSF により埋められていることがわかります。 JSP ページとエンティティ Bean ドメインモデル間でデータのコピーを何度も行うためのグルーコードは必要ありません。

ただし、 エンティティ Bean はトランザクション管理やデータベースアクセスを行わないので、 このコンポーネントを JSF のアクションリスナーとしては使用しないでください。 この場合、 セッション Bean の方がよいでしょう。

1.2.1.2. ステートレスセッション Bean クラス:RegisterAction.java

ほとんどの Seam アプリケーションではセッション Bean が JSF アクションリスナーとして使用されますが、 JavaBean を使用しても構いません。

このアプリケーションにはちょうどひとつだけ JSF アクションがあり、これにセッション Bean メソッドがひとつリンクしています。この場合、 アクションに関連付けられた状態はすべて **User** Bean で保持されるためステートレスセッション Bean を使用します。

関連コードを以下に示します。

例1.2 RegisterAction.java

```
@Stateless
```

```
@Name("register")
public class RegisterAction implements Register
{
    @In
```

```
    private User user;

    @PersistenceContext
```

```
    private EntityManager em;

    @Logger
```

```
    private Log log;

    public String register()
```

```
    {
        List existing = em.createQuery("select username " +
                                       "from User " +
                                       "where username = #
```

```
{user.username}")
            .getResultList();

        if (existing.size()==0)
            {
                em.persist(user);
                log.info("Registered new user #{user.username}");
```

```
                return "/registered.xhtml";
```

```
            }
        else
```

```
            {
                FacesMessages.instance().add("User #{user.username}
                                                9
already exists");
                return null;
            }
    }

}
```

1	EJB **@Stateless** アノテーションはこのクラスがステートレスセッション Bean であることを示しています。
2	**@In** アノテーションは Bean の属性を Seam でインジェクトされているとマークします。この場合、属性は **user** (インスタンス変数名) という名前のコンテキスト変数からインジェクトされます。
3	EJB 標準 **@PersistenceContext** アノテーションは、 EJB3 エンティティマネージャにインジェクトするために使用されます。
4	Seam **@Logger** アノテーションはコンポーネントの**Log** インスタンスをインジェクトするために使用されます。
5	アクションリスナーメソッドは標準 EJB3 **EntityManager** API を使用して、データベースとのやり取りを行い JSF 結果を返します。 これはセッション Bean であるため、**register()** メソッドが呼び出されるとトランザクションは自動的に開始され、 このメソッドが終了するとトランザクションがコミットされる点に注意してください。
6	Seam では EJB-QL 内で JSF EL 式を使用することができる点に注目してください。これにより普通の JPA **setParameter()** が標準 JPA **Query** オブジェクトで呼び出されることになります。
7	**Log** API によりテンプレートから作成されたログメッセージを簡単に表示することができ、JSF EL 式を使用することも可能です。
8	JSF アクションリスナーメソッドは、次にどのページを表示するかを決定する文字列値の結果を返します。 null の結果 (あるいは void アクションリスナーメソッド) の場合は前のページを再表示します。純粋な JSF では、 常に JSF **ナビゲーションルール**を使用してその結果から JSF ビュー ID を決定するのが普通です。 複雑なアプリケーションの場合はこの間接化が役立ち、適した方法となります。ただし、このような非常に簡単なサンプルの場合 Seam では結果として JSF ビュー ID を使用することができるため、ナビゲーションルールは必要なくなります。**結果としてビュー ID を使用している場合は Seam は常にブラウザリダイレクトを行う点に注意してください**。
9	Seam は一般的な問題の解決に役立つ **組み込みコンポーネント** をいくつか提供しています。**FacesMessages** コンポーネントを使用すると、テンプレートで作成したエラーや成功のメッセージを表示することは容易です (Seam 2.1 では代わりに **StatusMessages** を使用して JSF へのセマンティック依存を取り除くことが可能です)。Seam 組み込みコンポーネントはインジェクションで取得するか、組み込みコンポーネントのクラスで **instance()** メソッドを呼び出すことで取得できます。

ここでは **@Scope** を明示的に指定していないので注意してください。 各 Seam コンポーネントタイプにはデフォルトのスコープがあり、明示的にスコープが指定されない場合に使用されます。ステートレスセッション Bean では、デフォルトスコープはステートレスコンテキストです。

セッション Bean のアクションリスナーはこの小さなアプリケーションのビジネスロジックと永続ロ

ジックを実行しています。 さらに複雑なアプリケーションでは、別々のサービス層が必要になるかもしれませんが、 Seam ではアプリケーションの階層化に独自のストラテジーを実装することができます。必要に応じてアプリケーションをシンプルにすることも複雑にすることも自在です。

注記

このアプリケーションは明確なサンプルコードを提示する目的で必要以上に複雑になっています。 アプリケーションコードはすべて Seam のアプリケーションフレームワークコントローラを使用すると除外できます。

1.2.1.3. セッション Bean ローカルインタフェース : Register.java

セッション Bean にはローカルインターフェースが必要です。

例1.3 Register.java

```
@Local
public interface Register
{
     public String register();
}
```

Java コードは以上です。 次は ビューを見ましょう。

1.2.1.4. ビュー : register.xhtml と registered.xhtml

JSF に対応する技術であればいずれを使用しても Seam アプリケーションのビューページを実装することができます。 このサンプルでは Facelets で記述しました。

例1.4 register.xhtml

```
<?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:s="http://jboss.com/products/seam/taglib"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <head>
    <title>Register New User</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <s:validateAll>
          <h:panelGrid columns="2">
            Username: <h:inputText value="#{user.username}"
            required="true"/>
            Real Name: <h:inputText value="#{user.name}"
            required="true"/>
            Password: <h:inputSecret value="#{user.password}"
```

```
            required="true"/>
          </h:panelGrid>
        </s:validateAll>
        <h:messages/>
        <h:commandButton value="Register" action="#
{register.register}"/>
      </h:form>
    </f:view>
  </body>

</html>
```

ここで Seam 固有のタグとなるのは **<s:validateAll>** のみです。 この JSF コンポーネントは含まれるすべての入力フィールドをエンティティ Bean で指定される Hibernate Validator のアノテーションに対して検証するよう JSF に指示します。

例1.5 registered.xhtml

```
<?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:f="http://java.sun.com/jsf/core">

  <head>
    <title>Successfully Registered New User</title>
  </head>
  <body>
    <f:view>
      Welcome, #{user.name}, you are successfully
      registered as #{user.username}.
    </f:view>
  </body>

</html>
```

上記はインライン EL で作成したシンプルな Facelets ページです。 Seam 固有のものは何も含まれていません。

1.2.1.5. Seam コンポーネントデプロイメント記述子 :components.xml

デプロイメント記述子を見ていく前に、 Seam が最小限の設定を非常に重視していることは注目に値します。 これらの設定ファイルは Seam アプリケーションを作成すると自動的に生成されます。また、これらの設定ファイルを変更する必要性が生じることはめったにないでしょう。 ここに設定ファイルを示す目的は、 単に全サンプルコードの目的と機能の理解に役立てていただくためだけです。

以前に Java フレームワークを使用したことがある方なら、 XML ファイルにコンポーネントクラスを宣言することにも慣れていくことでしょう。 また プロジェクトが進化するに従い XML ファイルの管理が難しくなっていくことに気付かれると思います。 幸い、 Seam ではアプリケーションコンポーネントに XML を付随する必要がありません。 ほとんどの Seam アプリケーションは XML のほんの一部しか必要とせず、 プロジェクトが拡大してもこのサイズが増大していくことはありません。

ただし、特定のコンポーネント、特に Seam に組み込まれるコンポーネントなどに関する何らかの外部設定を提供できると便利な場合がよくあります。最も柔軟性のあるオプションは、**WEB-INF** ディレクトリにある **components.xml** と呼ばれるファイルにこの設定を与えることです。**components.xml** ファイルを使用して Seam に JNDI での EJB コンポーネントの検索方法を指示することができます。

例1.6 components.xml のサンプル

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
            "http://jboss.com/products/seam/core
             http://jboss.com/products/seam/core-2.2.xsd
             http://jboss.com/products/seam/components
             http://jboss.com/products/seam/components-2.2.xsd">

  <core:init jndi-pattern="@jndiPattern@"/>

</components>
```

上記のコードは **org.jboss.seam.core.init** という名前の Seam 組み込みコンポーネントに属する **jndiPattern** という名前のプロパティを設定します。アプリケーションがデプロイされるときに、@ 記号を使って Ant ビルドスクリプトが components.properties ファイルから正しい JNDI パターンを挿入するよう指示しています。このプロセスの詳細については 「**components.xml** によるコンポーネントの設定」を参照してください。

1.2.1.6. WEB デプロイメント記述子 : web.xml

この小さなアプリケーションのプレゼンテーション層は WAR にデプロイされます。したがって、Web デプロイメント記述子が必要です。

例1.7 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
         "http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5">

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

  <context-param>
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
  </context-param>

  <servlet>
```

```
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.seam</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>10</session-timeout>
  </session-config>

</web-app>
```

上記の **web.xml** ファイルは Seam と JSF の両方を設定します。 ここで見る設定は Seam のアプリケーション間でほとんど変わりません。

1.2.1.7. JSF 設定 : faces-config.xml

ほとんどの Seam アプリケーションはプレゼンテーション層として JSF ビューを使用します。 従って通常 **faces-config.xml** が必要です。 この場合、 ビュー定義に Facelets を使用しますので、JSF にテンプレートエンジンとして Faceles を使用することを指示する必要があります。

例1.8 faces-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation=
              "http://java.sun.com/xml/ns/javaee
               http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
              version="1.2">

  <application>
    <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
  </application>

</faces-config>
```

管理 Bean はアノテーション付きの Seam コンポーネントであるため、 JSF 管理 Bean 宣言は必要ないので注意してください。 Seam アプリケーションでは **faces-config.xml** の使用頻度は純粋な JSF に比べると非常に少なくなります。 ここでは Facelets (および JSP 以外) をビューハンドラとして有効にするためだけに使用します。

基本的な記述子の設定がすべて完了すると、 Seam アプリケーションに機能を追加するため記述する必要がある XML は ナビゲージョンルールまたは jBPM プロセス定義の編成に対してのみとなります。 Seam は **プロセスフロー** と **設定データ** はすべて XML に属するという原則で動作します。

上記のサンプルではビュー ID がアクションコードに埋め込まれているためナビゲーションルールは必要ありません。

1.2.1.8. EJB デプロイメント記述子 : ejb-jar.xml

ejb-jar.xml ファイルはアーカイブ中のすべてのセッション Bean に **SeamInterceptor** を付加することによって Seam を EJB3 と統合します。

例1.9 ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
         "http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">

  <interceptors>
    <interceptor>
      <interceptor-class>
        org.jboss.seam.ejb.SeamInterceptor
      </interceptor-class>
    </interceptor>
  </interceptors>

  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>
        org.jboss.seam.ejb.SeamInterceptor
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>

</ejb-jar>
```

1.2.1.9. EJB 永続デプロイメント記述子 : persistence.xml

persistence.xml ファイルは EJB 永続プロバイダに適切なデータソースを指示し、 またベンダー固有の設定を含んでいます。 このサンプルでは、 起動時に自動スキーマエキスポートを有効にします。

例1.10 persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation=
             "http://java.sun.com/xml/ns/persistence

http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
             version="1.0">
```

```
    <persistence-unit name="userDatabase">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
        <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
    </persistence-unit>

  </persistence>
```

1.2.1.10. EAR デプロイメント記述子 : application.xml

最後に、EAR としてアプリケーションがデプロイされるため、 デプロイメント記述子も必要になります。

例1.11 登録アプリケーション

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/javaee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation=
             "http://java.sun.com/xml/ns/javaee
              http://java.sun.com/xml/ns/javaee/application_5.xsd"
             version="5">

  <display-name>Seam Registration</display-name>

  <module>
    <web>
      <web-uri>jboss-seam-registration.war</web-uri>
      <context-root>/seam-registration</context-root>
    </web>
  </module>
  <module>
    <ejb>jboss-seam-registration.jar</ejb>
  </module>
  <module>
    <ejb>jboss-seam.jar</ejb>
  </module>
  <module>
    <java>jboss-el.jar</java>
  </module>

</application>
```

このデプロイメント記述子はエンタープライズアーカイブのモジュールとリンクし、Web アプリケーションをコンテキストルート **/seam-registration** にバインドします。

これでアプリケーションにあるすべてのファイルを見てきました。

1.2.2. 動作内容

フォームがサブミットされたとき、 JSF は Seam に **user** という名前の変数を解決するよう要求します。 その名前にはまだバインドされた値がないため (どの Seam コンテキストにも)、 Seam は **user** コンポーネントをインスタンス化し、結果となる **User** エンティティ Bean インスタンスを Seam セッションコンテキストに保管してからそれを JSF に返します。

フォームの入力値は、**User** エンティティで指定された Hibernate Validator 制約に対してデータ整合性検証が行われるようになります。 制約に違反していると JSF はそのページを再表示します。 これ以外は、JSF はフォームの入力値を **User** エンティティ Bean のプロパティにバインドします。

次に、 JSF は Seam に **register** という名前の変数を解決するよう要求します。 Seam は前述した JNDI パターンを使ってステートレスセッション Bean を探し、それを Seam コンポーネントとしてラップしてから返します。 次に Seam がこのコンポーネントを JSF に提示すると JSF は **register()** アクションリスナーメソッドを呼び出します。

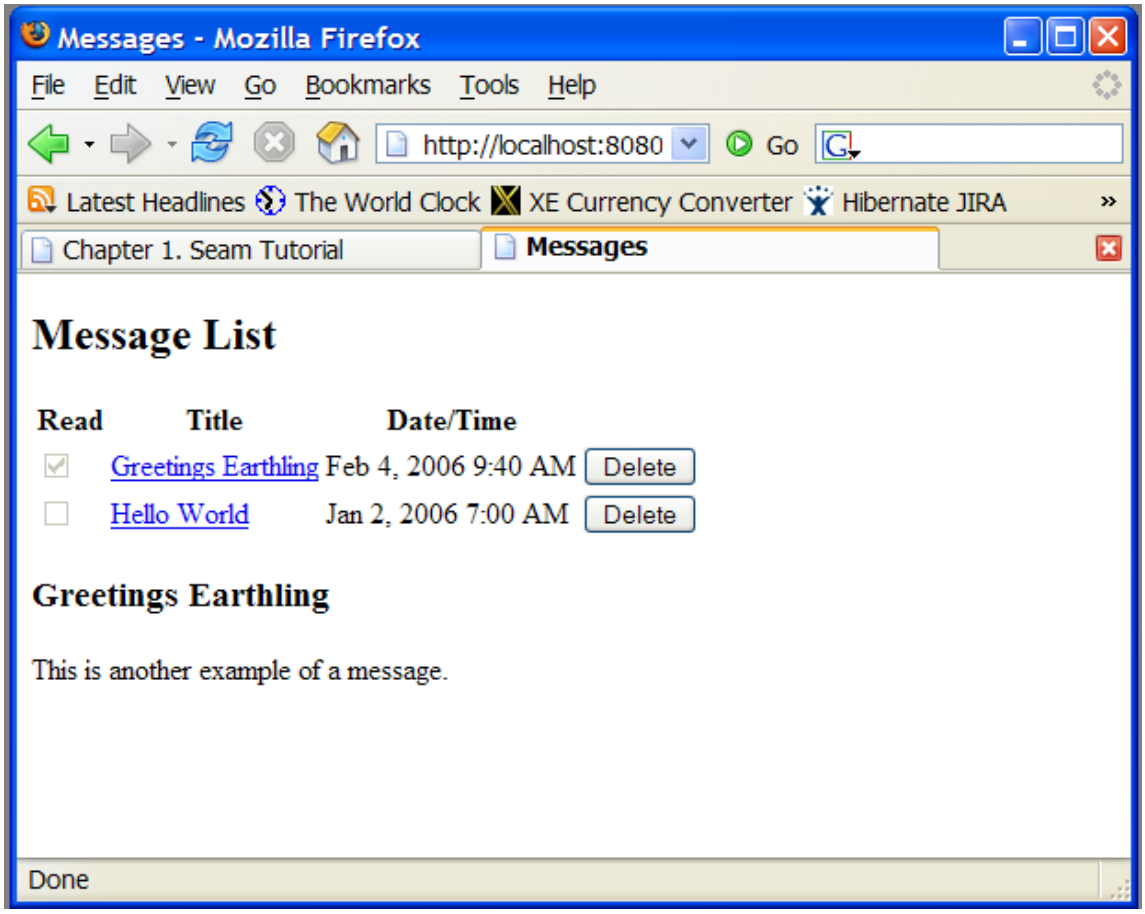
Seam はメソッド呼び出しをインターセプトし、呼び出しが進む前に Seam セッションコンテキストから **User** エンティティをインジェクトします。

register() メソッドは入力されたユーザー名が既に存在するかどうかを調べます。 存在した場合、エラーメッセージは **FacesMessages** コンポーネントでキューイングされ、 null 結果 が返されてページが再表示されることになります。**FacesMessages** コンポーネントはメッセージ文字列に組み込まれた JSF 式を補完し、 ビュー に JSF **FacesMessage** を追加します。

そのユーザー名を持つユーザーが存在しない場合、**"/registered.xhtml""** 結果により **registered.xhtml** ページへのブラウザリダイレクトが発生します。JSF がページのレンダリングに到達すると、 Seam に **user** という名前の変数の解決を要求し、 Seam のセッションスコープから返される **User** エンティティのプロパティ値を使用します。

1.3. SEAM のクリック可能な一覧 : メッセージサンプル

クリックするとデータベースの検索結果を一覧表示できる機能は、あらゆるオンラインアプリケーションの非常に重要な部分です。 Seam は JSF に加えて特殊な機能を提供することで EJB-QL や HQL でのデータ問い合わせを容易にし、 JSF **<h:dataTable>** を使ってクリック可能な一覧としてそれを表示します。メッセージサンプルがこの機能を示しています。



1.3.1. コードの理解

このメッセージサンプルは 1 つのエンティティ Bean (**Message**)、1 つのセッション Bean (**MessageListBean**)、1 つの JSP から構成されています。

1.3.1.1. エンティティ Bean : `Message.java`

Message エンティティ Bean は、タイトル、テキスト、メッセージの日付と時刻、そしてメッセージが既読か否かを示すフラグを定義します。

例1.12 Message.java

```
@Entity
@Name("message")
@Scope(EVENT)
public class Message implements Serializable {
    private Long id;
    private String title;
    private String text;
    private boolean read;
    private Date datetime;

    @Id @GeneratedValue
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
```

```
    @NotNull @Length(max=100)
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }

    @NotNull @Lob
    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }

    @NotNull
    public boolean isRead() {
        return read;
    }
    public void setRead(boolean read) {
        this.read = read;
    }

    @NotNull
    @Basic @Temporal(TemporalType.TIMESTAMP)
    public Date getDatetime() {
        return datetime;
    }
    public void setDatetime(Date datetime) {
        this.datetime = datetime;
    }
}
```

1.3.1.2. ステートフルセッション Bean : `MessageManagerBean.java`

前述のサンプル同様、 このサンプルは 1 つのセッション Bean (**`MessageManagerBean`**) から構成され、 フォームにある両方のボタンに対応するアクションリスナーメソッドを定義します。 前述のサンプルと同様、 ボタンの 1 つは一覧からメッセージを選択してそのメッセージを表示します。 もう 1 つのボタンはメッセージを削除します。

ただし、 はじめてメッセージ一覧のページに移動したときのメッセージの一覧取得も **`MessageManagerBean`** の役割となります。 ユーザーがこのページに到達するにはいろいろな経路があり、 必ずしも JSF アクションがすべての経路で先行するわけではありません。 (たとえば、 お気に入りからそのページに行く場合、 必ずしも JSF アクションを呼び出す必要はありません。) したがって、 メッセージ一覧の取得作業はアクションリスナーメソッドではなく Seam の **ファクトリメソッド** で行われなければなりません。

メッセージの一覧をサーバー要求にまたがってメモリにキャッシュしたいので、 ステートフルセッション Bean でこれを行います。

例1.13 MessageManagerBean.java

```
@Stateful
@Scope(SESSION)
@Name("messageManager")
public class MessageManagerBean
    implements Serializable, MessageManager
{
    @DataModel
```

```
    private List<Message> messageList;

    @DataModelSelection
```

```
    @Out(required=false)
```

```
    private Message message;

    @PersistenceContext(type=EXTENDED)
```

```
    private EntityManager em;

    @Factory("messageList")
```

```
    public void findMessages()
    {
        messageList = em.createQuery("select msg " +
                                     "from Message msg" +
                                     "order by msg.datetime desc")
                        .getResultList();
    }

    public void select()
```

```
    {
        message.setRead(true);
    }

    public void delete()
```

```
    {
        messageList.remove(message);
        em.remove(message);
        message=null;
    }

    @Remove
```

```
8
    public void destroy() {}

}
```

1	**@DataModel** アノテーションは、**java.util.List** タイプの属性を、**javax.faces.model.DataModel** インスタンスとして JSF ページに公開します。これにより、各行に対してクリック可能なリンクを持つ JSF **<h:dataTable>** 中の一覧を使用可能とします。 このサンプルでは、 **DataModel** は、 **messageList** という名前のセッションコンテキスト変数で利用可能です。
2	**@DataModelSelection** アノテーションは Seam にクリックされたリンクに該当する **List** エレメントをインジェクトするよう指示します。
3	次に **@Out** アノテーションは選択された値を直接ページに公開します。クリック可能な一覧の行が選択されるたびに **Message** がステートフル Bean の属性にインジェクトされ、続いて **message** というイベントコンテキスト変数に **アウトジェクト** されます。
4	このステートフル Bean は EJB3 **拡張永続コンテキスト** を持っています。この Bean が存在する限り、クエリで取得されたメッセージは管理状態に留まります。そのため、ステートフル Bean への後続のメッセージ呼び出しは、**EntityManager** へ明示的な呼び出しを行わずにそのメッセージを更新することができます。
5	初めて JSP ページに移動するとき、 **messageList** コンテキスト変数には値がありません。**@Factory** アノテーションは Seam に **MessageManagerBean** インスタンスを作成し、**findMessages()** メソッドを呼び出し、値を初期化するよう指示します。**findMessages()** を **messages** の **ファクトリメソッド** と呼びます。
6	**select()** アクションリスナーメソッドは、 選択された **Message** に既読マークを付け、 データベース中のそれを更新します。
7	**delete()** アクションリスナーメソッドは、 選択された **Message** をデータベースから削除します。
8	ステートフルセッション Bean の Seam コンポーネントは **@Remove** とマークされたパラメータを持たないメソッドを定義する **必要があります**。Seam コンテキストが終了すると、Seam はステートフル Bean を削除しサーバー側の状態を消去します。

注記

これはセッションスコープの Seam コンポーネントです。 ユーザーログインのセッションと関連付けられ、 ログインセッションからの要求はすべて同じコンポーネントのインスタンスを共有します。 Seam アプリケーションではセッションスコープのコンポーネントは通常、控えめに使用されます。

1.3.1.3. セッション Bean ローカルインタフェース : MessageManager.java

すべてのセッション Bean にビジネスインターフェースがあります。

例1.14 MessageManager.java

```
@Local
public interface MessageManager {
    public void findMessages();
    public void select();
    public void delete();
    public void destroy();
}
```

この時点から、ローカルインターフェースはこれらのコードサンプルには表示されなくなります。**Components.xml**、**persistence.xml**、**web.xml**、**ejb-jar.xml**、**faces-config.xml**、**application.xml** については前述までのサンプルとほぼ同じなので、JSP に進みます。

1.3.1.4. ビュー: messages.jsp

この JSP ページは JSF **<h:dataTable>** コンポーネントを使用した簡単なものです。繰り返しになりますが、この機能も Seam 固有ではありません。

例1.15 messages.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
  <head>
    <title>Messages</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <h2>Message List</h2>
        <h:outputText value="No messages to display"
                      rendered="#{messageList.rowCount==0}"/>
        <h:dataTable var="msg" value="#{messageList}"
                     rendered="#{messageList.rowCount>0}">
          <h:column>
            <f:facet name="header">
              <h:outputText value="Read"/>
            </f:facet>
            <h:selectBooleanCheckbox value="#{msg.read}"
disabled="true"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Title"/>
            </f:facet>
            <h:commandLink value="#{msg.title}"
                           action="#{messageManager.select}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Date/Time"/>
            </f:facet>
            <h:outputText value="#{msg.datetime}">
              <f:convertDateTime type="both" dateStyle="medium"
```

```
                                    timeStyle="short"/>
            </h:outputText>
          </h:column>
          <h:column>
            <h:commandButton value="Delete"
                             action="#{messageManager.delete}"/>
          </h:column>
        </h:dataTable>
        <h3><h:outputText value="#{message.title}"/></h3>
        <div><h:outputText value="#{message.text}"/></div>
      </h:form>
    </f:view>
  </body>
</html>
```

1.3.2. 動作内容

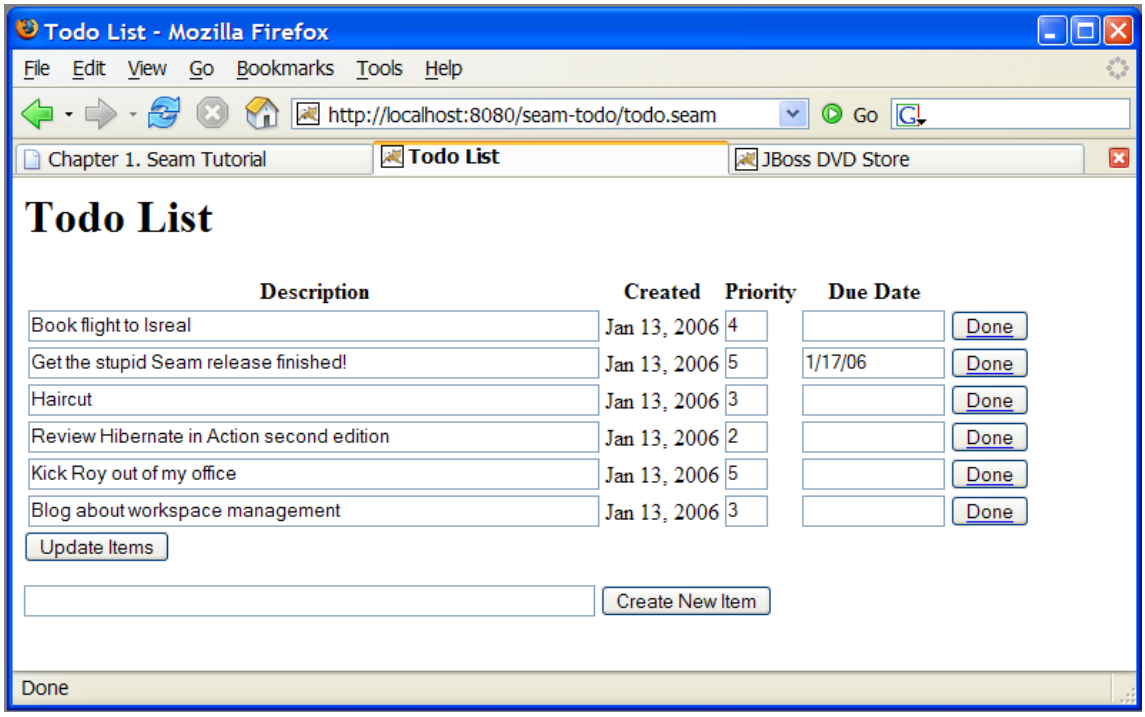
最初に **messages.jsp** ページに移動すると、 ページは **messageList** コンテキスト変数を解決しようとします。 この変数はまだ初期化されていないため、 Seam はファクトリメソッド **findMessages()** を呼び出します。 これがデータベースに問い合わせを行い **DataModel** を読み出します。 これにより **<h:dataTable>** の表示に必要な行データが提供されます。

ユーザーが **<h:commandLink>** をクリックすると、 JSF は **select()** アクションリスナーを呼び出します。 Seam はこの呼び出しをインターセプトして、選択された行データを **messageManager** コンポーネントの **message** 属性にインジェクトします。 アクションリスナーが実行され、 選択された **Message** に既読マークを付けます。 呼び出しの終わりに、 Seam は選択された **Message** を **message** コンテキスト変数にアウトジェクトします。 次に、 EJB コンテナはトランザクションをコミットし、 **Message** に対する変更がデータベースにフラッシュされます。 最後に、 このページが再度レンダリングされてメッセージ一覧を再表示、 その下に選択されたメッセージが表示されます。

ユーザーが **<h:commandButton>** をクリックすると、 JSF は **delete()** アクションリスナーを呼び出します。 Seam はこの呼び出しをインターセプトし、 選択された行データを **messageList** コンポーネントの **message** 属性にインジェクトします。 アクションリスナーが起動し、 選択された **Message** を一覧から削除、 **EntityManager** の **remove()** を呼び出します。 呼び出しの終わりに、 Seam は **messageList** コンテキスト変数を更新し、 **message** コンテキスト変数を消去します。 EJB コンテナはトランザクションをコミットし、 データベースから **Message** を削除します。 最後に、 このページが再度レンダリングされ、 メッセージ一覧を再表示します。

1.4. SEAM と JBPM : TODO 一覧サンプル

jBPM はワークフローやタスク管理に対して優れた機能を提供します。 jBPM の Seam との統合例として、 簡単な「todo 一覧」アプリケーションを見てみます。 タスク一覧の管理は jBPM の中核となる機能のため、 このサンプルには Java コードがほとんどありません。



1.4.1. コードの理解

このサンプルの中心は jBPM のプロセス定義です。 2 種類の JSP と 2種類の基本的な JavaBean も使用します。 (セッション Bean はデータベースにアクセスせず、トランザクション動作も持たないためここでは必要ありません。) プロセス定義から始めましょう。

例1.16 todo.jpdl.xml

```
<process-definition name="todo">


  <start-state name="start">
    <transition to="todo"/>
  </start-state>


  <task-node name="todo">

    <task name="todo" description="#{todoList.description}">

      <assignment actor-id="#{actor.id}"/>
    </task>
    <transition to="done"/>
  </task-node>


  <end-state name="done"/>

</process-definition>
```

1
2
3
4
5

1	**<start-state>** ノードはプロセスの論理的な開始を表します。プロセスが開始すると、直ちに **todo** ノードに遷移します。
2	**<task-node>** ノードは、**待ち状態** を表します。 ビジネスプロセスの実行が一時停止され、1 つまたは複数のタスクが行われるのを待機します。
3	**<task>** エレメントはユーザーにより実行されるタスクを定義します。このノードでは 1 つのタスクしか定義されていないため、それが完了すると実行が再開し、終了状態に遷移します。このタスクは **todoList** (JavaBeans の 1 つ) という Seam コンポーネントからその詳細を取得します。
4	タスクは生成されるとユーザーまたはユーザーグループに割り当てられる必要があります。 このサンプルでは、タスクは現在のユーザーに割り当てられ、**actor** という名前の Seam 組み込みコンポーネントから取得されます (いずれの Seam コンポーネントを使ってもタスク割り当てを実行できます)。
5	**<end-state>** ノードは、ビジネスプロセスの論理的な終了を定義します。 実行がこのノードに到達したとき、 プロセスインスタンスは破棄されます。

JBossIDE で提供されるプロセス定義エディタを使用してプロセス定義を見た場合、 以下のようになります。

このドキュメントはノードのグラフとして **ビジネスプロセス** を定義します。 これは可能な限りシンプルにしたビジネスプロセスです。 実行すべき **タスク** がひとつあり、 そのタスクが完了するとビジネスプロセスが終了します。

最初の JavaBean はログイン画面 **login.jsp** を処理します。 単に **actor** コンポーネントを使用して jBPM actor ID を初期化します。 実際のアプリケーションではユーザー認証も必要となります。

例1.17 Login.java

```
@Name("login")
public class Login {
    @In
    private Actor actor;

    private String user;

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }

    public String login() {
        actor.setId(user);
        return "/todo.jsp";
    }
}
```

ここでは、組み込み **Actor** コンポーネントをインジェクトするために**@In** を使用しているのがわかります。

次の JSP 自体は重要ではありません。

例1.18 login.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
  <head>
    <title>Login</title>
  </head>
  <body>
    <h1>Login</h1>
    <f:view>
      <h:form>
        <div>
          <h:inputText value="#{login.user}"/>
          <h:commandButton value="Login" action="#{login.login}"/>
        </div>
      </h:form>
    </f:view>
  </body>
</html>
```

2 つ目の JavaBean は、ビジネスプロセスインスタンスの開始とタスクの終了を担当します。

例1.19 TodoList.java

```
@Name("todoList")
```

```
public class TodoList
{

   private String description;

   public String getDescription()
   {
      return description;
   }

   public void setDescription(String description)
   {
      this.description = description;
   }


   @CreateProcess(definition="todo")
   public void createTodo() {}


   @StartTask @EndTask
   public void done() {}

}
```



1	description プロパティは JSP ページからユーザ入力を受け取りプロセス定義に公開して、タスクの description が設定されるようにします。
2	Seam **@CreateProcess** アノテーションは、名前付きプロセス定義のために jBPM プロセスインスタンスを生成します。
3	Seam **@StartTask** アノテーションはタスク上で作業を開始します。**@EndTask** はタスクを終了し、ビジネスプロセスの再開を可能にします。

より現実的なサンプルでは、 **@StartTask** と **@EndTask** は同じメソッドには登場しません。 タスクを完了するためにはアプリケーションを使用して何らかの作業が行われる必要があるためです。

最後に、このアプリケーションの中核となるのは **todo.jsp** です。

例1.20 todo.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://jboss.com/products/seam/taglib" prefix="s" %>
<html>
  <head>
    <title>Todo List</title>
  </head>
  <body>
    <h1>Todo List</h1>
    <f:view>
```

```
      <h:form id="list">
        <div>
          <h:outputText value="There are no todo items."
                        rendered="#{empty taskInstanceList}"/>
          <h:dataTable value="#{taskInstanceList}" var="task"
                       rendered="#{not empty taskInstanceList}">
            <h:column>
              <f:facet name="header">
                <h:outputText value="Description"/>
              </f:facet>
              <h:inputText value="#{task.description}"/>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="Created"/>
              </f:facet>
              <h:outputText value=
                           "#
{task.taskMgmtInstance.processInstance.start}">
                <f:convertDateTime type="date"/>
              </h:outputText>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="Priority"/>
              </f:facet>
              <h:inputText value="#{task.priority}" style="width: 30"/>
            </h:column>
            <h:column>
              <f:facet name="header">
                <h:outputText value="Due Date"/>
              </f:facet>
              <h:inputText value="#{task.dueDate}" style="width: 100">
                <f:convertDateTime type="date" dateStyle="short"/>
              </h:inputText>
            </h:column>
            <h:column>
              <s:button value="Done" action="#{todoList.done}"
                        taskInstance="#{task}"/>
            </h:column>
          </h:dataTable>
        </div>
        <div>
          <h:messages/>
        </div>
        <div>
          <h:commandButton value="Update Items" action="update"/>
        </div>
      </h:form>
      <h:form id="new">
        <div>
          <h:inputText value="#{todoList.description}"/>
          <h:commandButton value="Create New Item"
                           action="#{todoList.createTodo}"/>
        </div>
      </h:form>
```

```
    </f:view>
  </body>
</html>
```

簡単にするためにセクションごとに見ていきます。

ページはタスク一覧を表示しています。 **taskInstanceList** と呼ばれる Seam 組み込みコンポーネントから取得します。 この一覧は JSF フォームの中で定義されています。

例1.21 todo.jsp (taskInstanceList)

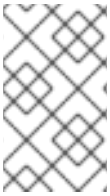
```
<h:form id="list">
  <div>
    <h:outputText value="There are no todo items."
                  rendered="#{empty taskInstanceList}"/>
    <h:dataTable value="#{taskInstanceList}" var="task"
                 rendered="#{not empty taskInstanceList}">
      ...
    </h:dataTable>
  </div>
</h:form>
```

一覧の各エレメントは jBPM クラス **TaskInstance** のインスタンスです。 以下のコードは一覧中の各タスクの特定のプロパティを表示しています。 記述内容 (Description)、 優先順 (Priority)、 納期の値 (Due Date) には入力コントロールを使用し、ユーザーはこれらの値を更新することができます。

例1.22 TaskInstance List のプロパティ

```
<h:column>
  <f:facet name="header">
    <h:outputText value="Description"/>
  </f:facet>
  <h:inputText value="#{task.description}"/>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Created"/>
  </f:facet>
  <h:outputText value="#{task.taskMgmtInstance.processInstance.start}">
    <f:convertDateTime type="date"/>
  </h:outputText>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Priority"/>
  </f:facet>
  <h:inputText value="#{task.priority}" style="width: 30"/>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Due Date"/>
  </f:facet>
```

```
    <h:inputText value="#{task.dueDate}" style="width: 100">
      <f:convertDateTime type="date" dateStyle="short"/>
    </h:inputText>
  </h:column>
```



注記

Seam では文字列を日付に変換するデフォルトの JSF 日付コンバータを提供しているので、 **#{task.dueDate}** にバインドされるフィールドにはコンバータは必要ありません。

このボタンは **@StartTask @EndTask** アノテーション付きのアクションメソッドを呼び出すことによりタスクを終了します。 タスク ID を要求パラメータとして Seam に渡します。

```
<h:column>
  <s:button value="Done" action="#{todoList.done}"
            taskInstance="#{task}"/>
</h:column>
```

seam-ui.jar パッケージから Seam **<s:button>** JSF コントロールを使用していることに留意してください。 このボタンがタスクのプロパティを更新します。 フォームがサブミットされると、 Seam と jBPM はタスクの永続に対して変更を加えます。 アクションリスナーメソッドには必要ありません。

```
  <h:commandButton value="Update Items" action="update"/>
```

ページの 2 つ目のフォームでは **@CreateProcess** アノテーション付きアクションメソッドを使って新しい項目を作成します。

```
  <h:form id="new">
    <div>
       <h:inputText value="#{todoList.description}"/>
       <h:commandButton value="Create New Item"
          action="#{todoList.createTodo}"/>
     </div>
</h:form>
```

1.4.2. 動作内容

ログイン後、 **todo.jsp** は現在のユーザーの未解決の todo 項目の表を表示するために **taskInstanceList** コンポーネントを使用します (最初は何もありません)。ページは新しいタスク項目を入力するためのフォームも表示します。 ユーザーが todo 項目を入力して **Create New Item** ボタンをクリックすると、 **#{todoList.createTodo}** が呼び出されます。これにより **todo.jpdl.xml** で定義したように todo プロセスが開始されます。

プロセスインスタンスが生成されると、直ちに **todo** 状態に遷移し、 新しいタスクが作成されます。**#{todoList.description}** に保存されるユーザーの入力に基づいてタスク description が設定されま

す。 次に、 Seam の actor コンポーネントに格納される現在のユーザーにタスクが割り当てられます。 このサンプルでは、 プロセスには追加のプロセス状態はありません。 状態はすべてタスク定義に保管されています。 プロセスとタスク情報は要求の最後でデータベースに格納されます。

todo.jsp が再表示されると、 **taskInstanceList** は新たに作成されたタスクを見つけて **h:dataTable** に表示します。 タスクの内部状態は **#{task.description}**、 **#{task.priority}**、 **#{task.dueDate}** の各列に表示されます。これらのフィールドはすべて編集してデータベースに保存することができます。

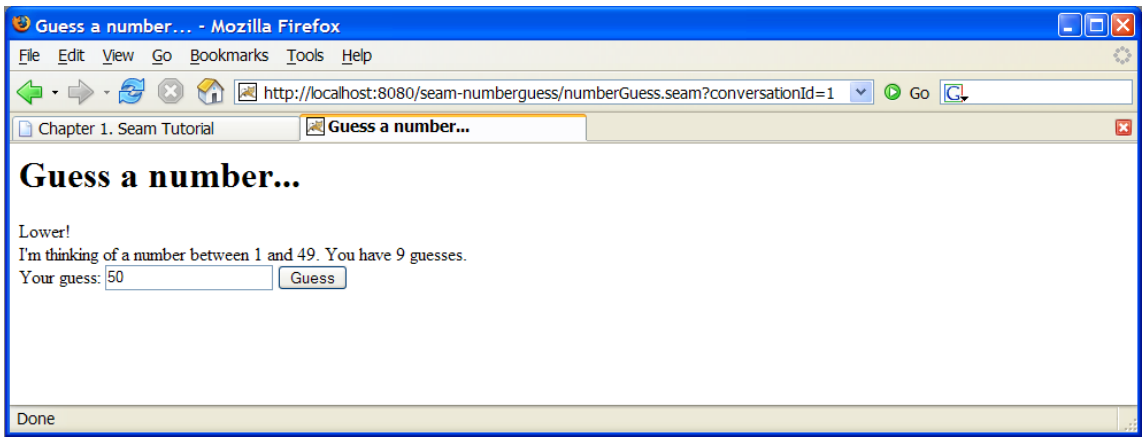
また、 各 todo 項目には **Done** ボタンがあり、 **#{todoList.done}** を呼び出します。 各ボタンは **taskInstance="#{task}"** を指定するため (表内のその特定行のタスク) **todoList** コンポーネントは完了されるタスクをはっきりと区別できます。 **@StartTask** と **@EndTask** のアノテーションがタスクをアクティブにして直ちに終了します。 次にオリジナルのプロセスが **done** 状態に遷移し (プロセス定義に従い) 終了します。 タスクとプロセスの状態がいずれもデータベース内で更新されます。

todo.jsp が再表示されると、 完了したタスクは **taskInstanceList** に表示されなくなります。このコンポーネントは未完了のタスクのみを表示するためです。

1.5. SEAM ページフロー : 数字当てゲームサンプル

自由型ナビゲーション (アドホック) 付きの Seam アプリケーションの場合、 ページフローの定義には JSF / Seam ナビゲーションルールが適しています。 ただし、 画面遷移に制約が多いスタイルのアプリケーションの場合、 特にさらにステートフルなユーザーインタフェースの場合、 ナビゲーションルールではシステムの流れを本当に理解するのは困難です。 ビューページ、 アクション、 ナビゲーションルールからの情報を組み合わせて考えると、このフローが理解し易くなります。

Seam では次に示す数字当てゲームのサンプルに見られるように、ページフローの定義に jPDL プロセス定義を使用することができます。



1.5.1. コードの理解

このサンプルは 1 つの JavaBean、3 つの JSP ページ、それと 1 つの jPDL プロセスフロー定義を使用します。 ページフローから見ていきましょう。

例1.23 pageflow.jpdl.xml

```
<pageflow-definition
    xmlns="http://jboss.com/products/seam/pageflow"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.com/products/seam/pageflow
                        http://jboss.com/products/seam/pageflow-
2.2.xsd"
```

```
    name="numberGuess">

  <start-page name="displayGuess" view-id="/numberGuess.jspx">
```

```
    <redirect/>
    <transition name="guess" to="evaluateGuess">
```

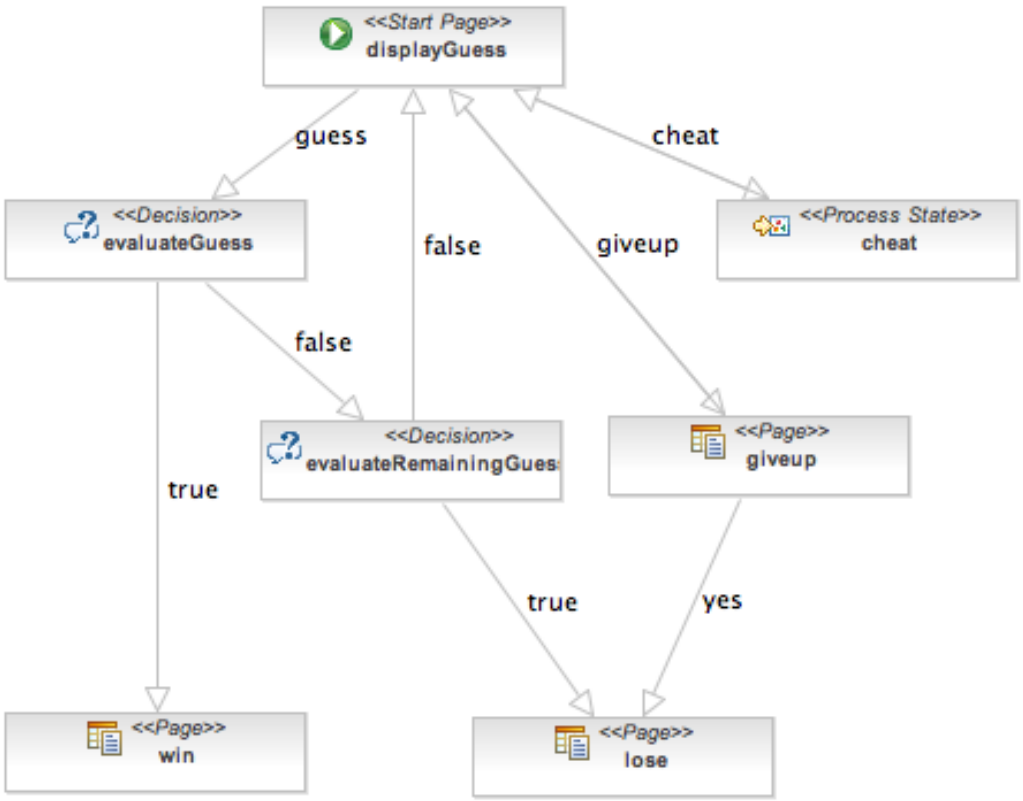
```
      <action expression="#{numberGuess.guess}"/>
```

```
    </transition>
    <transition name="giveup" to="giveup"/>
    <transition name="cheat" to="cheat"/>
  </start-page>

  <decision name="evaluateGuess" expression="#
```

```
{numberGuess.correctGuess}">
    <transition name="true" to="win"/>
    <transition name="false" to="evaluateRemainingGuesses"/>
  </decision>

  <decision name="evaluateRemainingGuesses" expression="#
{numberGuess.lastGuess}">
    <transition name="true" to="lose"/>
    <transition name="false" to="displayGuess"/>
  </decision>

  <page name="giveup" view-id="/giveup.jspx">
    <redirect/>
    <transition name="yes" to="lose"/>
    <transition name="no" to="displayGuess"/>
  </page>

  <process-state name="cheat">
    <sub-process name="cheat"/>
    <transition to="displayGuess"/>
  </process-state>

  <page name="win" view-id="/win.jspx">
    <redirect/>
    <end-conversation/>
  </page>

  <page name="lose" view-id="/lose.jspx">
    <redirect/>
    <end-conversation/>
  </page>

</pageflow-definition>
```

❶	**<page>** エレメントは待ち状態を定義し、そこではシステムは特定の JSF ビューを表示し、ユーザー入力を待ちます。**view-id** は純粋な JSF ナビゲーションルールで使用されているのと同じ JSF ビュー ID です。ページに移動すると **redirect** 属性は Seam に post-then-redirect を使用するよう指示します (これにより使い易いブラウザ URL となります)。
❷	**<transition>** エレメントは JSF 結果に名前を付けます。JSF アクションがその結果になると遷移が引き起こされます。jBPM 遷移アクションが呼び出された後、実行はページフローグラフの次のノードに進みます。
❸	遷移の **<action>** は jBPM の遷移が起こるときに発生するという点以外は、JSF アクションのようなものです。遷移アクションはどの Seam コンポーネントでも呼び出すことが可能です。
❹	**<decision>** ノードはページフローを分岐させ、 JSF EL 式を評価することによって次に実行するノードを決定します。

JBoss Developer Studio のページフローエディタではページフローは次のようになります。



このページフローを覚えておくと残りのアプリケーション部分を理解するのがとても簡単になります。

これがアプリケーションの中心のページ **numberGuess.jspx** です。

例1.24 numberGuess.jspx

```
<?xml version="1.0"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:h="http://java.sun.com/jsf/html"
```

```
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:s="http://jboss.com/products/seam/taglib"
          xmlns="http://www.w3.org/1999/xhtml"
          version="2.0">
  <jsp:output doctype-root-element="html"
              doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
              doctype-system=
              "http://www.w3c.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd"/>
  <jsp:directive.page contentType="text/html"/>
  <html>
    <head>
      <title>Guess a number...</title>
      <link href="niceforms.css" rel="stylesheet" type="text/css" />
      <script language="javascript" type="text/javascript"
              src="niceforms.js" />
    </head>
    <body>
      <h1>Guess a number...</h1>
      <f:view>
        <h:form styleClass="niceform">

          <div>
            <h:messages globalOnly="true"/>
            <h:outputText value="Higher!"
                          rendered="#{
                                    numberGuess.randomNumber gt
                                    numberGuess.currentGuess}"/>
            <h:outputText value="Lower!"
                          rendered="#{
                                    numberGuess.randomNumber lt
                                    numberGuess.currentGuess}"/>
          </div>

          <div>
            I'm thinking of a number between
            <h:outputText value="#{numberGuess.smallest}"/> and
            <h:outputText value="#{numberGuess.biggest}"/>. You have
            <h:outputText value="#{numberGuess.remainingGuesses}"/>
            guesses.
          </div>

          <div>
            Your guess:
            <h:inputText value="#{numberGuess.currentGuess}"
                         id="inputGuess" required="true" size="3"
                         rendered="#{
                                    (numberGuess.biggest-
numberGuess.smallest) gt
                                    20}">
              <f:validateLongRange maximum="#{numberGuess.biggest}"
                                   minimum="#{numberGuess.smallest}"/>
            </h:inputText>
            <h:selectOneMenu value="#{numberGuess.currentGuess}"
                             id="selectGuessMenu" required="true"
                             rendered="#{
```

```
                                        (numberGuess.biggest-
numberGuess.smallest) le
                                        20 and
                                        (numberGuess.biggest-
numberGuess.smallest) gt
                                        4}">
              <s:selectItems value="#{numberGuess.possibilities}"
                             var="i" label="#{i}"/>
            </h:selectOneMenu>
            <h:selectOneRadio value="#{numberGuess.currentGuess}"
                              id="selectGuessRadio"
                              required="true"
                              rendered="#{
                                        (numberGuess.biggest-
numberGuess.smallest) le
                                        4}">
              <s:selectItems value="#{numberGuess.possibilities}"
                             var="i" label="#{i}"/>
            </h:selectOneRadio>
            <h:commandButton value="Guess" action="guess"/>
            <s:button value="Cheat" view="/confirm.jspx"/>
            <s:button value="Give up" action="giveup"/>
          </div>

          <div>
            <h:message for="inputGuess" style="color: red"/>
          </div>

        </h:form>
      </f:view>
    </body>
  </html>
</jsp:root>
```

アクションを直接呼び出す代わりに、コマンドボタンが **guess** 遷移の名前付けを行っていることに注意してください。

win.jspx ページはごく普通のものです。

例1.25 win.jspx

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns="http://www.w3.org/1999/xhtml"
          version="2.0">
  <jsp:output doctype-root-element="html"
              doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
              doctype-system="http://www.w3c.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd"/>
  <jsp:directive.page contentType="text/html"/>
  <html>
    <head>
      <title>You won!</title>
```

```
      <link href="niceforms.css" rel="stylesheet" type="text/css" />
    </head>
    <body>
      <h1>You won!</h1>
      <f:view>
        Yes, the answer was
        <h:outputText value="#{numberGuess.currentGuess}" />.
        It took you
        <h:outputText value="#{numberGuess.guessCount}" /> guesses.
        <h:outputText value="But you cheated, so it doesn't count!"
                      rendered="#{numberGuess.cheat}"/>
        Would you like to <a href="numberGuess.seam">play again</a>?
      </f:view>
    </body>
  </html>
</jsp:root>
```

lose.jspx はほぼ同じですのでここでは記載しません。

最後に、 実際のアプリケーションコードを見ましょう。

例1.26 NumberGuess.java

```
@Name("numberGuess")
@Scope(ScopeType.CONVERSATION)
public class NumberGuess implements Serializable {

   private int randomNumber;
   private Integer currentGuess;
   private int biggest;
   private int smallest;
   private int guessCount;
   private int maxGuesses;
   private boolean cheated;



   @Create
   public void begin()
   {
      randomNumber = new Random().nextInt(100);
      guessCount = 0;
      biggest = 100;
      smallest = 1;
   }

   public void setCurrentGuess(Integer guess)
   {
      this.currentGuess = guess;
   }

   public Integer getCurrentGuess()
   {
      return currentGuess;
   }
```

1

```
public void guess()
{
   if (currentGuess>randomNumber)
   {
      biggest = currentGuess - 1;
   }
   if (currentGuess<randomNumber)
   {
      smallest = currentGuess + 1;
   }
   guessCount ++;
}

public boolean isCorrectGuess()
{
   return currentGuess==randomNumber;
}

public int getBiggest()
{
   return biggest;
}

public int getSmallest()
{
   return smallest;
}

public int getGuessCount()
{
   return guessCount;
}

public boolean isLastGuess()
{
   return guessCount==maxGuesses;
}

public int getRemainingGuesses() {
   return maxGuesses-guessCount;
}

public void setMaxGuesses(int maxGuesses) {
   this.maxGuesses = maxGuesses;
}

public int getMaxGuesses() {
   return maxGuesses;
}

public int getRandomNumber() {
   return randomNumber;
}

public void cheated()
```

```
    {
       cheated = true;
    }

    public boolean isCheat() {
       return cheated;
    }

    public List<Integer> getPossibilities()
    {
       List<Integer> result = new ArrayList<Integer>();
       for(int i=smallest; i<=biggest; i++) result.add(i);
       return result;
    }

}
```

1	初めて JSP ページが **numberGuess** コンポーネントを求めると、Seam によりそのページに対し新しいコンポーネントが作成され、**@Create** メソッドが呼び出され、コンポーネントがそれ自体を初期化することができます。

pages.xml ファイルにより Seam の **対話** が開始し、対話のページフローに使用するページフロー定義を指定します。詳細は 8章*対話とワークスペースの管理*を参照してください。

例1.27 pages.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<pages xmlns="http://jboss.com/products/seam/pages"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jboss.com/products/seam/pages
http://jboss.com/products/seam/pages-2.2.xsd">
  <page view-id="/numberGuess.jspx">
    <begin-conversation join="true" pageflow="numberGuess"/>
  </page>
</pages>
```

このコンポーネントは純粋なビジネスロジックです。 ユーザーインタラクションのフローについての情報は必要としないため、 再利用できる可能性が高くなります。

1.5.2. 動作内容

ゲームは **numberGuess.jspx** から始まります。 ページが初めて表示されると、**pages.xml** 設定は対話をアクティブにし **numberGuess** ページフローをその対話と関連付けます。 ページフローは **start-page** タグから開始されるので **(待機** 状態) **numberGuess.xhtml** が表示されます。

ビューは **numberGuess** コンポーネントを参照し、 これにより新しいインスタンスが生成され対話に保管されます。 **@Create** メソッドが呼び出され、 ゲームの状態が初期化されます。 ビューは **h:form** を表示し、ユーザーは **#{numberGuess.currentGuess}** を編集できます。

「Guess」ボタンは **guess** アクションを引き起こします。 Seam はアクションを処理するためにページフローを参照し、 このページフローが **evaluateGuess** を呼び出し (推測した数字と **numberGuess** コンポーネント内の最小と最大の候補を更新)、 **evaluateGuess** 状態に遷移します。

evaluateGuess 状態は **#{numberGuess.correctGuess}** の値をチェックし、 **win** または **evaluatingRemainingGuesses** 状態のいずれかに遷移します。 数字が間違っていたとすると、ページフローは **evaluatingRemainingGuesses** に遷移します。 これは決定する状態でもあり、 ユーザーがまだ数字当てを続行できるか否かを決定する **#{numberGuess.lastGuess}** 状態をテストします。 続行できる場合は (**lastGuess** が **false**) 最初の **displayGuess** 状態に戻ります。 これはページの状態となるため、関連ページの **/numberGuess.jspx** が表示されます。 また、 このページにはリダイレクトエレメントが含まれるため、 Seam はリダイレクトをユーザーのブラウザに送信し、 それがプロセスを再び開始します。

以降の要求により **win** または **lose** の遷移のいずれかが呼び出されると、 ユーザーは**/win.jspx** または **/lose.jspx** にそれぞれ移動されます。 いずれの状態も Seam が対話を終了、 ゲームとページフロー状態の保持を中止、 ユーザーを最終ページへとリダイレクトすることを指定します。

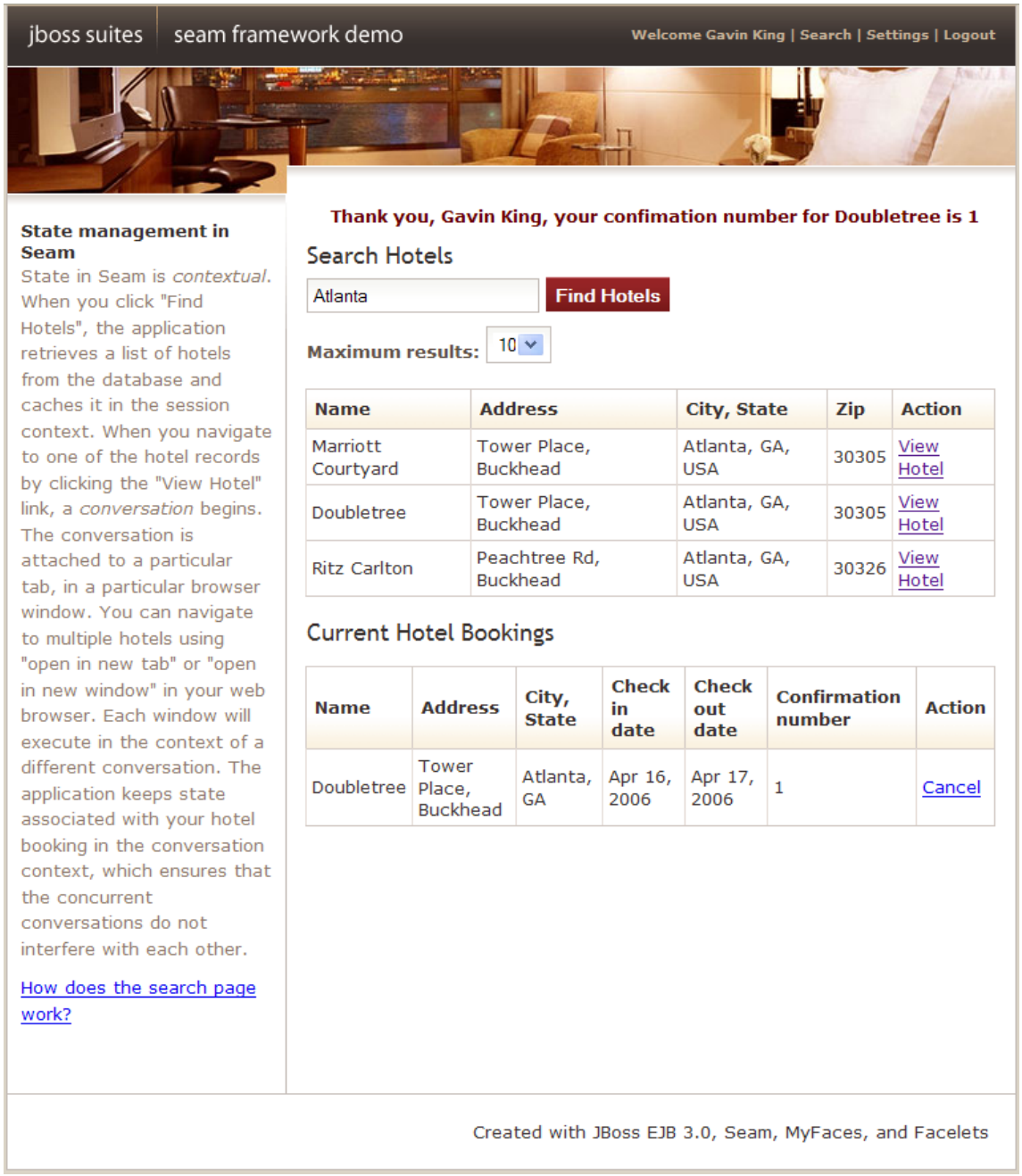
また、 数字当てゲームのサンプルには **Give up** と **Cheat** のボタンもあります。 両方の動作のページフロー状態の追跡は比較的簡単ですので、 ここでは説明しません。 **cheat** トランザクションに注目してください。これはサブプロセスを読み込みその特定の流れを処理します。 このアプリケーションではこのプロセスは余分ですが、 理解し易くするために複雑なページフローを細分化してより簡単な構造にする方法を示しています。

1.6. SEAM アプリケーションの全容 : ホテル予約サンプル

1.6.1. はじめに

予約アプリケーションにはホテルの部屋予約システムが完備されており、以下の機能が含まれています。

- ユーザー登録
- ログイン
- ログアウト
- パスワード設定
- ホテル検索
- ホテル選択
- 部屋予約
- 予約確認
- 現在の予約一覧



この予約アプリケーションは JSF、EJB 3.0、Seam とともにビューとして Facelet を使用しています。 JSF、Facelets、Seam、JavaBeans そして、Hibernate3 のアプリケーションの移植版もあります

このアプリケーションに関して気付かれることのひとつとして、 アプリケーションが極めて **堅牢** である点です。 複数のウィンドウを開いたり、 戻るボタンやブラウザ更新のボタンを押したり、 また適当なデータを入力してもアプリケーションはなかなかクラッシュしません。 Seam は堅牢な Web アプリケーションを容易に作成できるよう設計されているため、これまで手でコード化したことで得られた堅牢性は Seam を使用することで自動的に、かつ自然と得ることができます。

サンプルアプリケーションのソースコードを見れば、 どのようにアプリケーションが動作するか習得できます。 この堅牢性を実現するため、 どのように宣言的状態管理や統合されたデータ妥当性検証が使用されているかに注目してください。

1.6.2. 予約サンプルの概要

プロジェクトの構成は以前のプロジェクトとまったく同じです。 このアプリケーションをインストール、デプロイするには、 「Seam サンプルの使用」 を参照してください。 アプリケーションの起動に成功したら、 ブラウザで `http://localhost:8080/seam-booking/` をポイントするとアクセスで

きます。

このアプリケーションは 6 つのセッション Bean を使って以下の機能に対応するビジネスロジックを実装します。

- **AuthenticatorAction** はログイン認証ロジックを提供します。
- **BookingListAction** は現在ログインしているユーザーのために現状の予約を取得します。
- **ChangePasswordAction** は現在ログインしているユーザーのパスワードを更新します。
- **HotelBookingAction** は予約と確認の機能を実装します。**対話** として実装されるため、このアプリケーションの中で最も重要なクラスのひとつになります。
- **HotelSearchingAction** はホテル検索の機能を実装しています。
- **RegisterAction** は新しいシステムユーザーを登録します。

3 つのエンティティ Bean はアプリケーションの永続ドメインモデルを実装しています。

- **Hotel** はホテルを表すエンティティ Bean です。
- **Booking** は現在の予約を表すエンティティ Bean です。
- **User** はホテル予約ができるユーザーを表すエンティティ Bean です。

1.6.3. Seam 対話の理解

このチュートリアルでは、ホテル予約をするという特定の機能に焦点を絞って説明します。ユーザーの視点から見ると、ホテルの検索、選択、予約そして確認までは 1 つの連続した作業単位、つまり **対話** です。しかし、開発者側から見ると、検索は独立していることが重要で、これによりユーザーが同じ検索結果ページから複数のホテルを選択し、別々のブラウザタブにそれぞれ異なる対話を開くことができます。

ほとんどの Web アプリケーションのアーキテクチャは、対話を表すための優れた構造を持っていません。これは対話の状態を管理するために重大な問題となります。通常、Java Web アプリケーションはいくつかの技術を組み合わせて使用します。ある状態は URL に変換されますが、ここで変換できない状態は各要求の前後でデータベースに記録されるか、**HttpSession** に追加されます。

データベースは最もスケーラビリティに乏しい層なので、これが極端にスケーラビリティを低減させています。データベースを行き来する転送量の増加によっても待ち時間が増加します。冗長な転送量を減少させるために、Java アプリケーションでは要求間でよくアクセスされるデータを保管するデータキャッシュを導入することがよくあります。しかし、データが無効かどうかの判断はユーザーがデータの操作を終了したかどうかではなく LRU ポリシーを基にして行うため、このキャッシュは効率的ではありません。また、キャッシュは同時トランザクション間で共有されるので、キャッシュされた状態がデータベースの状態と一貫性を保持することに関連するさらなる問題も取り入れてしまうことになります。

HttpSession に保管された状態にも同様の問題が見られます。**HttpSession** は実際のセッションデータ (ユーザーとアプリケーション間の全要求に共通となるデータ) の保存については問題ありませんが、個別要求のシリーズに関連するデータの場合は問題となります。ここで保存される対話は複数のウィンドウや戻るボタンをクリックした場合にすぐに分解してしまいます。プログラミングに注意しないと **HttpSession** にあるデータも増大し、セッションのクラスタ化を困難にします。こうした手法から生じる問題に対応するためのメカニズムを開発するのは簡単ではありません (異なる同時対話に関連するセッション状態を分離して、ひとつの対話が中断したら対話状態が必ず破棄されるようフェイルセーフを組み込みます)。

Seam は優れた構造として **対話コンテキスト** を導入することにより状況を大幅に改善します。対話状態はこのコンテキストで安全に保管され、 明確に定義されたライフサイクルを持ちます。さらに良いことに、 対話コンテキストはユーザーが現在作業しているデータの自然なキャッシュとなるため、 アプリケーションサーバーとデータベース間でデータを継続的にプッシュする必要がありません。

次のアプリケーションではステートフルセッション Bean の保存に対話コンテキストが使用されています。 これらはスケーラビリティという観点からは弊害をもたらすとみなされることがあり、 また過去にはそうだったかもしれません。 ただし、 最近のアプリケーションサーバーはステートフルセッション Bean の複製に関して高度に優れたメカニズムを備えています。 JBoss AS は微細な複製機能で、変更された Bean の属性値のみを複製することが可能です。 ステートフルセッション Bean を正しく使用すればスケーラビリティに関する問題を引き起こすことはありません。 ただし、 ステートフルセッション Bean に不慣れな場合や使用したくない場合は POJO を使用することもできます。

この予約サンプルでは、 複雑な動作を実現するために異なるスコープを持つステートフルコンポーネントを連携させることができる一例を示しています。 予約アプリケーションのメインページではユーザーによるホテル検索が可能です。 検索結果は Seam セッションスコープに保管されます。 ユーザーがこれらのホテルの 1 つに移動すると、 対話が開始され、 対話スコープのコンポーネントがセッションスコープのコンポーネントから選択したホテルを読み出します。

予約サンプルは、 手書きの JavaScript を使用することなくリッチクライアントの動作を実装する場合の RichFaces Ajax の使い方も示しています。

検索機能はセッションスコープのステートフルセッション Bean を使用して実装されます。 メッセージ一覧サンプルに使用されているものと同様です。

例1.28 HotelSearchingAction.java

```
@Stateful
(1)
@Name("hotelSearch")
@Scope(ScopeType.SESSION)
@Restrict("#{identity.loggedIn}")
(2)
public class HotelSearchingAction implements HotelSearching
{

    @PersistenceContext
    private EntityManager em;

    private String searchString;
    private int pageSize = 10;
    private int page;

    @DataModel
(3)

    private List<Hotel> hotels;

    public void find()
    {
        page = 0;
        queryHotels();
```

```
    }
    public void nextPage()
    {
        page++;
        queryHotels();
    }

    private void queryHotels()
    {
        hotels =
            em.createQuery("select h from Hotel h where lower(h.name)
like #{pattern} " +
                           "or lower(h.city) like #{pattern} " +
                           "or lower(h.zip) like #{pattern} " +
                           "or lower(h.address) like #{pattern}")
              .setMaxResults(pageSize)
              .setFirstResult( page * pageSize )
              .getResultList();
    }

    public boolean isNextPageAvailable()
    {
        return hotels!=null && hotels.size()==pageSize;
    }

    public int getPageSize() {
        return pageSize;
    }

    public void setPageSize(int pageSize) {
        this.pageSize = pageSize;
    }

    @Factory(value="pattern", scope=ScopeType.EVENT)
    public String getSearchPattern()
    {
        return searchString==null ?
            "%" : '%' + searchString.toLowerCase().replace('*', '%') +
'%';
    }

    public String getSearchString()
    {
        return searchString;
    }

    public void setSearchString(String searchString)
    {
        this.searchString = searchString;
    }

    @Remove
    4

    public void destroy() {}
}
```

1	EJB 標準 **@Stateful** アノテーションは、 このクラスがステートフルセッション Bean であることを識別しています。 ステートフルセッション Bean は、 デフォルトで対話コンテキストのスコープを持ちます。
2	**@Restrict** アノテーションはコンポーネントにセキュリティ制限を適用します。 コンポーネントへのアクセスを制限し、 ログインしているユーザーにのみアクセスを許可します。「セキュリティ」の章では Seam におけるセキュリティについてさらに詳細に説明します
3	**@DataModel** アノテーションは JSF**ListDataModel** として **List** を公開します。これにより検索画面でのクリック可能な一覧の実装が容易になります。このサンプルでは、ホテル一覧が **hotels** という名前の対話変数で**ListDataModel** としてページに公開されています。
4	EJB 標準の **@Remove** アノテーションはアノテーションが付けられたメソッドが呼び出された後ステートフルセッション Bean が取り除かれてその状態が破棄されることを規定しています。 Seam では、 すべてのステートフルセッション Bean はパラメータなしの **@Remove** メソッドを定義しなければなりません。Seam がセッションコンテキストを破棄するとこのメソッドが呼び出されます。

アプリケーションのメインページは Facelets ページです。 ホテル検索に関連する部分を見てみましょう。

例1.29 main.xhtml

```
<div class="section">

  <span class="errors">
    <h:messages globalOnly="true"/>
  </span>

  <h1>Search Hotels</h1>

  <h:form id="searchCriteria">
    <fieldset>
      <h:inputText id="searchString" value="#
{hotelSearch.searchString}"
                   style="width: 165px;">
        <a:support event="onkeyup" actionListener="#{hotelSearch.find}"
```

1

```
                   reRender="searchResults" />
      </h:inputText>
       
      <a:commandButton id="findHotels" value="Find Hotels" action="#
{hotelSearch.find}"
                       reRender="searchResults"/>
       
      <a:status>
```

2

```
        <f:facet name="start">
          <h:graphicImage value="/img/spinner.gif"/>
        </f:facet>
```

```
      </a:status>
      <br/>
      <h:outputLabel for="pageSize">Maximum
results:</h:outputLabel> 
      <h:selectOneMenu value="#{hotelSearch.pageSize}" id="pageSize">
        <f:selectItem itemLabel="5" itemValue="5"/>
        <f:selectItem itemLabel="10" itemValue="10"/>
        <f:selectItem itemLabel="20" itemValue="20"/>
      </h:selectOneMenu>
    </fieldset>
  </h:form>

</div>

<a:outputPanel id="searchResults">
```

3

```
  <div class="section">
    <h:outputText value="No Hotels Found"
                  rendered="#{hotels != null and hotels.rowCount==0}"/>
    <h:dataTable id="hotels" value="#{hotels}" var="hot"
                 rendered="#{hotels.rowCount>0}">
      <h:column>
        <f:facet name="header">Name</f:facet>
        #{hot.name}
      </h:column>
      <h:column>
        <f:facet name="header">Address</f:facet>
        #{hot.address}
      </h:column>
      <h:column>
        <f:facet name="header">City, State</f:facet>
        #{hot.city}, #{hot.state}, #{hot.country}
      </h:column>
      <h:column>
        <f:facet name="header">Zip</f:facet>
        #{hot.zip}
      </h:column>
      <h:column>
        <f:facet name="header">Action</f:facet>
        <s:link id="viewHotel" value="View Hotel"
```

4

```
                action="#{hotelBooking.selectHotel(hot)}"/>
      </h:column>
    </h:dataTable>
    <s:link value="More results" action="#{hotelSearch.nextPage}"
            rendered="#{hotelSearch.nextPageAvailable}"/>
  </div>
</a:outputPanel>
```

1	RichFaces Ajax **<a:support>** タグを使用すると、**onkeyup** のような JavaScript イベントの発生時に、非同期の **XMLHttpRequest** により JSF アクションイベントリスナーが呼び出されます。さらに良いことには **reRender** 属性により非同期の応答を受け取ると JSF ページの一部を表示し、一部のページを更新することが可能です。
2	RichFaces Ajax **<a:status>** タグを使用すると、非同期の要求が返されるのを待つ間に動画イメージを表示させます。
3	RichFaces Ajax **<a:outputPanel>** タグは非同期要求によって再レンダリング可能なページの領域を定義します。
4	Seam **<s:link>** タグを使用すると、JSF アクションリスナーを普通の (非 JavaScript) HTML リンクにつなげることができます。標準 JSF **<h:commandLink>** と比べて有利な点は、「新しいウィンドウで開く」や「新しいタブで開く」という動作を維持することです。パラメータ **#{hotelBooking.selectHotel(hot)}** 付きのメソッドバインディングを使用している点に注目してください。これは標準 Unified EL では不可能ですが、Seam はすべてのメソッドバインディング式でパラメータを使用できるよう EL を拡張します。 ナビゲーションのルールは **WEB-INF/pages.xml** に記載されています。詳細は「ナビゲーション」で説明します。

このページはユーザーの入力に応じて検索結果を動的に表示して、選択したホテルを **HotelBookingAction** の **selectHotel()** メソッドに渡します。 ここで実際の作業が発生します。

次のコードでは予約サンプルアプリケーションがどのように対話スコープのステートフルセッション Bean を使用し、 対話関連の永続データの自然なキャッシュを実現しているのかを示しています。コードを対話の各種ステップを実装するスクリプト化された動作の一覧と考えると理解できます。

例1.30 HotelBookingAction.java

```
@Stateful
@Name("hotelBooking")
@Restrict("#{identity.loggedIn}")
public class HotelBookingAction implements HotelBooking
{

    @PersistenceContext(type=EXTENDED)
(1)

    private EntityManager em;

    @In
        private User user;

    @In(required=false) @Out
    private Hotel hotel;

    @In(required=false)
    @Out(required=false)
(2)

    private Booking booking;
```

```
    @In
    private FacesMessages facesMessages;

    @In
    private Events events;

    @Logger
        private Log log;

    private boolean bookingValid;

    @Begin
```

3

```
    public void selectHotel(Hotel selectedHotel)
    {
        hotel = em.merge(selectedHotel);
    }

    public void bookHotel()
    {
        booking = new Booking(hotel, user);
        Calendar calendar = Calendar.getInstance();
        booking.setCheckinDate( calendar.getTime() );
        calendar.add(Calendar.DAY_OF_MONTH, 1);
        booking.setCheckoutDate( calendar.getTime() );
    }

    public void setBookingDetails()
    {
        Calendar calendar = Calendar.getInstance();
        calendar.add(Calendar.DAY_OF_MONTH, -1);
        if ( booking.getCheckinDate().before( calendar.getTime() ) )
            {
                facesMessages.addToControl("checkinDate",
                                           "Check in date must be a
future date");
                bookingValid=false;
            }
        else if ( !booking.getCheckinDate().before(
booking.getCheckoutDate() ) )
            {
                facesMessages.addToControl("checkoutDate",
                                           "Check out date must be
later " +
                                           "than check in date");
                bookingValid=false;
            }
        else
            {
                bookingValid=true;
            }
    }

    public boolean isBookingValid()
```

```
    {
        return bookingValid;
    }

    @End
```

```
    public void confirm()
    {
        em.persist(booking);
        facesMessages.add("Thank you, #{user.name}, your confimation
number " +
                          " for #{hotel.name} is #{booki g.id}");
        log.info("New booking: #{booking.id} for #{user.username}");
        events.raiseTransactionSuccessEvent("bookingConfirmed");
    }

    @End
    public void cancel() {}

    @Remove
```

```
    public void destroy() {}
}
```

❶	この Bean は EJB3 **拡張永続コンテキスト** を使用するため、 エンティティインスタンスはステートフルセッション Bean のライフサイクル全体に対して管理されたままとなります。
❷	**@Out** アノテーションはメソッド呼び出しの後に属性値がコンテキスト変数に**アウトジェクト** されることを宣言します。このサンプルでは、アクションリスナーの呼び出しが完了するごとに、**hotel** という名前のコンテキスト変数が**hotel** インスタンス変数の値に設定されます。
❸	**@Begin** アノテーションは、 アノテーション付きメソッドが**長期実行の対話** を開始することを指定するため、現在の対話コンテキストは要求の終わりに破棄されません。その代わりに、現在のウィンドウからのあらゆる要求に再度関連付けられ、非アクティブな対話によるタイムアウトまたは適合する **@End** メソッドにより破棄されます。
❹	**@End** アノテーションはアノテーション付きメソッドが現在の長期実行の対話を終了することを指定します。したがって要求の終わりで現在の対話コンテキストは破棄されます。
❺	この EJB remove メソッドは Seam が対話コンテキストを破棄すると呼び出されます。このメソッドを定義するのを忘れないようにしてください。

HotelBookingAction は選択、 予約、 予約確認を実装するすべてのアクションリスナーのメソッドを持っており、 この操作に関連する状態をそのインスタンス変数に保持しています。 このコードは **HttpSession** 属性の取得と設定に比べるとより明確でかつシンプルです。

さらに良いことに、 ユーザーはログインセッション毎に複数の分離された対話を持つことが可能です。 ログインして検索を試行したり、 複数のブラウザタブを開いて異なるホテルのページを表示させたりしてみてください。 同時に 2 つの異なるホテル予約の作成を行うことが可能です。 いずれかの対

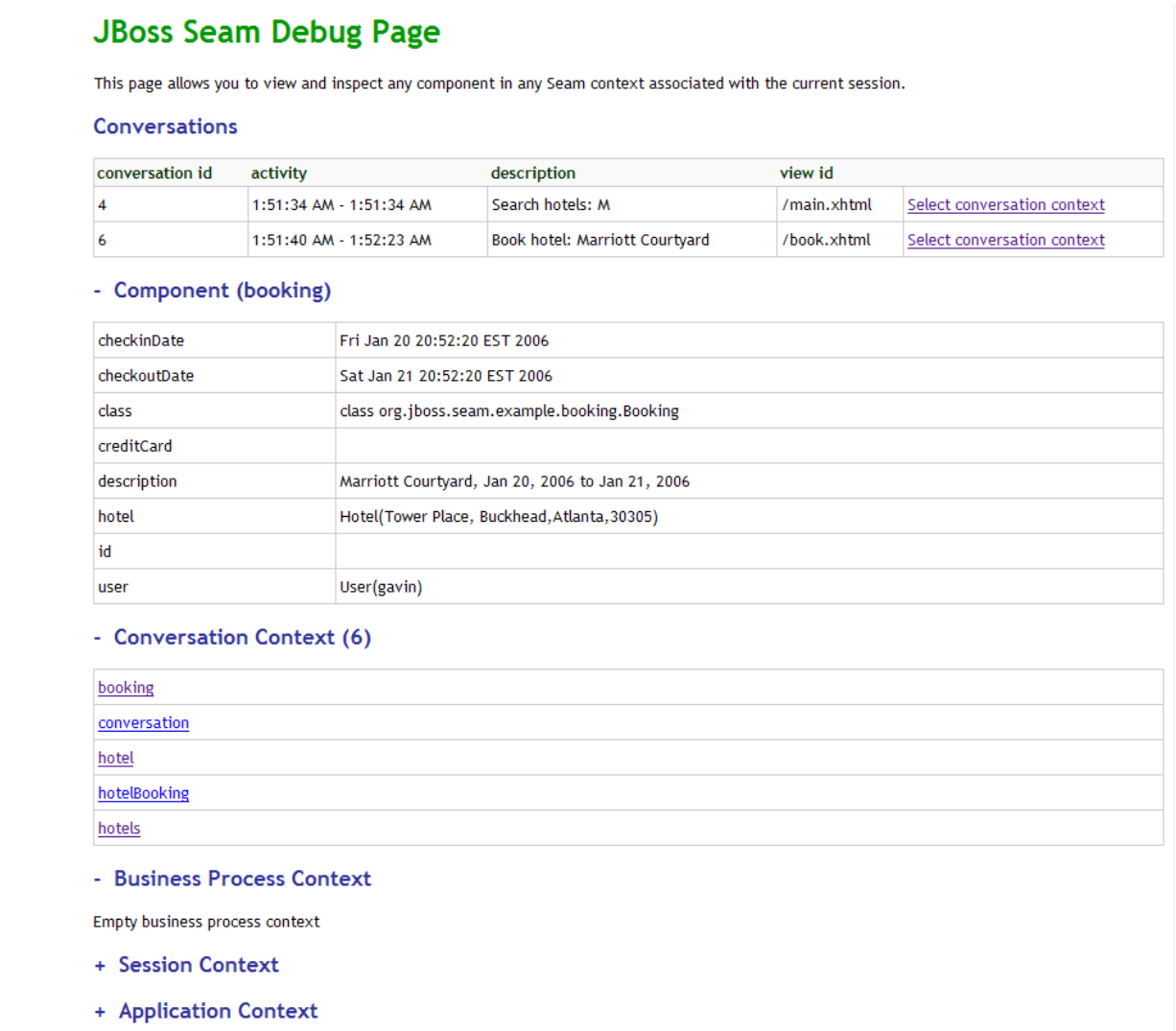
話を長時間放置すると Seam は最終的にはその対話をタイムアウトし状態を破棄します。 対話の終了後に、 その対話ページに戻るボタンを押して戻り何らかの操作を行おうとすると、 Seam によって対話が既に終了したことが検出され、検索ページにリダイレクトされます。

1.6.4. Seam デバッグページ

WAR は **seam-debug.jar** も含みます。 Seam デバッグページの使用を有効にするには、 Facelets と一緒に **WEB-INF/lib** にこの jar をデプロイし、 **init** コンポーネントの debug プロパティを以下のように設定します。

```
<core:init jndi-pattern="@jndiPattern@" debug="true"/>
```

デバッグページでは、 現在のログインセッションに関連するすべての Seam コンテキスト中の Seam コンポーネントを閲覧、検査することができます。 ブラウザで **http://localhost:8080/seam-booking/debug.seam** をポイントするだけです。



JBoss Seam Debug Page

This page allows you to view and inspect any component in any Seam context associated with the current session.

Conversations

conversation id	activity	description	view id	
4	1:51:34 AM - 1:51:34 AM	Search hotels: M	/main.xhtml	Select conversation context
6	1:51:40 AM - 1:52:23 AM	Book hotel: Marriott Courtyard	/book.xhtml	Select conversation context

- Component (booking)

checkinDate	Fri Jan 20 20:52:20 EST 2006
checkoutDate	Sat Jan 21 20:52:20 EST 2006
class	class org.jboss.seam.example.booking.Booking
creditCard	
description	Marriott Courtyard, Jan 20, 2006 to Jan 21, 2006
hotel	Hotel(Tower Place, Buckhead,Atlanta,30305)
id	
user	User(gavin)

- Conversation Context (6)

booking
conversation
hotel
hotelBooking
hotels

- Business Process Context

Empty business process context

+ Session Context

+ Application Context

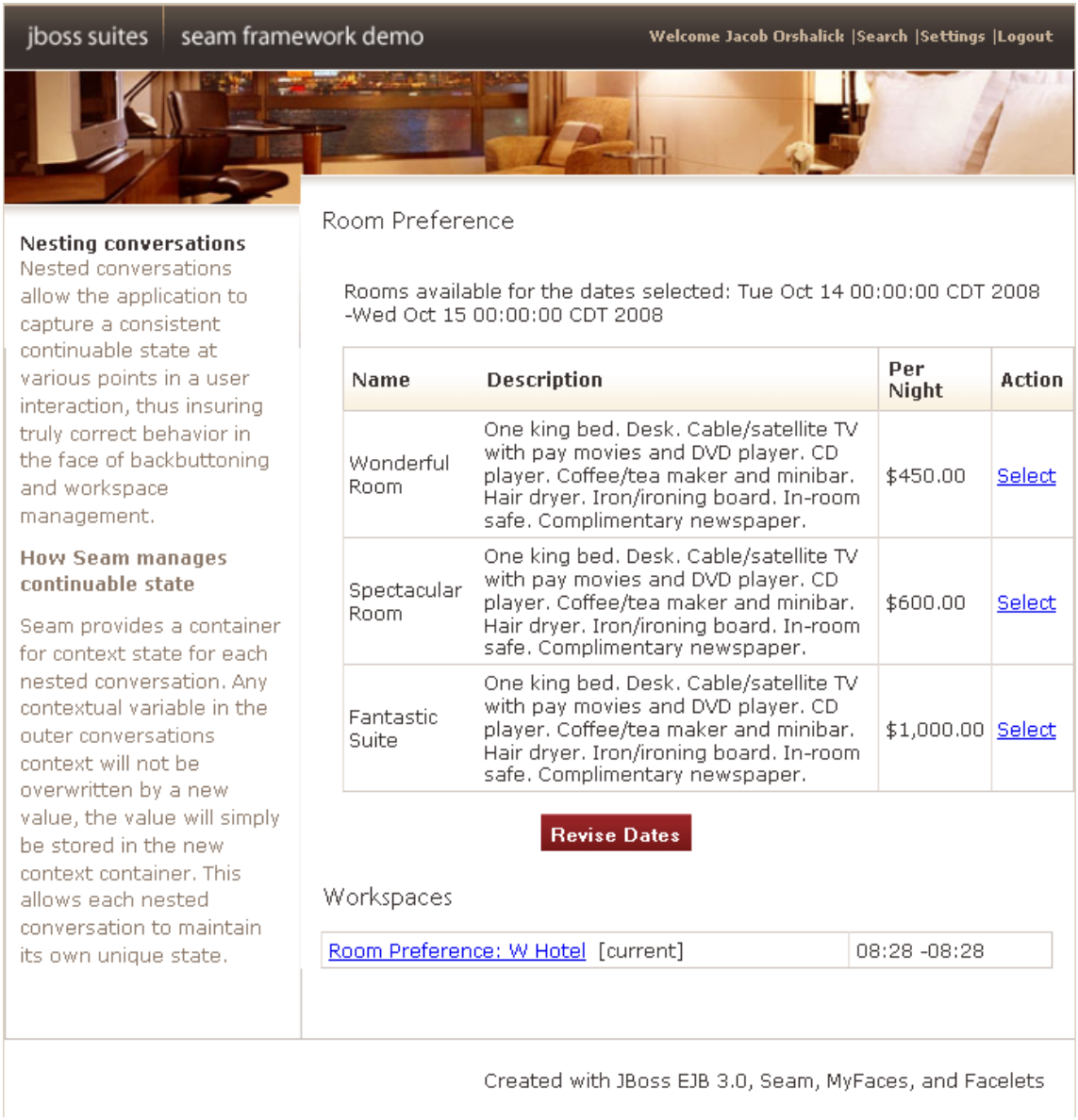
1.7. ネストされた対話 : ホテル予約サンプルの拡張

1.7.1. はじめに

長期実行の対話では、 複数ウインドウの操作や戻るボタン操作も含めてアプリケーション内の状態の整合性を容易に維持することができます。 残念ながら、 長期実行の対話を単に開始して終了するだけでは不十分な場合があります。 アプリケーション要件によっては、 ユーザーの期待値とアプリケー

ションの状態間で整合性を欠く場合があります。

ネストされた予約アプリケーションでは、部屋の選択機能を組み込むためにホテル予約アプリケーションの機能を拡張します。 各ホテルにはユーザーが選択できる宿泊可能な部屋の一覧があります。 これによりホテルの予約の流れに部屋選択ページを追加することが必要となります。



これでユーザーは宿泊可能な部屋を選択して予約に含めることができます。 部屋選択が同じ対話コンテキスト内に残っていた場合、 状態の整合性に関する問題を招く恐れがあります。 対話変数が変更されると同じ対話コンテキスト内のすべてのウィンドウ操作に影響します。

たとえば、 ユーザーが新しいウィンドウでまったく同じ部屋選択の画面を作成したとします。 次にユーザーは **Wonderful Room** を選択して確認画面に進みます。 もう少し高い料金の部屋を見るため元の画面に戻り、 **Fantastic Suite** を選択して再び確認画面に進みます。 総額を再確認した後、 ユーザーは **Wonderful Room** を表示しているウィンドウに戻り確認作業を行います。

このシナリオでは、すべての状態が対話に保存されると同じ対話内で複数ウィンドウを柔軟に操作することは難しくなります。 ネストされた対話では、 同じ対話内でコンテキストが変化する場合にも正しい動作をさせることができます。

1.7.2. ネストされた対話の理解

次のコードでは、ネストされた対話の拡張された動作によるホテル予約アプリケーションの動作を示しています。繰り返しになりますが、コードを順を追って読むステップの1セットとして考えると理解できます。

例1.31 RoomPreferenceAction.java

```
@Stateful
@Name("roomPreference")
@Restrict("#{identity.loggedIn}")
public class RoomPreferenceAction implements RoomPreference
{

    @Logger
        private Log log;

    @In private Hotel hotel;
```

```


    @In private Booking booking;

    @DataModel(value="availableRooms")
    private List<Room> availableRooms;

    @DataModelSelection(value="availableRooms")
    private Room roomSelection;

    @In(required=false, value="roomSelection")
    @Out(required=false, value="roomSelection")
    private Room room;

    @Factory("availableRooms")
    public void loadAvailableRooms()
    {
        availableRooms =
hotel.getAvailableRooms(booking.getCheckinDate(),

booking.getCheckoutDate());
        log.info("Retrieved #0 available rooms",
availableRooms.size());
    }

    public BigDecimal getExpectedPrice()
    {
        log.info("Retrieving price for room #0",
roomSelection.getName());

        return booking.getTotal(roomSelection);
    }

    @Begin(nested=true)
```

```
    public String selectPreference()
    {
```

```
        log.info("Room selected");

        this.room = this.roomSelection;
```

```
        return "payment";
    }

    public String requestConfirmation()
    {
        // all validations are performed through the s:validateAll, so
checks are
        // already performed
        log.info("Request confirmation from user");

        return "confirm";
    }

    @End(beforeRedirect=true)
```

```
    public String cancel()
    {
        log.info("ending conversation");

        return "cancel";
    }

    @Destroy @Remove
        public void destroy() {}
}
```

❶	**hotel** インスタンスは対話コンテキストからインジェクトされます。ホテルは**拡張永続コンテキスト** により読み込まれるため、エンティティは対話全体を通じて管理されたままとなります。これにより、単に関連付けることで、**@Factory** メソッドから**availableRooms** を遅延して読み込むことができます。
❷	**@Begin(nested=true)** が出現すると、 ネストされた対話は対話スタックにプッシュされます。ネストされた対話中で実行する場合、 コンポーネントはまだ外側の対話状態すべてにアクセスできますが、 ネストされた対話の状態コンテナに値を設定しても対話の外側には影響しません。 また、ネストされた対話は同じ外側の対話に対して同時並行的にスタックして存在することが可能で、それぞれの状態は独立しています。
❸	**roomSelection** は **@DataModelSelection** に基づいた対話にアウトジェクトされます。ネストされた対話は独立したコンテキストを持つので、**roomSelection** は新しいネストされた対話にのみ設定されることに留意してください。 ユーザーが別のウインドウまたはタブで異なる選択をした場合、新しいネストされた対話が開始されます。
❹	**@End** アノテーションは対話スタックをポップしてから外側の対話を再開します。**roomSelection** は対話コンテキストと共に破棄されます。

ネストした対話を開始すると対話スタックにプッシュされます。 **nestedbooking** サンプルでは、 対話スタックは外部の長期実行の対話 (予約) とネストした各対話 (部屋選択) から構成されます。

例1.32 rooms.xhtml

```
<div class="section">
  <h1>Room Preference</h1>
</div>

<div class="section">
  <h:form id="room_selections_form">
    <div class="section">
      <h:outputText styleClass="output"
         value="No rooms available for the dates selected: "
         rendered="#{availableRooms != null and availableRooms.rowCount
== 0}"/>
      <h:outputText styleClass="output"
         value="Rooms available for the dates selected: "
         rendered="#{availableRooms != null and availableRooms.rowCount
> 0}"/>

      <h:outputText styleClass="output" value="#
{booking.checkinDate}"/>
      <h:outputText styleClass="output" value="#
{booking.checkoutDate}"/>

      <br/><br/>

      <h:dataTable value="#{availableRooms}" var="room"
(1)

         rendered="#{availableRooms.rowCount > 0}">
        <h:column>
          <f:facet name="header">Name</f:facet>
          #{room.name}
        </h:column>
        <h:column>
          <f:facet name="header">Description</f:facet>
          #{room.description}
        </h:column>
        <h:column>
          <f:facet name="header">Per Night</f:facet>
          <h:outputText value="#{room.price}">
            <f:convertNumber type="currency" currencySymbol="$"/>
          </h:outputText>
        </h:column>
        <h:column>
          <f:facet name="header">Action</f:facet>
          <h:commandLink id="selectRoomPreference"
            action="#

{roomPreference.selectPreference}">Select</h:commandLink>
(2)
        </h:column>
      </h:dataTable>
    </div>
```

```
    <div class="entry">
      <div class="label"> </div>
      <div class="input">
        <s:button id="cancel" value="Revise Dates" view="/book.xhtml"/>
3
      </div>
    </div>
  </h:form>
</div>
```

1	EL から求められると、**RoomPreferenceAction** に定義された**@Factory** メソッドにより**#{availableRooms}** がロードされます。**@Factory** メソッドは、**@DataModel** インスタンスとして現在のコンテキストに値をロードするために 1 度だけ実行されます。
2	**#{roomPreference.selectPreference}** アクションを呼び出すことにより、行が選択され**@DataModelSelection** に値が設定されます。 そして値はネストされた対話コンテキストにアウトジェクトされます。
3	日付を変更すると単純に **/book.xhtml** に戻されます。 まだ対話をネストしていないため (選択された部屋がない)、 現在の対話は単に再開可能であることに留意してください。**<s:button>** コンポーネントは **/book.xhtml** ビューを表示するときに単に現在の対話を伝播します。

ここまでは対話をネストする方法を見てきました。 次のコードでは、 **HotelBookingAction** の動作を拡張して選択した部屋の予約を確認する方法を示しています。

例1.33 HotelBookingAction.java

```
@Stateful
@Name("hotelBooking")
@Restrict("#{identity.loggedIn}")
public class HotelBookingAction implements HotelBooking
{

   @PersistenceContext(type=EXTENDED)
   private EntityManager em;

   @In
   private User user;

   @In(required=false) @Out
   private Hotel hotel;

   @In(required=false)
   @Out(required=false)
   private Booking booking;

   @In(required=false)
   private Room roomSelection;

   @In
   private FacesMessages facesMessages;
```

```
   @In
   private Events events;

   @Logger
   private Log log;

   @Begin
   public void selectHotel(Hotel selectedHotel)
   {
      log.info("Selected hotel #0", selectedHotel.getName());
      hotel = em.merge(selectedHotel);
   }

   public String setBookingDates()
   {
      // the result will indicate whether or not to begin the nested
conversation
      // as well as the navigation.  if a null result is returned, the
nested
      // conversation will not begin, and the user will be returned to
the current
      // page to fix validation issues
      String result = null;

      Calendar calendar = Calendar.getInstance();
      calendar.add(Calendar.DAY_OF_MONTH, -1);

      // validate what we have received from the user so far
      if ( booking.getCheckinDate().before( calendar.getTime() ) )
      {
         facesMessages.addToControl("checkinDate",
                                    "Check in date must be a future
date");
      }
      else if ( !booking.getCheckinDate().before(
booking.getCheckoutDate() ) )
      {
         facesMessages.addToControl("checkoutDate",
                                    "Check out date must be later than
check in date");
      }
      else
      {
         result = "rooms";
      }

      return result;
   }

   public void bookHotel()
   {
      booking = new Booking(hotel, user);
      Calendar calendar = Calendar.getInstance();
      booking.setCheckinDate( calendar.getTime() );
      calendar.add(Calendar.DAY_OF_MONTH, 1);
```

```
      booking.setCheckoutDate( calendar.getTime() );
   }

   @End(root=true)
```

```
   public void confirm()
   {
      // on confirmation we set the room preference in the booking.  the
room preference
      // will be injected based on the nested conversation we are in.
      booking.setRoomPreference(roomSelection);
```

```
      em.persist(booking);
      facesMessages.add("Thank you, #{user.name}, your confimation
number" +
                        " for #{hotel.name} is #{booking.id}");
      log.info("New booking: #{booking.id} for #{user.username}");
      events.raiseTransactionSuccessEvent("bookingConfirmed");
   }

   @End(root=true, beforeRedirect=true)
```

```
   public void cancel() {}

   @Destroy @Remove
   public void destroy() {}
}
```

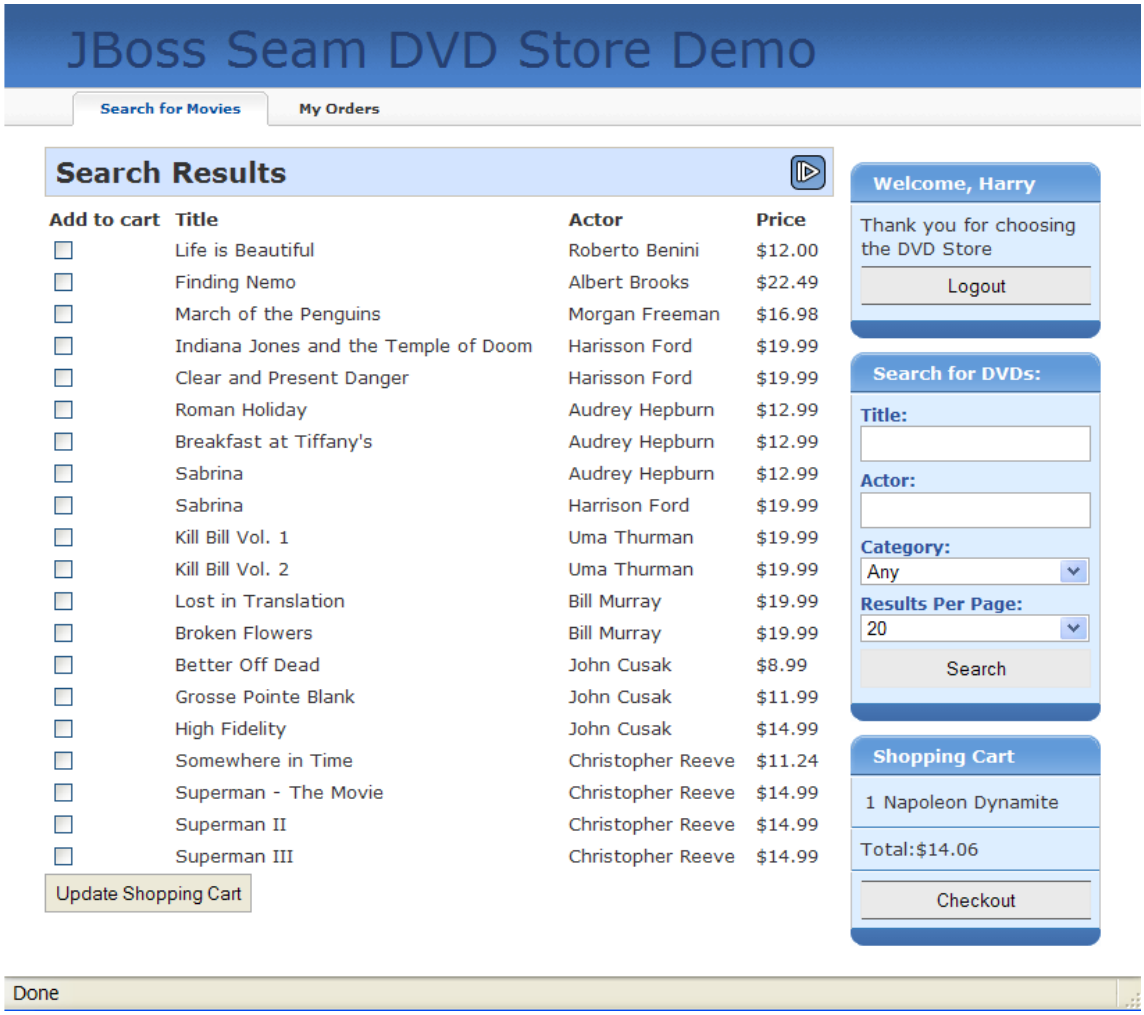
1	動作に **@End(root=true)** アノテーションを付けるとルートの対話を終了させます。 これは効率的に対話スタック全体を破棄します。 対話が終了したらその中にネストされた対話も終了されます。 ルートはオリジナルの対話であるため、これが予約の確認が終了したら作業領域に関連付けられたすべての状態を破棄して解放するシンプルな方法です。
2	**roomSelection** はユーザー確認で**booking** にのみ関連付けられます。 ネストされた対話コンテキストに値をアウトジェクトしても外部の対話には影響ありませんが、 外側の対話からインジェクトされるオブジェクトは参照によりインジェクトされます。 つまり、 これらのオブジェクトに対する変更はすべて親の対話で反映されるだけでなく、 他の同時にネストされた対話にも反映されます。
3	取り消し動作に **@End(root=true, beforeRedirect=true)** アノテーションを付与するだけで、 作業領域に関連するすべての状態を容易に破棄、 解放してからユーザーをホテル選択ビューにリダイレクトさせることができます。

ぜひアプリケーションをデプロイしてご自身でテストしてみてください。多くのウィンドウやタブを開き、 さまざまなホテルと部屋の組み合わせで試してみてください。予約の確認作業を行うとネストされた対話モデルにより常に正しいホテルと部屋が表示されます。

1.8. SEAM と JBPM を使ったアプリケーションの全容 : DVD ストアサンプル

DVD ストアのデモアプリケーションは、 タスク管理とページフローのための jBPM の実践的な使用法を見せてくれます。

ユーザー画面は検索やショッピングカート機能を実装するため jPDL ページフローの利点を利用しています。

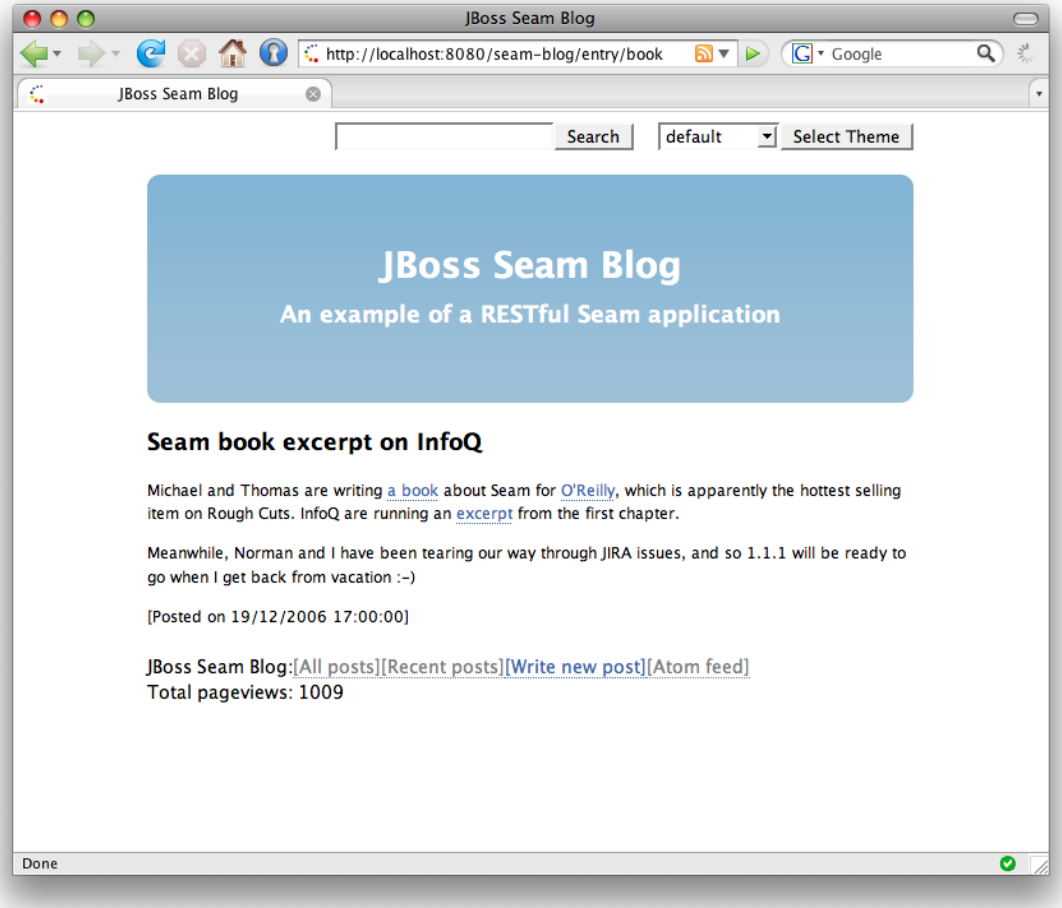


この管理画面は jBPM を利用して、注文の承認やショッピングサイクルを管理しています。ビジネスプロセスも異なるプロセス定義を選択することで動的に変更することができます。

Seam の DVD ストアデモは前述のアプリケーションと同様、**dvdstore** ディレクトリから起動できます。

1.9. ブログサンプルを使ったブックマーク可能な URL の説明

Seam によりサーバー側で状態を保持するアプリケーションの実装が容易になります。 しかし、 特にコンテンツを提供する機能の場合など、 サーバー側の状態が常に適切であるとは限りません。 このため、 アプリケーションの状態は URL の一部として保存されることが多く、 これによりいつでもどのページにもブックマークを使ってアクセスが可能となります。 ブログサンプルでは検索結果ページも含め全体的にブックマーク機能に対応するアプリケーションの実装方法を示しています。 このサンプルでは Seam による URL でのアプリケーション状態の管理を示します。



このブログサンプルでは「プル (PULL)」スタイルのモデルビューコントロール (MVC) の使い方を示します。ビュー用のデータ取得および準備にアクションリスナーのメソッドを使用するのではなく、データは表示されているためビューはコンポーネントからデータを引き出します。

1.9.1. 「プル」型 MVC の使用

index.xhtml Facelets ページの一部は最新のブログエントリの一覧を表示しています。

```
<h:dataTable value="#{blog.recentBlogEntries}" var="blogEntry" rows="3">
  <h:column>
    <div class="blogEntry">
      <h3>#{blogEntry.title}</h3>
      <div>
        <s:formattedText value="#{blogEntry.excerpt==null ?
                                  blogEntry.body : blogEntry.excerpt}"/>
      </div>
      <p>
        <s:link view="/entry.xhtml" rendered="#{blogEntry.excerpt!=null}"
                propagation="none" value="Read more...">
          <f:param name="blogEntryId" value="#{blogEntry.id}"/>
        </s:link>
      </p>
      <p>
        [Posted on 
        <h:outputText value="#{blogEntry.date}">
          <f:convertDateTime timeZone="#{blog.timeZone}"
                             locale="#{blog.locale}" type="both"/>
        </h:outputText>]
```

```
             
            <s:link view="/entry.xhtml" propagation="none" value="[Link]">
              <f:param name="blogEntryId" value="#{blogEntry.id}"/>
            </s:link>
        </p>
      </div>
    </h:column>
</h:dataTable>
```

ブックマークからこのページに移動した場合、 **<h:dataTable>** で使用される **#{blog.recentBlogEntries}** データは要求されると Seam の **blog** というコンポーネントにより遅延して読み出されます(「プルされる」)。 この制御の流れは従来の動作ベースの Web フレームワーク Struts で使用されているものとは逆です 。

例1.34

```
 @Name("blog")
@Scope(ScopeType.STATELESS)
@AutoCreate
public class BlogService
{
   @In EntityManager entityManager;
```

(1)

```
   @Unwrap
```

(2)

```
   public Blog getBlog()
   {
      return (Blog) entityManager.createQuery("select distinct b from
Blog b left join fetch b.blogEntries")
         .setHint("org.hibernate.cacheable", true)
         .getSingleResult();
   }
}
```

(1)	このコンポーネントは **Seam 管理永続コンテキスト** を使用しています。 これまで見てきた他のサンプルとは異なり、 この永続コンテキストは EJB3 コンテナではなく Seam によって管理されます。 永続コンテキストは Web 要求全体におよび、 ビューでフェッチしていない関連にアクセスすると発生する例外を回避することができます。
(2)	**@Unwrap** アノテーションは Seam にクライアントに対する実際の**BlogService** コンポーネントではなく **Blog** メソッドの戻り値を与えるよう指示します。 これが Seam の**マネージャコンポーネントパターン** です。

これは基本的なビューコンテンツを保存しますが、 検索結果ページのようなフォームサブミッションの結果もブックマークします。 この他必要な定義がいくつかあります。

1.9.2. ブックマーク可能な検索結果ページ

このブログサンプルには各ページの右上に小さなフォームがあり、ユーザーはブログ記事を検索することが可能です。 **menu.xhtml** に定義され、Facelet テンプレートの **template.xhtml** で含まれます。

```
<div id="search">
  <h:form>
    <h:inputText value="#{searchAction.searchPattern}"/>
    <h:commandButton value="Search" action="/search.xhtml"/>
  </h:form>
</div>
```

ブックマーク可能な検索結果ページを実装するには、 検索フォームのサブミットを処理した後に、 ブラウザのリダイレクトを実行する必要があります。 アクションの結果として JSF ビュー ID を使用しているので、 Seam はフォームがサブミットされると自動的に ビュー ID にリダイレクトします。 別の方法として、 以下のようなナビゲーションルールを定義することも可能です。

```
<navigation-rule>
  <navigation-case>
    <from-outcome>searchResults</from-outcome>
    <to-view-id>/search.xhtml</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

この場合、 フォームは以下と似たようなものになるでしょう。

```
<div id="search">
  <h:form>
    <h:inputText value="#{searchAction.searchPattern}"/>
    <h:commandButton value="Search" action="searchResults"/>
  </h:form>
</div>
```

ただし、 **http://localhost:8080/seam-blog/search/** のようなブックマーク可能な URL を取得するには、フォームでサブミットされる値をその URL に含ませる必要があります。 JSF でこれを行うのは簡単ではありませんが、 Seam なら **ページパラメータ** と **URL の書き換え** の 2 つの機能があれば行うことができます。 いずれも **WEB-INF/pages.xml** で定義します。

```
 <pages>
  <page view-id="/search.xhtml">
    <rewrite pattern="/search/{searchPattern}"/>
    <rewrite pattern="/search"/>

    <param name="searchPattern" value="#{searchService.searchPattern}"/>

  </page>
  ...
</pages>
```

検索ページが求められたり、 検索ページへのリンクが生成されたりする場合は必ず、**searchPattern**

要求パラメータを **#{searchService.searchPattern}** が保持する値にリンクするようページパラメータが Seam に指示します。 Seam は URL の状態ととアプリケーションの状態を結ぶリンクを管理する役割を担います。

用語 ***book*** の検索 URL は、通常 **http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern=book** です。Seam は書き換えのルールを使ってこの URL を簡略化することが可能です。**/search/{searchPattern}** パターンの最初の書き換えルールには、search.xhtml の URL が searchPattern 要求パラメータを含む場合は常にその URL は簡略化された URL に圧縮されることが可能であると記述しています。このため、前述した URL (**http://localhost:8080/seam-blog/seam/search.xhtml?searchPattern= book**) は、**http://localhost:8080/seam-blog/search/book** と記述することが可能です。

ページパラメータと同様に、 URL の書き換えは両方向です。 Seam は簡略化した URL の要求を適切なビューに転送し自動的に簡略化ビューを生成するため、 ユーザーは URL を構成する必要がありません。全プロセスは透過的に処理されます。 URL の書き換えに必要なのは **components.xml** で書き換えフィルタを有効にすることだけです。

```
<web:rewrite-filter view-mapping="/seam/*" />
```

リダイレクトによって **search.xhtml** ページに移動します。

```
<h:dataTable value="#{searchResults}" var="blogEntry">
 <h:column>
   <div>
      <s:link view="/entry.xhtml" propagation="none"
              value="#{blogEntry.title}">
        <f:param name="blogEntryId" value="#{blogEntry.id}"/>
      </s:link>
      posted on
      <h:outputText value="#{blogEntry.date}">
        <f:convertDateTime timeZone="#{blog.timeZone}"
                           locale="#{blog.locale}" type="both"/>
      </h:outputText>
    </div>
  </h:column>
</h:dataTable>
```

これもまた Hibernate Search を使用し実際の検索結果を取得するために「プル」型 MVC を使用します。

```
@Name("searchService")
public class SearchService {

  @In
  private FullTextEntityManager entityManager;

  private String searchPattern;

  @Factory("searchResults")
  public List<BlogEntry> getSearchResults() {
  if (searchPattern==null || "".equals(searchPattern) )
    {
      searchPattern = null;
      return entityManager.createQuery(
                "select be from BlogEntry be order by date desc"
```

```
                ).getResultList();
    }
      else
        {
          Map<String,Float> boostPerField = new HashMap<String,Float>();
          boostPerField.put( "title", 4f );
          boostPerField.put( "body", 1f );
          String[] productFields = {"title", "body"};
          QueryParser parser = new MultiFieldQueryParser(productFields,
                               new StandardAnalyzer(), boostPerField);
          parser.setAllowLeadingWildcard(true);
          org.apache.lucene.search.Query luceneQuery;
          try
            {
              luceneQuery = parser.parse(searchPattern);
            }
          catch (ParseException e)
            {
              return null;
            }

          return entityManager
            .createFullTextQuery(luceneQuery, BlogEntry.class)
            .setMaxResults(100)
            .getResultList();
        }
    }

    public String getSearchPattern()
    {
      return searchPattern;
    }

    public void setSearchPattern(String searchPattern)
    {
      this.searchPattern = searchPattern;
    }

}
```

1.9.3. RESTful アプリケーションの「プッシュ」型 MVC の使用

RESTful ページの処理にプッシュ型 MVC が使われることがあるため、 Seam では **ページアクション** という概念を提供しています。 ブログのサンプルはブログの記入ページ **entry.xhtml** にページアクションを使用しています。

注記

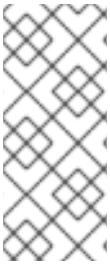
ここでは一例として示すためにプッシュ型を使用していますが、 この特定の機能についてはプル型 MVC を使った実装の方がシンプルです。

entryAction コンポーネントの動作は Struts のような従来のプッシュ型 MVC アクション指向のフレームワークでのアクションクラスの動作とよく似ています。

```
@Name("entryAction")
@Scope(STATELESS)
public class EntryAction
{
    @In Blog blog;

    @Out BlogEntry blogEntry;

    public void loadBlogEntry(String id) throws EntryNotFoundException {
        blogEntry = blog.getBlogEntry(id);
        if (blogEntry==null) throw new EntryNotFoundException(id);
    }

}
```

ページアクションは、**pages.xml**でも宣言されます。

```
<pages>
  ...

  <page view-id="/entry.xhtml">
    <rewrite pattern="/entry/{blogEntryId}" />
    <rewrite pattern="/entry" />

    <param name="blogEntryId"
           value="#{blogEntry.id}"/>

    <action execute="#{entryAction.loadBlogEntry(blogEntry.id)}"/>
  </page>

  <page view-id="/post.xhtml" login-required="true">
    <rewrite pattern="/post" />

    <action execute="#{postAction.post}"
            if="#{validation.succeeded}"/>

    <action execute="#{postAction.invalid}"
            if="#{validation.failed}"/>

    <navigation from-action="#{postAction.post}">
      <redirect view-id="/index.xhtml"/>
    </navigation>
  </page>

  <page view-id="*">
    <action execute="#{blog.hitCount.hit}"/>
  </page>

</pages>
```



注記

このサンプルは検証後およびページビューのカウンタにページアクションを使用している点に留意してください。 また、 ページアクションのメソッドバインディングでのパラメータの使い方にも注意してください。 これは標準 JSF EL の機能ではありませんが、 Seam ではページアクションだけでなく JSF メソッドバインディングでも使用することができます。

entry.xhtml ページが要求されると Seam はまずページパラメータ **blogEntryId** をそのモデルにバインドします。 URL を書き換えているため blogEntryId パラメータ名は URL に表れないことを思い出してください。 次に Seam はページアクションを実行して、 必要なデータ **blogEntry** を読み出し、それを Seam イベントコンテキスト内に配置します。 最後に以下を表示させます。

```
<div class="blogEntry">
  <h3>#{blogEntry.title}</h3>
  <div>
    <s:formattedText value="#{blogEntry.body}"/>
  </div>
  <p>
  [Posted on 
  <h:outputText value="#{blogEntry.date}">
     <f:convertDateTime timeZone="#{blog.timeZone}" locale="#
{blog.locale}" type="both"/>
  </h:outputText>]
  </p>
</div>
```

ブログエントリがデータベースで見つからない場合、 **EntryNotFoundException** 例外が送出されます。 この例外は 505 エラーではなく 404 エラーにさせたいので例外クラスにアノテーションを付けます。

```
@ApplicationException(rollback=true)
@HttpError(errorCode=HttpServletResponse.SC_NOT_FOUND)
public class EntryNotFoundException extends Exception {
    EntryNotFoundException(String id) {
        super("entry not found: " + id);
    }
}
```

メソッドバインディングでパラメータを使用しない別の実装例を示します。

```
@Name("entryAction")
@Scope(STATELESS)
public class EntryAction {
    @In(create=true)
        private Blog blog;

    @In @Out
    private BlogEntry blogEntry;

    public void loadBlogEntry() throws EntryNotFoundException {
        blogEntry = blog.getBlogEntry( blogEntry.getId() );
```

```
        if (blogEntry==null) throw new EntryNotFoundException(id);
    }
}
```

```
<pages>
  ...
  <page view-id="/entry.xhtml" action="#{entryAction.loadBlogEntry}">
  <param name="blogEntryId" value="#{blogEntry.id}"/>
</page>
  ...
</pages>
```

使用する実装はその選択により完全に異なります。

また、ブログのサンプルでは非常にシンプルなパスワード認証、ブログへの投稿、ページの部分的なキャッシング、Atom フィードの生成も示しています。

第2章 移行

Seam の旧バージョンがすでにインストールされている場合は本章の説明にしたがって最新バージョン (2.0.0) に移行する必要があります。 最新バージョンは JBoss Enterprise Application Platform に同梱されています。

Seam 2.0 をすでに使用している場合は、直接 「Seam 2.0 から Seam 2.1 または 2.2 への移行」 まで進んでください。現在 Seam 1.2.x を使用している場合は、「Seam 1.2.x から Seam 2.0 への移行」 および 「Seam 2.0 から Seam 2.1 または 2.2 への移行」 の両方の説明に従ってください。

2.1. SEAM 1.2.X から SEAM 2.0 への移行

本項では Seam 1.2.x から Seam 2.0 に移行する方法について見ていきます。 また、 バージョン間での Seam コンポーネントへの変更点も記載しています。

2.1.1. JavaServer Faces 1.2 への移行

Seam 2.0 は正しく動作するには JSF 1.2 が必要です。 JBoss 4.2 などほとんどの Java EE 5 アプリケーションサーバーに同梱される Sun の JSF Reference Implementation (RI) を推奨します。 JSF RI に切り替えるには **web.xml** に次の変更を行う必要があります。

- MyFaces の **StartupServletContextListener** を削除します。
- AJAX4JSF フィルタ、 マッピング、 **org.ajax4jsf.VIEW_HANDLERS** コンテキストパラメータを削除します。
- **org.jboss.seam.web.SeamFilter** の名前を **org.jboss.seam.servlet.SeamFilter** に変更します。
- **org.jboss.seam.servlet.ResourceServlet** の名前を **org.jboss.seam.servlet.SeamResourceServlet** に変更します。
- **web-app** のバージョンを **2.4** から **2.5** に変更します。名前空間 URL で **j2ee** を **javaee** に変更します。 たとえば、

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
  http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
  ...
</web-app>
```

Seam 1.2 では、 **SeamExceptionFilter** と **SeamRedirectFilter** を **web.xml** に明示的に宣言せずに **SeamFilter** を **web.xml** に宣言することができます。

クライアント側の状態保存には JSF RI を必要としないため削除が可能です。クライアント側の状態保存は **javax.faces.STATE_SAVING_METHOD** コンテキストパラメータで定義します。

また、 **faces-config.xml** に以下の変更が必要になります。

- **TransactionalSeamPhaseListener** または **SeamPhaseListener** の宣言を使用している場合は削除します。

- **SeamELResolver** 宣言を使用している場合は削除します。
- **SeamFaceletViewHandler** 宣言を標準の **com.sun.facelets.FaceletViewHandler** に変更してからそれが有効になっていることを確認します。
- ドキュメント上の DTD を削除して XML Schema 宣言をその **<faces-config>** ルートタグに追加します。

```
<faces-config version="1.2"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
   http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
   ...
</faces-config>
```

2.1.2. コードの移行

Seam の組み込みコンポーネントは再編成されより簡単に理解できるようになり、 特定の技術依存と固有パッケージが分離されました。

- 永続関連のコンポーネントは **org.jboss.seam.persistence** に移動しました。
- jBPM 関連のコンポーネントは **org.jboss.seam.bpm** に移動しました。
- JSF 関連のコンポーネント、 特に **org.jboss.seam.faces.FacesMessages** は **org.jboss.seam.faces** に移動しました。
- サーブレット関連のコンポーネントは **org.jboss.seam.web** に移動しました。
- 非同期に関連するコンポーネントは **org.jboss.seam.async** に移動しました。
- 国際化関連のコンポーネントは **org.jboss.seam.international** に移動しました。
- ページフローのコンポーネントは **org.jboss.seam.pageflow** に移動しました。
- ページコンポーネントは **org.jboss.seam.navigation** に移動しました。

こうした API に依存するコードはすべて新しい Java パッケージ名を反映するよう変更する必要があります。

アノテーションについても再編成が行われました。

- BPM 関連のアノテーションは **org.jboss.seam.annotations.bpm** パッケージに含まれています。
- JSF 関連のアノテーションは **org.jboss.seam.annotations.faces** パッケージに含まれています。
- インターセプタのアノテーションは **org.jboss.seam.annotations.intercept** パッケージに含まれています。
- 非同期に関連するアノテーションは **org.jboss.seam.annotations.async** パッケージに含まれています。

- **@RequestParameter** は **org.jboss.seam.annotations.web** パッケージに含まれています。
- **@WebRemote** は **org.jboss.seam.annotations.remoting** パッケージに含まれています。
- **@Restrict** は **org.jboss.seam.annotations.security** パッケージに含まれています。
- 例外処理のアノテーションは **org.jboss.seam.annotations.exception** パッケージに含まれています。
- **@Intercept(NEVER)** ではなく **@BypassInterceptors** を使用します。

2.1.3. components.xml の移行

前項で概要を説明した新しいパッケージングシステムにより、**components.xml** を新しいスキーマと名前空間で更新する必要があります。

名前空間は当初は **org.jboss.seam.foobar** の形式をとっていました。 新しい名前空間の形式は **http://jboss.com/products/seam/foobar** になり、 スキーマの形式は **http://jboss.com/products/seam/foobar-2.0.xsd** です。 **components.xml** ファイルで名前空間とスキーマの形式を更新する必要があります。 これにより URL は移行先となる Seam のバージョン (2.0 または 2.1) に対応するようになります。

次の宣言はその位置を修正するか完全に削除する必要があります。

- **<core:managed-persistence-context>** を **<persistence:managed-persistence-context>** で置換します。
- **<core:entity-manager-factory>** を **<persistence:entity-manager-factory>** で置換します。
- **<core:manager/>** エレメントから **conversation-is-long-running** パラメータを削除します。
- **<core:ejb/>** を削除します。
- **<core:microcontainer/>** を削除します。
- **<core:transaction-listener/>** を **<transaction:ejb-transaction/>** で置換します。
- **<core:resource-bundle/>** を **<core:resource-loader/>** で置換します。

例2.1 components.xml の注記

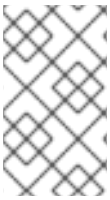
Seam トランザクション管理はデフォルトで有効です。 **faces-config.xml** の JSF フェーズリスナー宣言ではなく、**components.xml** で制御されています。 Seam 管理トランザクションを無効にする場合は次を使用します。

```
<core:init transaction-management-enabled="false"/>
```

event action にある **expression** 属性は廃止され **execute** になります。 たとえば、

```
<event type="org.jboss.seam.security.notLoggedIn">
   <action execute="#{redirect.captureCurrentView}"/>
</event>
<event type="org.jboss.seam.loginSuccessful">
   <action execute="#{redirect.returnToCapturedView}"/>
</event>
```

Seam 2.2 では、セキュリティイベントは **org.jboss.seam** ではなく **org.jboss.seam.security>** プレフィックスを使用します (例えば **org.jboss.seam.security.notLoggedIn**)。



注記

org.jboss.seam.postAuthenticate イベントの代わりに、**org.jboss.seam.security.loginSuccessful** イベントを使ってキャプチャしたビューに戻ります。

2.1.4. Embedded JBoss への移行

Embedded JBoss は JBoss Embeddable EJB3 または JBoss Microcontainer の使用に対応しなくなりました。代わりに、 新しい Embedded JBoss ディストリビューションにより簡略化されたデプロイメントを持つ Java EE 互換の API セットが提供されます。

テスト行う場合には次をクラスパスに含ませる必要があります。

- Seam の **lib/** ディレクトリにある **jar**
- **bootstrap/** ディレクトリ

embeddded-ejb ディレクトリや **jboss-beans.xml** など JBoss Embeddable EJB3 関連の参照やアーティファクトはすべて削除します。 (Seam のサンプルを参照として使用できます。)

Tomcat デプロイメントに関する特殊な設定やパッケージング要件はすべてなくなりました。Tomcat でデプロイする場合はユーザーガイドの説明にしたがってください。

注記

Embedded JBoss はデータソースを **-ds.xml** ファイルからブートストラップできるため、 **jboss-beans.xml** ファイルは必要なくなりました。

2.1.5. jBPM 3.2 への移行

ビジネスプロセスに対する jBPM およびページフローを使用している場合は **tx** サービスを **jbpm.cfg.xml** に追加する必要があります。

```
<service name="tx" factory="org.jbpm.tx.TxServiceFactory" />
```

2.1.6. RichFaces 3.1 への移行

RichFaces および AJAX4JSF にはコードベースの大幅な再編成が行われました。 **ajax4jsf.jar** と **richfaces.jar** の **jar** は **richfaces-api.jar** (EAR の **lib/** ディレクトリに配置される)、**richfaces-impl.jar** および **richfaces-ui.jar** (**WEB-INF/lib** に配置される) で置き換えられま

した。

<s:selectDate> は廃止され **<rich:calendar>** になります。 **<s:selectDate>** の開発は行われなくなります。 ご使用のスタイルシートからデータピッカー関連のスタイルを削除してバンド幅の使用を減らします。

名前空間およびパラメータ名に対する変更については RichFaces のドキュメントを確認してください。

2.1.7. コンポーネントにおける変更点

パッケージングの変更点

application.xml でモジュールとしてこれまで宣言されていた依存性は、**jboss-seam.jar** を **除き** すべて EAR の **lib/** ディレクトリに配置されるようになるはずです。**jboss-seam.jar** は EJB モジュールとして **application.xml** に宣言されるはずです。

Seam のユーザーインターフェースにおける変更点

<s:decorate> は命名コンテナになりました。 したがってクライアント ID は **fooForm:fooInput** から **fooForm:foo:fooInput** に変更になり、 次の宣言を前提としています。

```
<h:form id="fooForm">
  <s:decorate id="foo">
     <h:inputText id="fooInput" value="#{bean.property}"/>
   </s:decorate>
</h:form>
```

ID を **<s:decorate>** に与えないと JSF は ID を自動的に生成します。

seam-gen における変更点

Seam 2.0.0.CR2 からは、 **generate-entities** が実行されるときに **seam-gen** で生成されたクラスの編成に変更が発生しました。

元々はクラス郡は次のように生成されました。

```
        src/model/com/domain/projectname/model/EntityName.java
        src/action/com/domain/projectname/model/EntityNameHome.java
        src/action/com/domain/projectname/model/EntityNameList.java
```

現在では次のように生成されます。

```
        src/model/com/domain/projectname/model/EntityName.java
        src/action/com/domain/projectname/action/EntityNameHome.java
        src/action/com/domain/projectname/action/EntityNameList.java
```

Home および Query のオブジェクトは **モデル** コンポーネントではなく **アクション** コンポーネントになるため、 **action** パッケージに配置されます。 これにより **generate-entities** 対話が **new-entity** コマンドのそれと整合性を持ちます。

モデルのクラスはホット再ロードができないため別々に記載されます。

テストシステムの JBoss Embeddable EJB3 から Embedded JBoss への変更に伴い、 Seam 2.x の **seam-gen** でプロジェクトを生成し、その **build.xml** ファイルを新しいプロジェクトのベースとして使用することをお勧めします。 **build.xml** に大幅な変更を行っている場合は、テスト関連の対象のみの移行に焦点を絞ることが可能です。

Embedded JBoss で動作するかのテストでは、**resources/META-INF/persistence-test.xml** (または **persistence-test-war.xml**) の **<datasource>** エレメントの値を **java:/DefaultDS** に変更する必要があります。 代わりに **-ds.xml** ファイルを **bootstrap/deploy** フォルダにデプロイして、そのファイルで定義される JNDI 名を使用する方法もあります。

説明のとおり Seam 2.x **build.xml** を使用する場合は **deployed-*.list** ファイルも必要になります。 これらのファイルは EAR または WAR アーカイブにパッケージ化される **jar** ファイル群を指定します。 **build.xml** ファイルから **jar** セットを取り除くために導入されました。

次の CSS をスタイルシートに追加してスタイルシートが RichFaces パネルでの変更に対応できるようにします。 このコードの追加に失敗すると **generate-entities** で作成されるすべてのページで**検索基準** ブロックが **結果表** に流出することになります。

```
.rich-stglpanel-body {
        overflow: auto;
        }
```

2.2. SEAM 2.0 から SEAM 2.1 または 2.2 への移行

本項では Seam 2.0 と比較した Seam 2.1 または 2.2 における変更点を説明します。 Seam 1.2.x から移行する場合には前項 「Seam 1.2.x から Seam 2.0 への移行」 を先にお読み頂いてから、 本項の手順に進んでください。

2.2.1. 依存 jar の名前における変更点

Seam フレームワークに含まれる JAR、除去された他の JAR の一覧については 表2.1「含まれた JAR」 および 表2.2「削除された JAR」 をそれぞれ参照してください。

表2.1 含まれた JAR

ファイル名	詳細
ant-launcher.jar	
common-codec.jar	
commons-httpclient.jar	
concurrent.jar	
darkX.jar	新しいプラグ可能な RichFaces スキン *DarkX*
drools-api.jar	Drools 5 API
drools-decisiontables.jar	Drools 5 ディシジョンルールの機能
drools-templates.jar	Drools 5 ルールテンプレートの機能
ehcache.jar	
glassX.jar	新しいプラグ可能な RichFaces スキン *GlassX*

ファイル名	詳細
hibernate-core.jar	
htmlparser.jar	HTML パーサー、OpenID 機能の依存性
httpclient.jar	
httpcore.jar	
itext-rtf.jar	itext から RTF にエクスポートするときの拡張オプションの依存性
jaxrs-api.jar	
jbosscache-core.jar	
jboss-common-core.jar	
jboss-logging.spi.jar	
jboss-seam-excel.jar	Microsoft Excel 統合モジュール
jboss-seam-resteasy.jar	RestEasy 統合モジュール
jboss-transaction-api.jar	
jboss-vfs.jar	
jcip-annotations.jar	
jcl-over-slf4j.jar	レイテンシロギング API のブリッジ、Resteasy 統合モジュールの依存性
jettison.jar	
jms.jar	
joda-time.jar	
junit.jar	
jxl.jar	Microsoft Excel 統合モジュールの依存性
laguna.jar	新しいプラグ可能な RichFaces スキン *laguna*
mvel2.jar	Drools 5 向けの式言語の依存性

ファイル名	詳細
openid4java.jar	Security Seam モジュールでの統合のための OpenID Java API
openxri-client.jar	OpenRXI resolver、OpenID 統合の依存性
openrxi-syntax.jar	OpenXRI パーサー、OpenID 統合の依存性
resteasy-atom-provider.jar	Resteasy 統合モジュールの依存性
resteasy-jaxb-provider.jar	Seam の Resteasy 統合モジュールの依存性
resteasy-jaxrs.jar	Resteasy 統合モジュールの依存性
resteasy-jettison-provider.jar	
slf4j-api.jar	Hibernate および他の依存関係が使用する **log4j** のロギングブリッジ
slf4j-log4j12.jar	Hibernate および他の依存関係が使用する **log4j** のロギングブリッジ
testng-jdk15.jar	TestNG フレームワーク

削除された JAR の多くは、Platform の複数のバージョンから除外されました。このリストは歴史的な目的のために含めました。

表2.2 削除された JAR

JAR	削除の理由
activation.jar	アクティブ化は Java 6 に組み込まれているため、ディストリビューションから削除できます。
commons-lang.jar	Commons Lang ライブラリは必要でなくなりました。
geronimo-jms_1.1_spec.jar	
geronimo-jtaB_spec-1.0.1.jar	
hibernate3.jar	
jboss-cache-jdk50.jar	
jboss-jmx.jar	
jboss-system.jar	

JAR	削除の理由
mvel.jar	
testng.jar	

2.2.2. コンポーネントにおける変更点

テスト

SeamTest は各クラスの起動時ではなく各スィートの起動時に Seam を起動するようになり、速度が向上します。 デフォルトを変更したい場合はリファレンスガイドをご確認ください。

DTD およびスキーマの形式における変更点

Seam XML ファイルの Document Type Declarations (DTD) には対応しなくなります。 検証には代わりに XML Schema Declaration (XSD) を使用してください。 Seam 2.0 XSD を使用するファイルはすべて Seam 2.1 XSD を参照するよう更新が必要です。

例外処理における変更点

キャッチされた例外が **#{org.jboss.seam.caughtException}** として EL で見ることができます。 **#{org.jboss.seam.exception}** 形式ではなくなります。

EntityConverter 設定における変更点

entity-loader コンポーネントから使用されるエンティティマネージャを設定できるようになりました。 詳細についてはドキュメントをご覧ください。

管理 Hibernate セッションにおける変更点

Seam Application Framework など Seam のいくつかの部分は Seam 管理永続コンテキスト (JPA) と Hibernate Session 間での共通命名規則の存在に依存しています。 Seam 2.1 以前のバージョンでは管理 Hibernate Session の名前は **session** になるとみなされていました。**session** は Seam や Java Servlet API で過剰使用されているため、 デフォルトを **hibernateSession** に変更することで曖昧性を軽減しました。 つまり、 Hibernate Session のインジェクトまたはリゾルブを行う場合に適切なセッションの識別が非常に容易になります。

以下のいずれかの方法を使用して Hibernate Session のインジェクトを行うことができます。

```
        @In private Session hibernateSession;
```

```
        @In(name = "hibernateSession") private Session session;
```

Seam 管理の Hibernate Session がまだ **session** という名前の場合は、 **session** プロパティで明示的に参照をインジェクトできます。

```
<framework:hibernate-entity-home session="#{session}".../>
      <transaction:entity-transaction session="#{session}".../>
```

代わりに、 次のように Seam Application Framework のすべての永続コントローラで **getPersistenceContextName()** メソッドを無効化することもできます。

```
        public String getPersistenceContextName() {
        "session";
        }
```

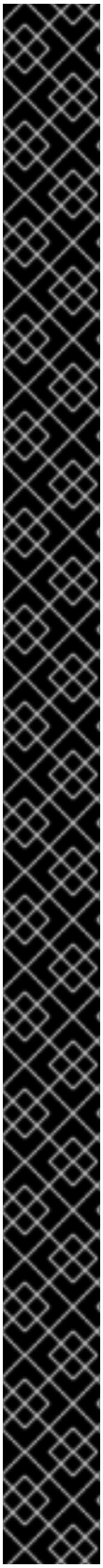
セキュリティにおける変更点

components.xml にあるセキュリティルールに関する構成がルールベースのセキュリティを使用するプロジェクトに対して変更されました。 以前は、ルールは **identity** コンポーネントのプロパティとして設定されていました。

```
<security:identity security-rules="#{securityRules}"
      authenticate-method="#{authenticator.authenticate}"/>
```

Seam 2.1 は **ruleBasedPermissionResolver** コンポーネントをそのルールベースのパーミッションチェックに使用します。このコンポーネントをアクティブにして **identity** コンポーネントの代わりにこのコンポーネントでセキュリティルールを登録する必要があります。

```
<security:rule-based-permission-resolver
      security-rules="#{securityRules}"/>
```



重要

パーミッション の定義が変更されました。 Seam 2.1 以前では、パーミッションは 3 つのエレメントから構成されていました。

- 名前
- 動作
- コンテキスト依存のオブジェクト (オプション)

名前 は一般的には Seam のコンポーネントの名前、 エンティティクラスまたはビュー ID になります。**動作** はメソッド名、 JSF フェーズ (復元またはレンダリング)、 または動作の意図を表す割り当てられた表現になります。オプションで、ひとつまたは複数のコンテキスト依存のオブジェクトをワーキングメモリに直接挿入してデシジョンメイキングに役立てることができます。 一般的にはこれが動作の目的となります。 たとえば、

```
s:hasPermission('userManager', 'edit', user)
```

Seam 2.1 ではパーミッションが簡略化されているため、含むエレメントは 2 つです。

- ターゲット
- 動作

ターゲット は **名前** エレメントを置換してパーミッションの中心となります。**動作** は安全化を図る動作の目的と連携します。ルールファイル内で、 ほとんどのチェックが **ターゲット** オブジェクトを中心とするようになりました。 たとえば、

```
s:hasPermission(user, 'edit')
```

この変更によりルールがさらに幅広く適用できるようになるため、 Seam は永続パーミッションリゾルバ (ACL) の他、 ルールベースのリゾルバとも動作するようになります。

また、 既存のルールが奇異な動作をする場合があるので留意してください。 次のパーミッションチェック形式が原因です。

```
s:hasPermission('userManager', 'edit', user)
```

Seam は次を置き換えて新しいパーミッション形式を適用します。

```
s:hasPemrission(user, 'edit')
```

新しい設計に関する全概要は「セキュリティ」の章をお読みください。

Identity.isLoggedIn() における変更点

このメソッドは資格情報が設定されている場合は認証チェックの試行を行わなくなります。代わりに、ユーザーが現在未認証の場合は **true** を返します。 以前の動作を利用する場合は **`Identity.tryLogin()`** を使用します。

Seam のトークンベースの **Remember-Me** 機能を使用する場合、 次のセクションを **`components.xml`** に追加してアプリケーションが初めてアクセスされたときにユーザーが自動的にログインされるようにします。

```
<event type="org.jboss.seam.security.notLoggedIn">
      <action execute="#{redirect.captureCurrentView}"/>
      <action execute="#{identity.tryLogin}"/>
      </event>
      <event type="org.jboss.seam.security.loginSuccessful">
      <action execute="#{redirect.returnToCapturedView}"/>
      </event>
```

iText (PDF) における変更点

documentStore コンポーネントは外部の **pdf/itext** モジュールから Seam 自体に移動されました。そのため **components.xml** にある **pdf:document-store** への参照はすべて **document:document-store** で置換されるはずです。 同様に、 **web.xml** が **org.jboss.seam.pdf.DocumentStoreServlet** を参照する場合には、その参照を **org.jboss.seam.document.DocumentStoreServlet** に変更してください。

クラスタリングにおける変更点

Seam の **ManagedEntityInterceptor** (以前は **ManagedEntityIdentityInterceptor**) はデフォルトでは無効です。クラスタ対話のフェールオーバーに **ManagedEntityInterceptor** が必要な場合、 次のようにして **components.xml** で有効できます。

```
<core:init>
      <core:interceptors>
      <value>org.jboss.seam.core.SynchronizationInterceptor</value>
      <value>org.jboss.seam.async.AsynchronousInterceptor</value>
      <value>org.jboss.seam.ejb.RemoveInterceptor</value>

<value>org.jboss.seam.persistence.HibernateSessionProxyInterceptor</value>

<value>org.jboss.seam.persistence.EntityManagerProxyInterceptor</value>
      <value>org.jboss.seam.core.MethodContextInterceptor</value>
      <value>org.jboss.seam.core.EventInterceptor</value>
      <value>org.jboss.seam.core.ConversationalInterceptor</value>
      <value>org.jboss.seam.bpm.BusinessProcessInterceptor</value>
      <value>org.jboss.seam.core.ConversationInterceptor</value>
      <value>org.jboss.seam.core.BijectionInterceptor</value>
      <value>org.jboss.seam.transaction.RollbackInterceptor</value>
      <value>org.jboss.seam.transaction.TransactionInterceptor</value>
      <value>org.jboss.seam.webservice.WSSecurityInterceptor</value>
      <value>org.jboss.seam.security.SecurityInterceptor</value>
      <value>org.jboss.seam.persistence.ManagedEntityInterceptor</value>
      </core:interceptors>
      </core:init>
```

非同期の例外処理における変更点

非同期の呼び出しはすべて例外処理でラップされます。 デフォルトでは非同期の呼び出しから伝播する例外はすべてエラーレベルでキャッチされログ記録されます。 詳細は 21章*非同期性とメッセージング*を参照してください。

イベントの再デプロイにおける変更点

org.jboss.seam.postInitialization イベントは再デプロイメント時に呼び出されなくなりました。 代わりに **org.jboss.seam.postReInitialization** が呼び出されます。

キャッシュへの対応における変更点

Seam でのキャッシュへの対応は JBoss Cache 3.2、JBoss Cache 2、Ehcache に対応するよう記述し直されました。詳細は 22章 *キャッシュ* を参照してください。

<s:cache /> には変更はありませんが、**pojoCache** コンポーネントはインジェクトできなくなりました。

CacheProvider はマップのようなインターフェースを提供します。**getDelegate()** メソッドを使って基礎となるキャッシュを取得できます。

Maven 依存性における変更点

提供されているプラットフォームは JBoss AS 5.1.0 であるため、**javaassist:javaassist** と **dom4j:dom4j** は **provided** とマークされています。

Seam Application Framework における変更点

いくつかのプロパティが値式を予期するようになりました。

- **entityHome.createdMessage**
- **entityHome.updatedMessage**
- **entityHome.deletedMessage**
- **entityQuery.restrictions**

これらのオブジェクトを **components.xml** で設定すると変更は必要ありません。オブジェクトを JavaScript で設定する場合は、値式を次のように作成する必要があります。

```
        public ValueExpression getCreatedMessage() {
        return createValueExpression("New person #{person.firstName}
        #{person.lastName} created");
        }
```

第3章 SEAM-GEN を使った SEAM の紹介

Seam には、 Eclipse プロジェクトの設定、シンプルな Seam のスケルトンコード生成、 既存データベースからのアプリケーションのリバースエンジニアリングを容易にするコマンドラインユーティリティが含まれます。これで Seam を簡単に理解できます。

本リリースでは **seam-gen** は JBoss Enterprise Application Platform との併用が最適です。

Eclipse がなくても seam-gen は使用可能ですが、このチュートリアルでは Eclipse で seam-gen を使用してデバッグや統合テストを行う方法を示したいと思います。 Eclipse を使用したくない方も、このチュートリアルを続けることができます。 すべてのステップをコマンドラインから行うことができます。

3.1. 始める前に

このチュートリアルを始める前に、JDK 6 (詳細は 「Java Development Kit (JDK) の依存性」 を参照)、JBoss Enterprise Application Platform 5、Ant 1.7.0、さらには Eclipse の最新版、Eclipse の JBoss IDE プラグイン、Eclipse の TestNG プラグインが正しくインストールされていることを確認してください。Eclipse の JBoss Server View に JBoss インストールを追加します。次に、デバッグモードで JBoss を起動します。最後に、Seam ディストリビューションを解凍したディレクトリでコマンドプロンプトを起動します。

JBoss は **WAR** と **EAR** のホット再デプロイメントに対して高度なサポートを提供します。残念ながら、JVM のバグのため EAR の再デプロイメントを繰り返すと (開発段階では一般的)、JVM の perm gen 領域を使い果たすことになります。このため、開発段階では大規模な perm gen 領域を持つ JVM で JBoss を稼動させることが推奨されます。 JBoss IDE から JBoss を稼動させる場合は、「VM arguments」の下にあるサーバ起動設定でこれを設定することができます。以下の値が推奨されます。

```
-Xms512m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=512m
```

以下が推奨される最小値です。

```
-Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=256m
```

コマンドラインから JBoss を実行している場合は **bin/run.conf** の JVM オプションを設定することが可能です。

3.2. 新しいプロジェクトの設定

まず、 使用環境にあわせて seam-gen を設定します。コマンドラインインターフェースで次を入力します。**cd *seam_distribution_dir*; seam setup**

以下のように必要な情報の入力が求められます。

```
Buildfile: build.xml

init:

setup:

    [echo] Welcome to seam-gen 2.2.2.EAP5 :-)

    [echo] Answer each question or hit ENTER to accept the default (in
```

```
brackets)

    [echo]

   [input] Enter the directory where you want the project to be created
(should not contain spaces) [/home/mnovotny/projects]
[/home/mnovotny/projects]



   [input] Enter your JBoss AS home directory [/var/lib/jbossas]
[/var/lib/jbossas]

/home/mnovotny/apps/jboss-eap-5.1/jboss-as

   [input] Enter your JBoss AS domain [default] [default]



   [input] Enter the project name [myproject] [myproject]

helloworld

    [echo] Accepted project name as: helloworld

   [input] Select a RichFaces skin [glassX] (blueSky, classic, darkX,
deepMarine, DEFAULT, emeraldTown, [glassX], japanCherry, laguna, ruby,
wine)



   [input] Is this project deployed as an EAR (with EJB components) or a
WAR (with no EJB support)? [war] (ear, [war])

ear

   [input] Enter the base package name for your Java classes
[com.mydomain.helloworld] [com.mydomain.helloworld]



   [input] Enter the Java package name for your session beans
[com.mydomain.helloworld.action] [com.mydomain.helloworld.action]



   [input] Enter the Java package name for your entity beans
[com.mydomain.helloworld.model] [com.mydomain.helloworld.model]



   [input] Enter the Java package name for your test cases
[com.mydomain.helloworld.test] [com.mydomain.helloworld.test]
```

```
   [input] What kind of database are you using? [hsql] ([hsql], mysql,
derby, oracle, postgres, mssql, db2, sybase, enterprisedb, h2)

mysql

   [input] Enter the filesystem path to the JDBC driver jar [] []

/usr/share/java/mysql.jar

   [input] skipping input as property driver.license.jar.new has already
been set.

   [input] Enter the Hibernate dialect for your database
[org.hibernate.dialect.MySQLDialect] [org.hibernate.dialect.MySQLDialect]



   [input] Enter the JDBC driver class for your database
[com.mysql.jdbc.Driver] [com.mysql.jdbc.Driver]



   [input] Enter the JDBC DataSource class for your database
[com.mysql.jdbc.jdbc2.optional.MysqlDataSource]
[com.mysql.jdbc.jdbc2.optional.MysqlDataSource]



   [input] Enter the JDBC URL for your database [jdbc:mysql:///test]
[jdbc:mysql:///test]



   [input] Enter the database username [sa] [sa]

root

   [input] Enter the database password [] []



   [input] skipping input as property hibernate.default_schema.entered has
already been set.

   [input] Enter the database catalog name (Enter '-' to clear previous
value) [] []



   [input] Are you working with tables that already exist in the database?
[n] (y, [n])

y

   [input] Do you want to recreate the database tables and execute
import.sql each time you deploy? [n] (y, [n])
```

```
[propertyfile] Creating new property file: /home/mnovotny/apps/jboss-eap-
5.1/seam/seam-gen/build.properties

    [echo] Installing JDBC driver jar to JBoss AS

    [copy] Copying 1 file to /home/mnovotny/apps/jboss-eap-5.1/jboss-
as/server/default/lib



init:



init-properties:

    [echo] /home/mnovotny/apps/jboss-eap-5.1/jboss-as



validate-workspace:



validate-project:



settings:

    [echo] JBoss AS home: /home/mnovotny/apps/jboss-eap-5.1/jboss-as

    [echo] Project name: helloworld

    [echo] Project location: /home/mnovotny/projects/helloworld

    [echo] Project type: ear

    [echo] Action package: com.mydomain.helloworld.action

    [echo] Model package: com.mydomain.helloworld.model

    [echo] Test package: com.mydomain.helloworld.test

    [echo] JDBC driver class: com.mysql.jdbc.Driver

    [echo] JDBC DataSource class:
com.mysql.jdbc.jdbc2.optional.MysqlDataSource

    [echo] Hibernate dialect: org.hibernate.dialect.MySQLDialect

    [echo] JDBC URL: jdbc:mysql:///test
```

```
    [echo] Database username: root

    [echo] Database password:

    [echo]

    [echo] Type './seam create-project' to create the new project
```

ツールが適当なデフォルト値を選択します。選択されたデフォルト値で問題なければ、そのまま **Enter** を押します。

ここでの最も重要な選択は、プロジェクトのデプロイを **EAR** または **WAR** アーカイブとして行うかどうかということです。**EAR** プロジェクトは Enterprise JavaBeans 3.0 (EJB3) に対応しており、Java EE 5 が必要です。**WAR** プロジェクトは EJB3 に対応していませんが、J2EE 環境にデプロイ可能です。WAR は EAR に比べシンプルなパッケージです。JBoss のような EJB3 に対応済みのアプリケーションサーバーがインストールされている場合は **ear** を選択してください。これ以外は**war** を選択してください。以降、このチュートリアルでは **EAR** デプロイメントを使用していると仮定しますが、プロジェクトが **WAR** デプロイメントの場合もこのステップに沿って進めることができます。

既存のデータモデルで作業をしている場合は、既にテーブルが存在していることを必ずデータベースに seam-gen に知らせてください。

seam new-project を入力して、Eclipse ワークスペースディレクトリに新しいプロジェクトを作成します。

```
Buildfile: build.xml
...
new-project:
  [echo] A new Seam project named 'helloworld' was created in the
C:\Projects directory
  [echo] Type 'seam explode' and go to http://localhost:8080/helloworld
  [echo] Eclipse Users: Add the project into Eclipse using File > New >
Project and select General > Project (not Java Project)
  [echo] NetBeans Users: Open the project in NetBeans
BUILD SUCCESSFUL Total time: 7 seconds
C:\Projects\jboss-seam>
```

これにより Seam の **JAR** 郡、依存する **JAR** 郡、 JDBC ドライバの **JAR** 郡を新しい Eclipse プロジェクトにコピーします。 必要となるすべてのリソースや設定ファイル、 Facelets テンプレートファイルとスタイルシートの他、 Eclipse メタデータと Ant ビルドスクリプトを生成します。 プロジェクトを追加すると直ちに Eclipse プロジェクトは JBoss 内の展開されたディレクトリ構造に自動的にデプロイされます。 プロジェクトを追加するには、 **New** → **Project...** → **General** → **Project** → **Next** の順で進み、 **Project name** (この場合は **helloworld**) を入力し **Finish** をクリックします。New Project のウィザードで **Java Project** は選択しないようにしてください。

Eclipse のデフォルト JDK が Java SE 6 JDK ではない場合は、 準拠型の JDK を選択する必要があります。**Project** → **Properties** → **Java Compiler** の順に選択します。

別の方法として、 **seam explode** と入力し Eclise の外部からプロジェクトをデプロイすることもできます。

ウェルカムページは **http://localhost:8080/helloworld** にあります。これは **view/layout/template.xhtml** にあるテンプレートを使って作成された Facelets のページ (**view/home.xhtml**) です。 Eclipse からこのページやテンプレートの編集が可能です。 ブラウザを更新すると結果をすぐに見ることもできます。

プロジェクトディレクトリに XML 設定ドキュメントが生成されます。 これらのドキュメントは複雑に見えるかもしれませんが、 主に標準 Java EE で構成されるため複数の Seam プロジェクト間であっても変更を行う必要性はほとんどありません。

生成されたプロジェクトには 3 種類のデータベースと永続性設定があります。 HSQLDB に対して TestNG ユニットテストを実行するときに **persistence-test.xml** と **import-test.sql** ファイルが使用されます。 **import-test.sql** のデータベーススキーマとテストデータは常にテストが実行される前にデータベースにエキスポートされます。 **myproject-dev-ds.xml**、 **persistence-dev.xml**、 **import-dev.sql** はアプリケーションを開発データベースにデプロイするときに使用します。 seam-gen に既存データベースで作業していることを認識させている場合は、 デプロイ時にスキーマが自動的にエクスポートされる場合があります。 **myproject-prod-ds.xml**、 **persistence-prod.xml** と **import-prod.sql** は実稼働用のデータベースにアプリケーションをデプロイするときに使用します。 スキーマはデプロイ時に自動的にはエクスポートされません。

3.3. 新しいアクションの作成

seam new-action を入力するとステートレスなアクションメソッドを持つシンプルな Web ページを作成することができます。

Seam が情報の入力を促したあと、 プロジェクト用の新しい Facelets ページと Seam コンポーネントが生成されます。

```
Buildfile: build.xml

validate-workspace:

validate-project:

action-input:
    [input] Enter the Seam component name
ping
    [input] Enter the local interface name [Ping]

    [input] Enter the bean class name [PingBean]

    [input] Enter the action method name [ping]

    [input] Enter the page name [ping]


setup-filters:

new-action:
     [echo] Creating a new stateless session bean component with an action
method
     [copy] Copying 1 file to
C:\Projects\helloworld\src\hot\org\jboss\helloworld
     [copy] Copying 1 file to
C:\Projects\helloworld\src\hot\org\jboss\helloworld
     [copy] Copying 1 file to
C:\Projects\helloworld\src\hot\org\jboss\helloworld\test
     [copy] Copying 1 file to
C:\Projects\helloworld\src\hot\org\jboss\helloworld\test
     [copy] Copying 1 file to C:\Projects\helloworld\view
     [echo] Type 'seam restart' and go to
```

```
http://localhost:8080/helloworld/ping.seam

BUILD SUCCESSFUL
Total time: 13 seconds
C:\Projects\jboss-seam>
```

新しい Seam コンポーネントを追加したので、 展開したディレクトリのデプロイメントを再実行する必要があります。 再起動するには、 **seam restart** と入力するか、 Eclipse 内から生成されたプロジェクトの **build.xml** ファイルの **restart** ターゲットを実行します。別の方法として、 Eclipse の **resources/META-INF/application.xml** ファイルを編集することでも可能です。

注記

アプリケーションを変更するたびに JBoss を再起動する必要はありません。

ここで **http://localhost:8080/helloworld/ping.seam** に移動し、ボタンをクリックします。プロジェクトの **src** ディレクトリを見れば、 このアクションの背後のコードを見ることができます。 **ping()** メソッドにブレークポイントを追加し、 再度ボタンをクリックします。

最後に、 テストパッケージ内の **PingTest.xml** ファイルを探し Eclipse 用の TestNG プラグインを使用して統合テストを実行します。 また、 **seam test** や生成されたビルドの **test** ターゲットを使ってテストを実行することも可能です。

3.4. アクションのあるフォームの作成

次のステップはフォームの作成です。**seam new-form** を入力します。

```
Buildfile: C:\Projects\jboss-seam\seam-gen\build.xml

validate-workspace:

validate-project:

action-input:
    [input] Enter the Seam component name
hello
    [input] Enter the local interface name [Hello]

    [input] Enter the bean class name [HelloBean]

    [input] Enter the action method name [hello]

    [input] Enter the page name [hello]


setup-filters:

new-form:
     [echo] Creating a new stateful session bean component with an action
method
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello\test
     [copy] Copying 1 file to C:\Projects\hello\view
```

```
     [copy] Copying 1 file to C:\Projects\hello\src\hot\com\hello\test
     [echo] Type 'seam restart' and go to
http://localhost:8080/hello/hello.seam

BUILD SUCCESSFUL
Total time: 5 seconds
C:\Projects\jboss-seam>
```

再びアプリケーションを再起動させ、**http://localhost:8080/helloworld/hello.seam** に移動します。 生成されたコードを見てみましょう。 テストを実行します。フォームと Seam コンポーネントに新しいフィールドを追加してみてください (Java コードを変更したら常にデプロイメントを再実行することを忘れないようにしてください)。

3.5. 既存のデータベースからのアプリケーションの生成

手動でデータベースの中にテーブルを作成します (別のデータベースに切り替えるには再度 **seam setup** を実行します)。ここで **seam generate-entities** を入力します。

デプロイメントを再実行して、**http://localhost:8080/helloworld** に移動します。 データベースの閲覧、 既存オブジェクトの編集、 新しいオブジェクトの作成が可能です。 生成されたコードは非常にシンプルです。Seam は seam-gen がなくても手作業でデータアクセスコードを簡単に書けるよう設計されているからです。

3.6. 既存の JPA/EJB3 エンティティからのアプリケーションの生成

既存の有効なエンティティクラスを **src/main** ディレクトリの内側に配置します。ここで**seam generate-ui** を入力します。

デプロイメントを再実行して、**http://localhost:8080/helloworld** に移動します。

3.7. EAR 形式でのアプリケーションのデプロイ

標準 Java EE 5 パッケージを使用してアプリケーションをデプロイするには変更が数点必要です。まず、 **seam unexplode** を実行して展開したディレクトリを削除します。 EAR をデプロイするには、コマンドプロンプトで **seam deploy** を入力するか、 生成されたプロジェクトのビルドスクリプトの **deploy** ターゲットを実行します。 デプロイ解除するには、**seam undeploy** または **undeploy** ターゲットを使用します。

デフォルトでは、アプリケーションは **dev profile** でデプロイします。EAR は **persistence-dev.xml** ファイルと **import-dev.sql** ファイルを含み、**myproject-dev-ds.xml** をデプロイします。**seam -Dprofile=prod deploy** を入力して、プロファイルを **prod profile** に変更できます。

アプリケーション用に新しいデプロイメントプロファイルを定義することもできます。 プロジェクトに適切な名前が付いたファイルを追加するだけで可能です。 例えば、 **persistence-staging.xml**、 **import-staging.sql**、 **myproject-staging-ds.xml** です。 **-Dprofile=staging** を使ってプロファイルの名前を選択します。

3.8. SEAM と増分ホットデプロイメント

展開したディレクトリとして Seam アプリケーションをデプロイする場合、開発段階で増分ホットデプロイメントに対するサポートが含まれます。次の行を **components.xml** に追加して Seam と Facelets でデバッグモードを有効にします。

```
<core:init debug="true">
```

警告

ホットデプロイメントスキャナーがそのサーバープロファイルに対して有効になって以内場合、Facelet のホットデプロイメントは機能しません。

これで Web アプリケーションを完全に再起動する必要なく次のファイルが再デプロイできます。

- Facelets のすべてのページ
- すべての **pages.xml** ファイル

ただし、いずれかの Java コードを変更したい場合はアプリケーションを完全に再起動する必要があります。JBoss ではトップレベルのデプロイメント記述子を修正することでこれを行えます。EAR デプロイメントでは **application.xml**、WAR デプロイメントでは**web.xml** です。

Seam は高速の編集 / コンパイル / テストのサイクルを目的とした JavaBean コンポーネントの増分再デプロイメントをサポートしています。この機能を使用するには、JavaBean コンポーネントを **WEB-INF/dev** ディレクトリにデプロイする必要があります。これで、JavaBean コンポーネントは WAR あるいは EAR クラスローダの代わりに特別な Seam クラスローダによってロードされます。

この機能には以下のような制限があります。

- コンポーネントは JavaBean コンポーネントでなければなりません。 EJB3 Bean コンポーネントは使用できません (この制約は修正中です)。
- エンティティはホットデプロイされることはありません。
- **components.xml** でデプロイされたコンポーネントはホットデプロイできない場合があります。
- ホットデプロイ可能なコンポーネントは、**WEB-INF/dev** の外部にデプロイされたクラスからは見えません。
- Seam デバッグモードを有効にして **jboss-seam-debug.jar** を **WEB-INF/lib** に配置する必要があります。
- **web.xml** に Seam フィルタをインストールしなければなりません。
- システムに負荷がかかりデバッグが有効な場合は、エラーが表示されることがあります。

seam-gen で作成した WAR プロジェクトでは、 **src/hot** ソースディレクトリにあるクラスに対してはそのまま増分ホットデプロイメントが使用可能です。 ただし、 seam-gen は EAR プロジェクトの増分ホットデプロイには対応していません。

第4章 JBOSS DEVELOPER STUDIO を使って SEAM を始めよう

JBoss Developer Studio は、Seam 用のプロジェクト作成ウィザード、Facelets および Java の Unified Expression Language (EL) 用の content assistant、jPDL 用のグラフィカルエディタ、Seam 設定ファイル用のグラフィカルエディタ、Eclipse 内からの Seam 統合テストの実行サポートなど Eclipse プラグインを集めたものです。詳細は 『 JBoss Developer Studio Getting Started Guide 』 を参照してください。

4.1. 始める前に

始める前に、JDK 6、JBoss Enterprise Application Platform 5、Eclipse 3.5、JBoss Developer Studio のプラグイン (少なくとも Seam Tool、Visual Page Editor、jBPM Tools、JBoss AS Tool が必要)、または JBoss Developer Studio と Eclipse 用の TestNG プラグインが正しくインストールされていることを確認してください。

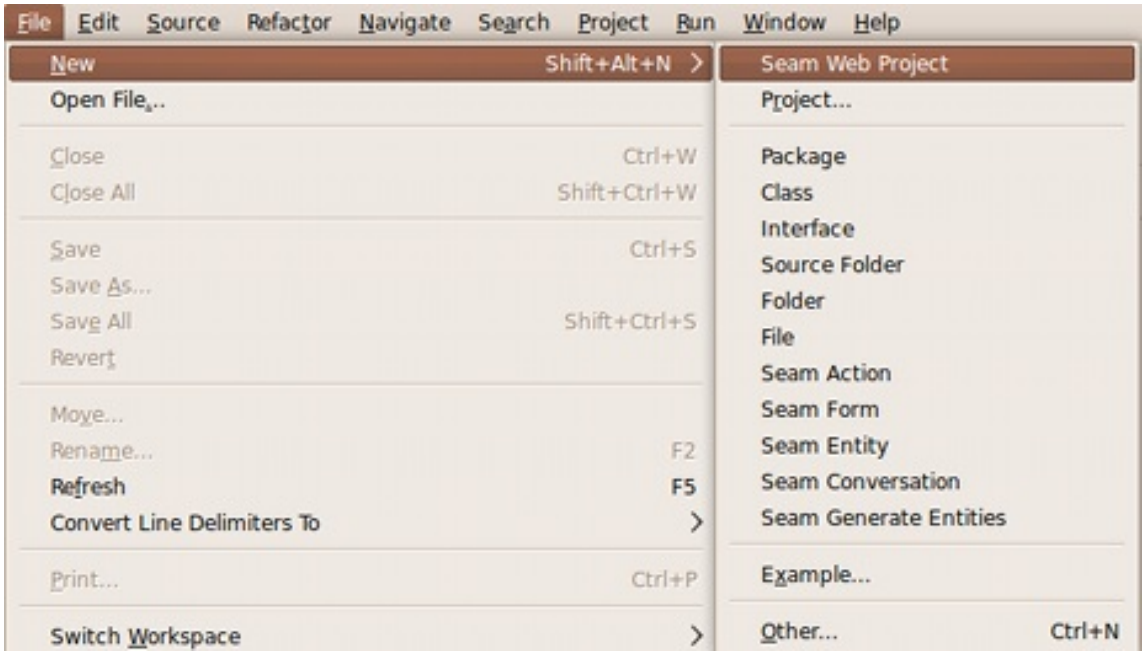
JBoss Developer Studio を使用している場合は、『 JBDS Update Guide 』 の JBDS 文書をお読みください。

JBoss Developer Studio を設定する最も迅速な方法は、 here 文書をご覧ください。

4.2. 新しい SEAM プロジェクトの設定

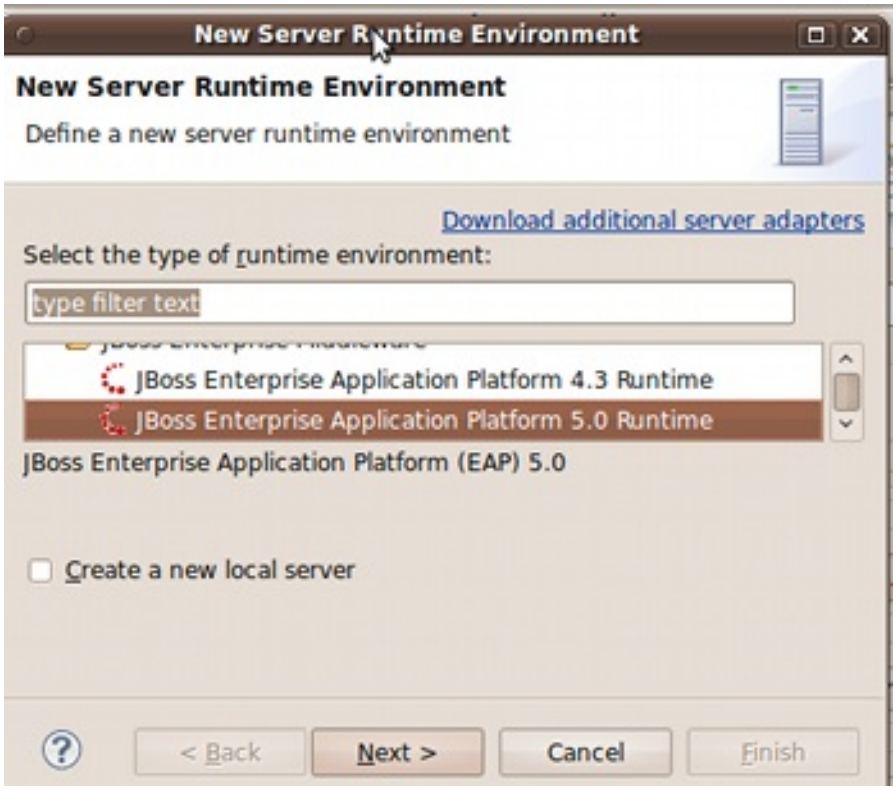
Eclipse を起動して **Seam** パースペクティブを選択してください。

File → **New** → **Seam Web Project** の順で選択します。

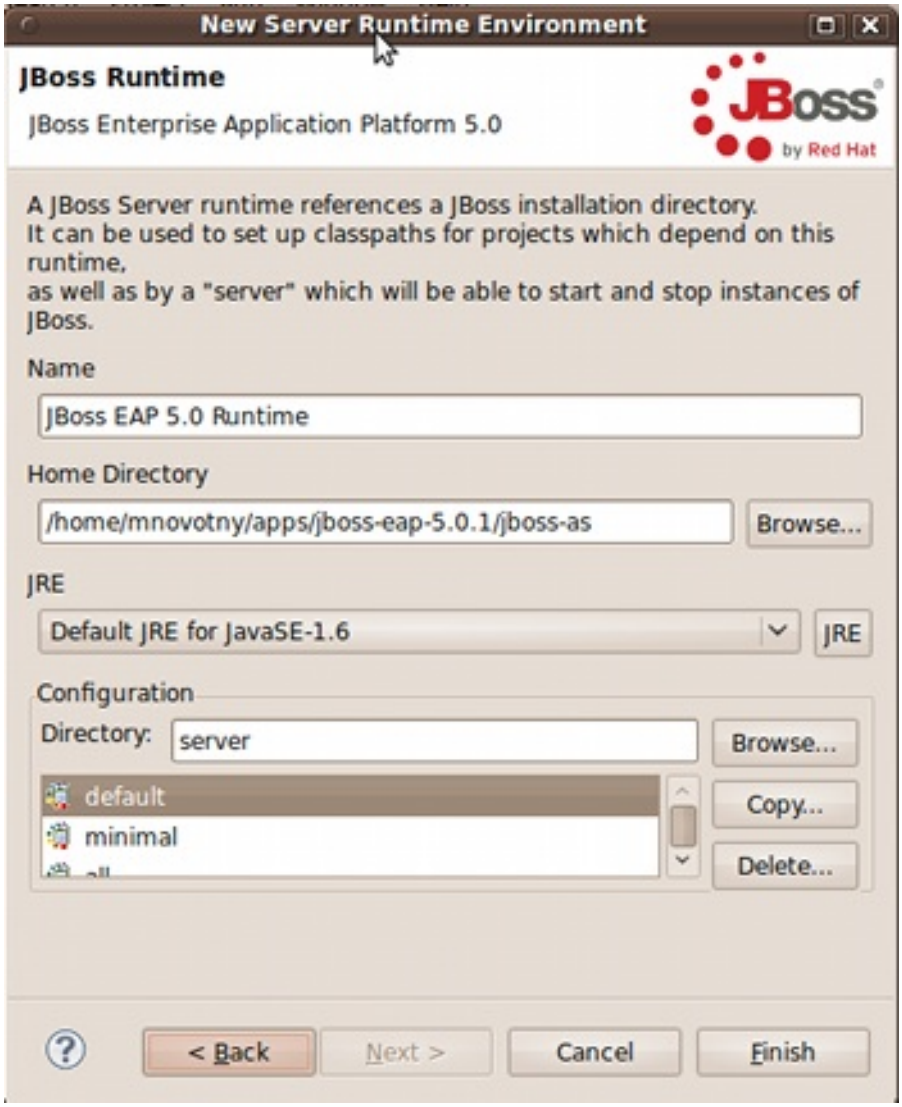


最初に新しいプロジェクト名を入力します。 このチュートリアルではプロジェクト名に **`helloworld`** を使用します。

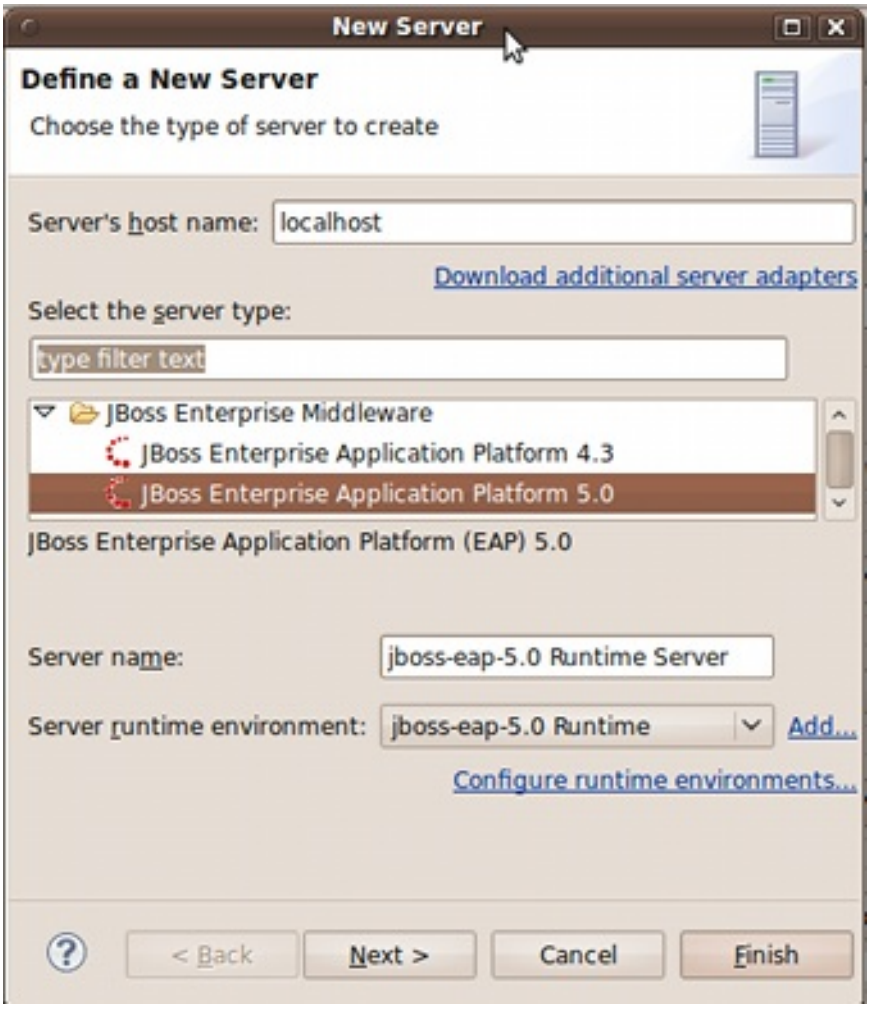
次に、JBoss Developer Studio に JBoss Enterprise Application Platform ランタイムを伝える必要があります。この例では、JBoss Enterprise Application Platform 5 を使用しています。JBoss Enterprise Application Platform を選択するプロセスは 2 つあります。最初にランタイム環境を決めます。この場合、JBDS から事前定義した JBoss Enterprise Application Platform を選択することになります。



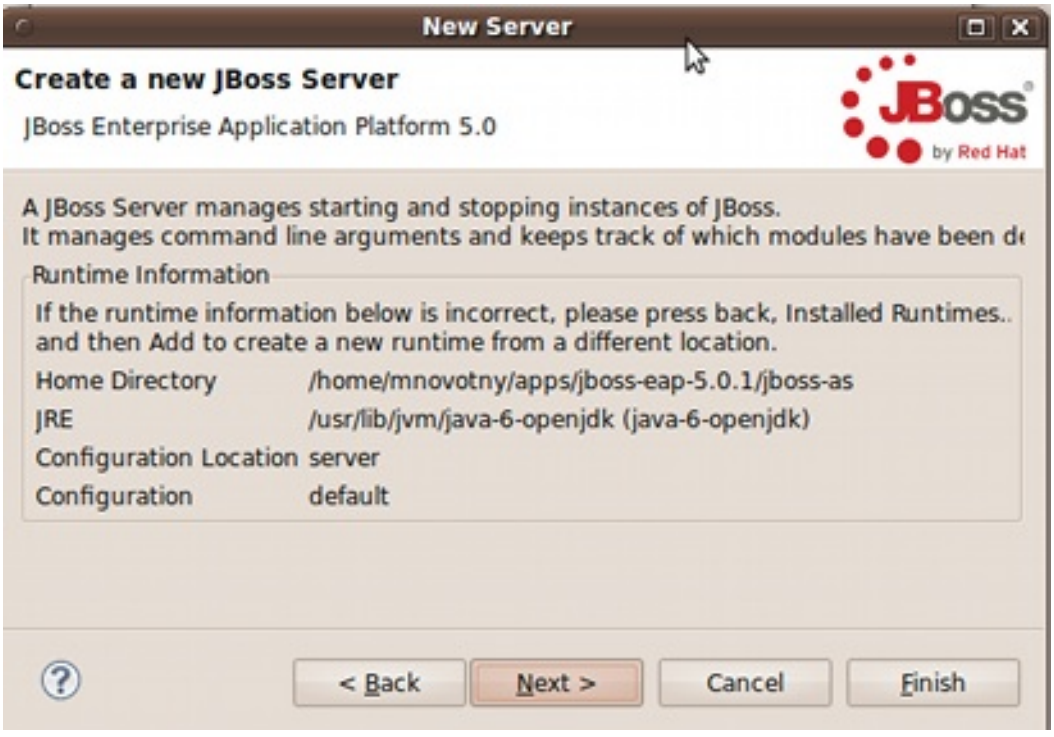
ランタイムの名前を登録し、ハードドライブ上でそれを検索します。



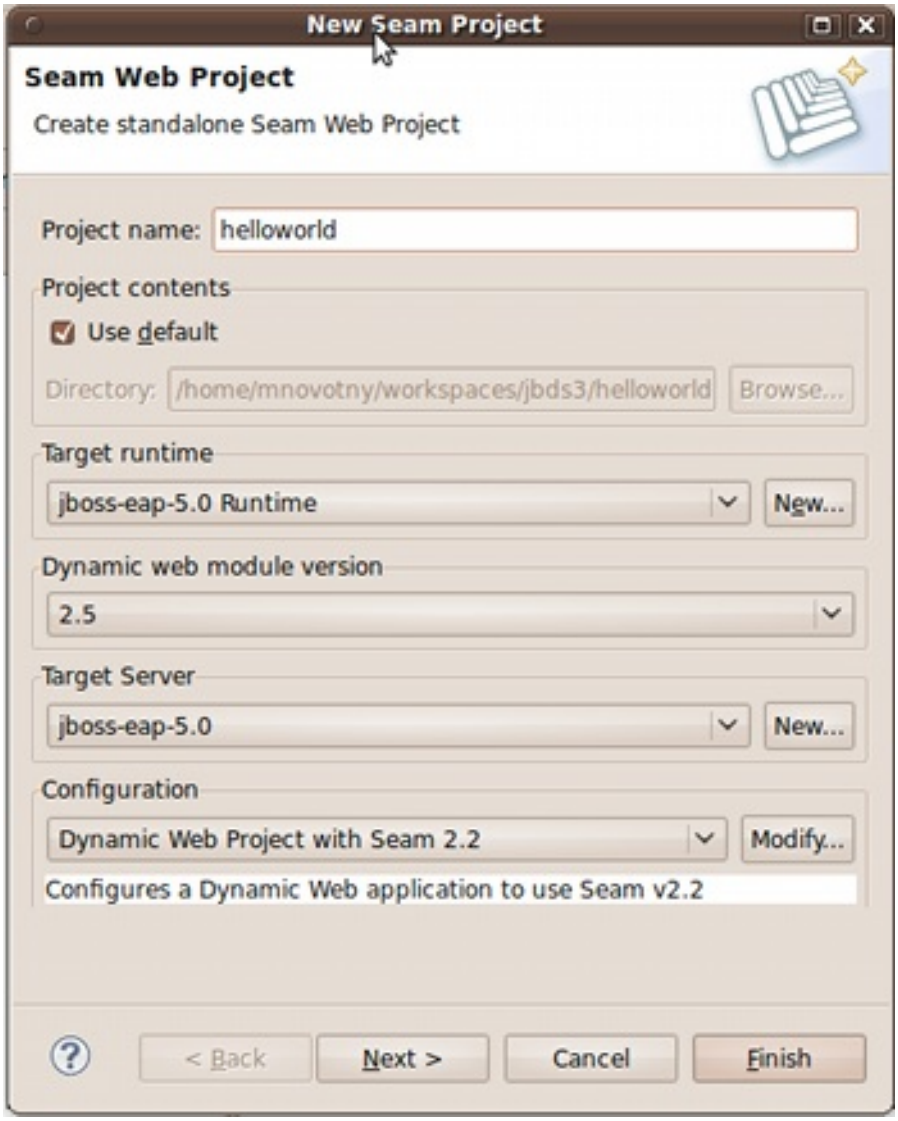
次に JBoss Developer Studio がプロジェクトをデプロイできるサーバーを定義する必要があります。JBoss Enterprise Application Platform 5.0 と 1 つ前のステップで決めたランタイムを選択し、サーバー名を入力して **Next** をクリックします。



次の画面では、サーバーの設定を確認して **Finish** をクリックします。

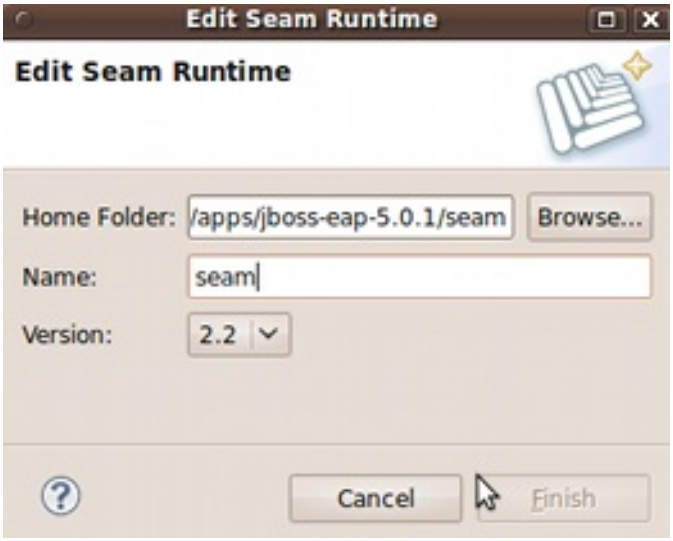


いま作成したランタイムとサーバーが選択されていることを確認します。**Configurations** で **Dynamic Web Project with Seam 2.2** を選択し、**Next** をクリックします。



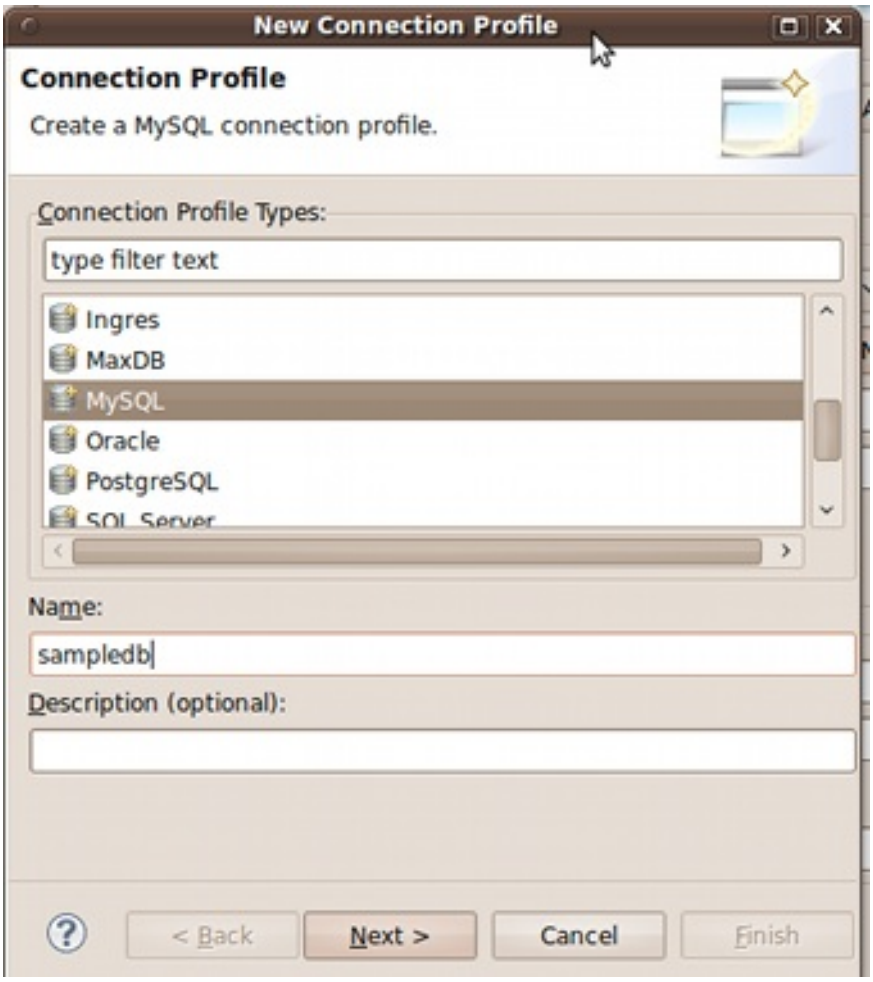
次の 3 画面ではプロジェクトをさらにカスタマイズすることができますが、ここではデフォルトが適切です。最後の画面まで **Next** をクリックして進んでください。

ここでは JBoss Developer Studio にはご使用の Seam ダウンロードに関する情報が必要です。新しい **Seam Runtime** を追加します。その名前を入力し、バージョンとして **2.2** を選択するようにしてください。

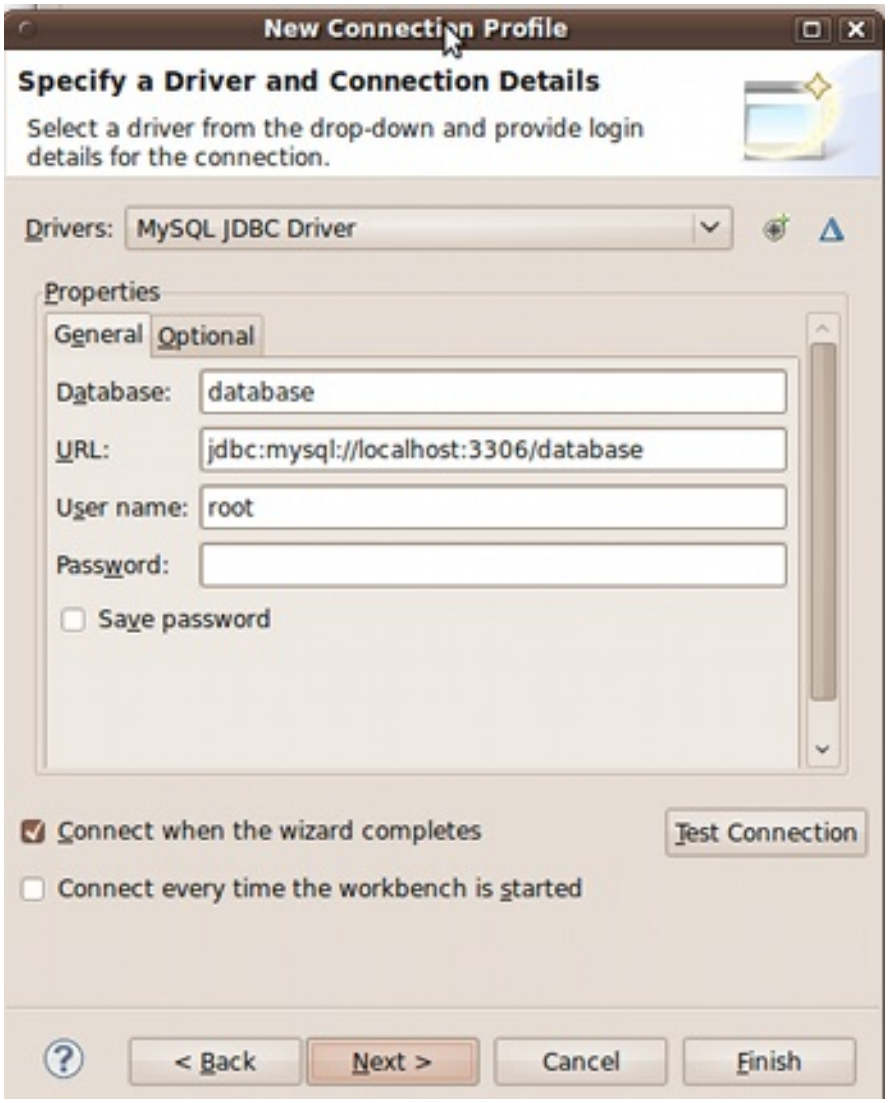


ここでの最も重要な選択は、プロジェクトのデプロイを EAR、WAR のどちらかで行うかということです。EAR プロジェクトは Enterprise JavaBeans 3.0 (EJB3) に対応しており、Java EE 5 が必要です。WAR プロジェクトは EJB3 に対応していませんが、J2E 環境にデプロイ可能です。WAR のパッケージも理解しやすく、このチュートリアルでは WAR デプロイメントを使用していると仮定しています。ただし、プロジェクトが EAR でデプロイされていても、これらのステップで進んでいくことは可能です。Enterprise Application Platform はいずれのデプロイメントのタイプにも対応しています。

次に、使用しているデーターベースのタイプを選択します。このチュートリアルでは、既存のスキーマで MySQL がインストールされていると仮定します。JBoss Developer Studio にデーターベースについて伝え、データーベースとして **MySQL** を選択し、接続プロファイルのタイプ MySQL を選びます。**MySQL** を選択し、名前を入力します

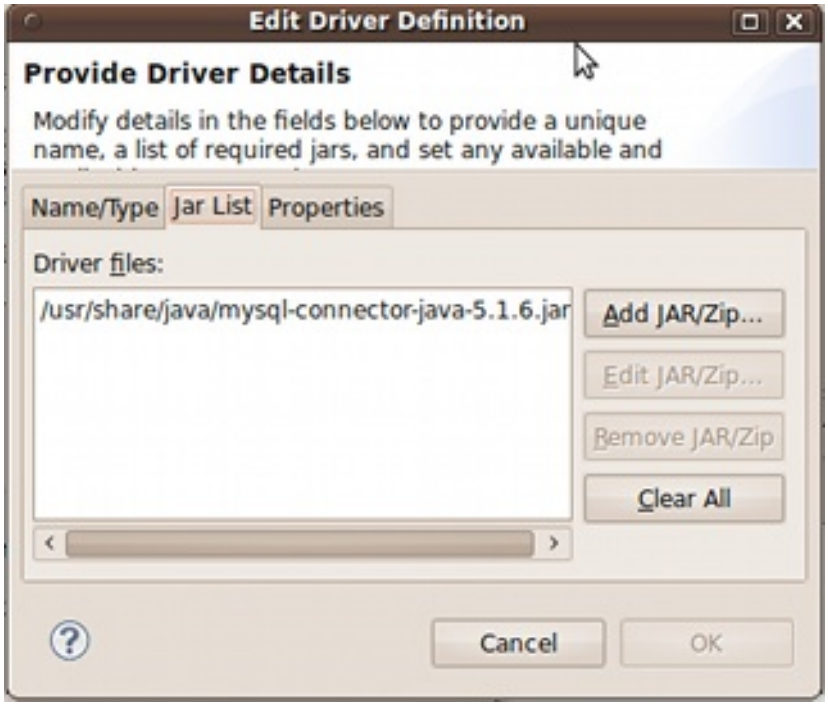


ドロップダウンメニューより既存の MySQL ドライバを選択し、データーベースの名前、JDBC、URL、JDBC ユーザー名、パスワードを指定します。



JBoss Developer Studio にはどのデーターベースにも共通のドライバが付いています。MySQL の場合、JBoss Developer Studio に、ご使用の MySQL JDBC ドライバの場所を伝える必要があります。ドライバの場所を伝えるには、ドロップダウンの一覧のすぐ右にある Δ (デルタ) のアイコンをクリックします。

2 つ目のタブ **jar list** で、MySQL 5 JDBC ドライバの場所を設定する必要があります。次に **Edit JAR/Zip...** をクリックします。

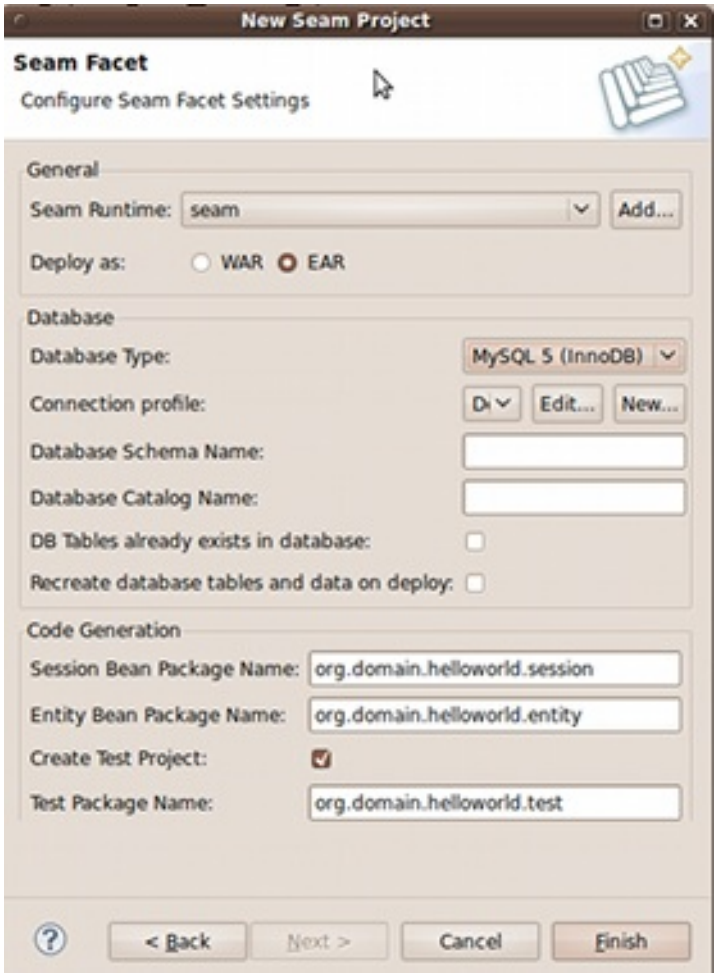


最後に新たに作成されたドライバを選択します。

既存のデータモデルで作業をしている場合は、既にテーブルが存在していることを必ずデータベースに JBoss Developer Studio に知らせてください。

接続に使用するユーザー名とパスワードを確認して、 **Test Connection** ボタンで接続をテストします。テストが正しく動作したら **Finish** をクリックして Seam Project Wizard に戻ります。

最後に、 生成された Bean のパッケージ名を確認して、 問題なければ **Finish** をクリックします。

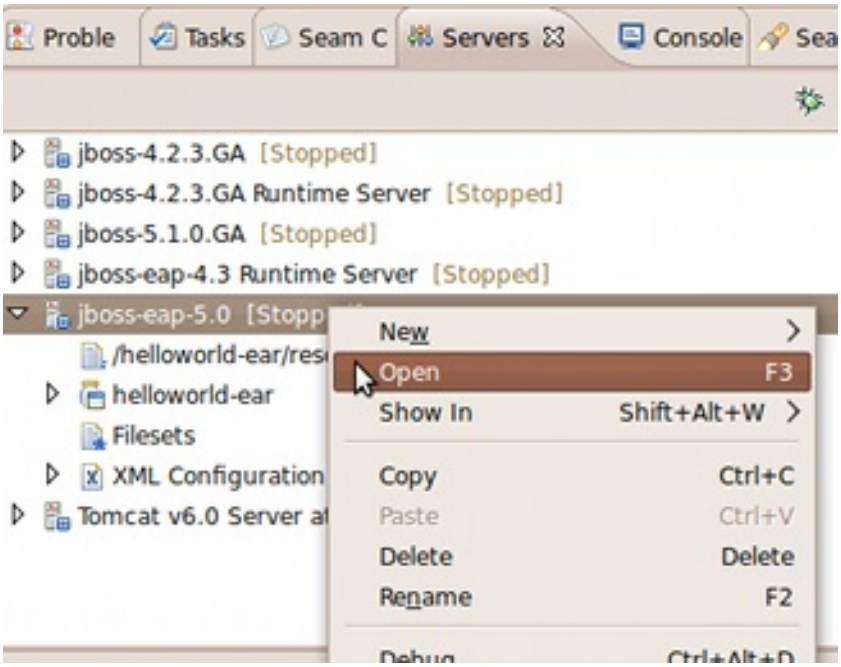


JBoss Developer Studio は WAR と EAR のホット再デプロイメントに対して高度なサポートを提供します。残念ながら、JVM のバグのため EAR の再デプロイメントを繰り返すと (開発段階では一般的です)、最終的には JVM が perm gen (permanent generation) 領域を使い果たすことになります。このため、開発段階では大規模な perm gen 領域を持つ JVM で JBoss Enterprise Application Platform を稼働させることが推奨されます。事前定義した JBoss Enterprise Application Platform 5.0 ランタイムを使用する場合は、最適値は設定されますが、ご使用の環境に適するようカスタマイズすることが可能です。

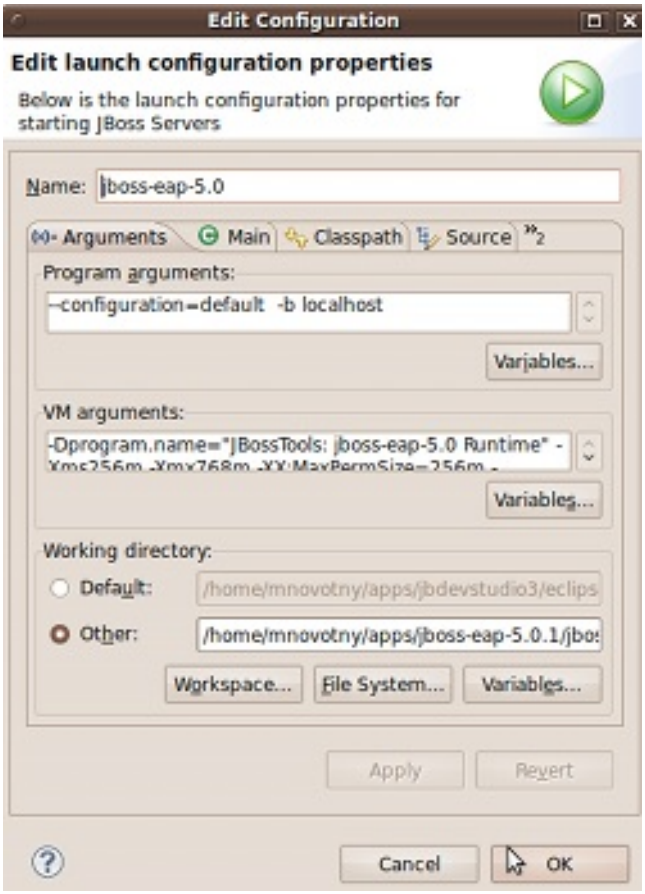
メモリが限られている場合は、以下が推奨される最小値です。

```
-Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=256
```

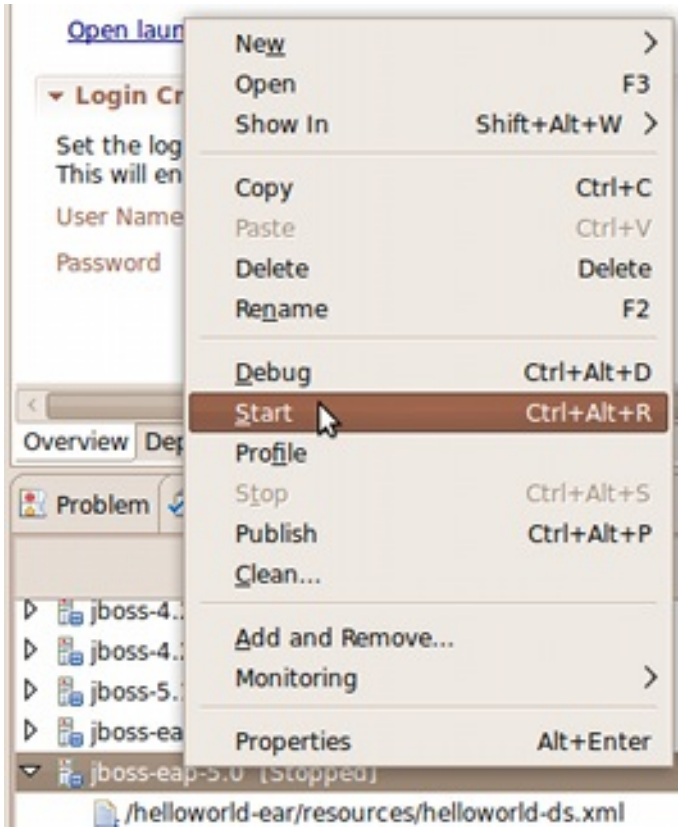
Servers View のサーバーで右クリックし、**Open** を選択します。



次に、開いている Server プロパティで **Open launch configuration** をクリックし VM 引数を変更します。



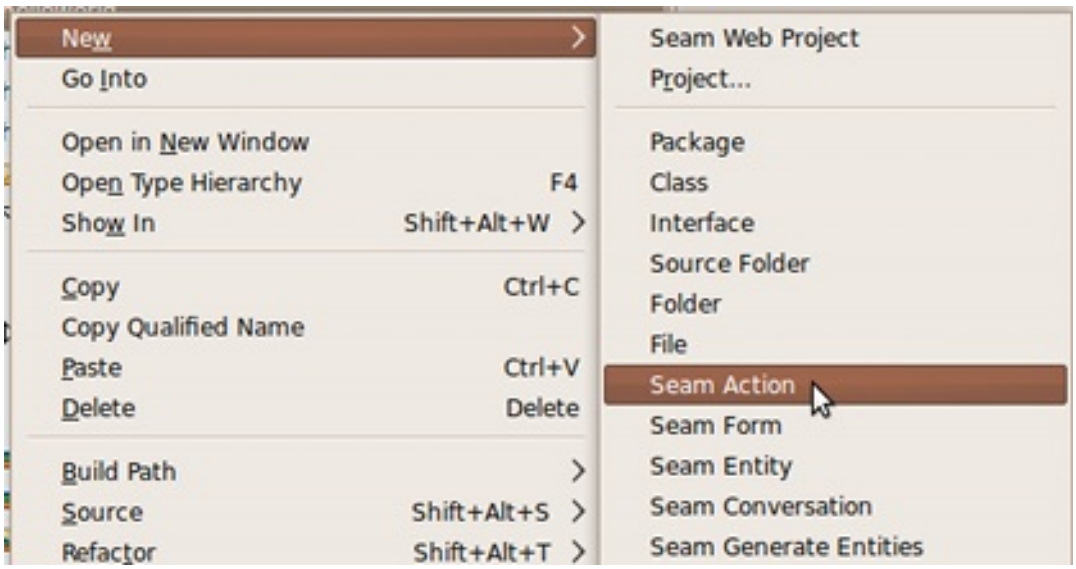
JBoss Enterprise Application Platform を起動してプロジェクトをデプロイするには、作成したサーバーで右クリックし、**Start** をクリックします (またはデバッグモードで開始するには**Debug** をクリックします)。



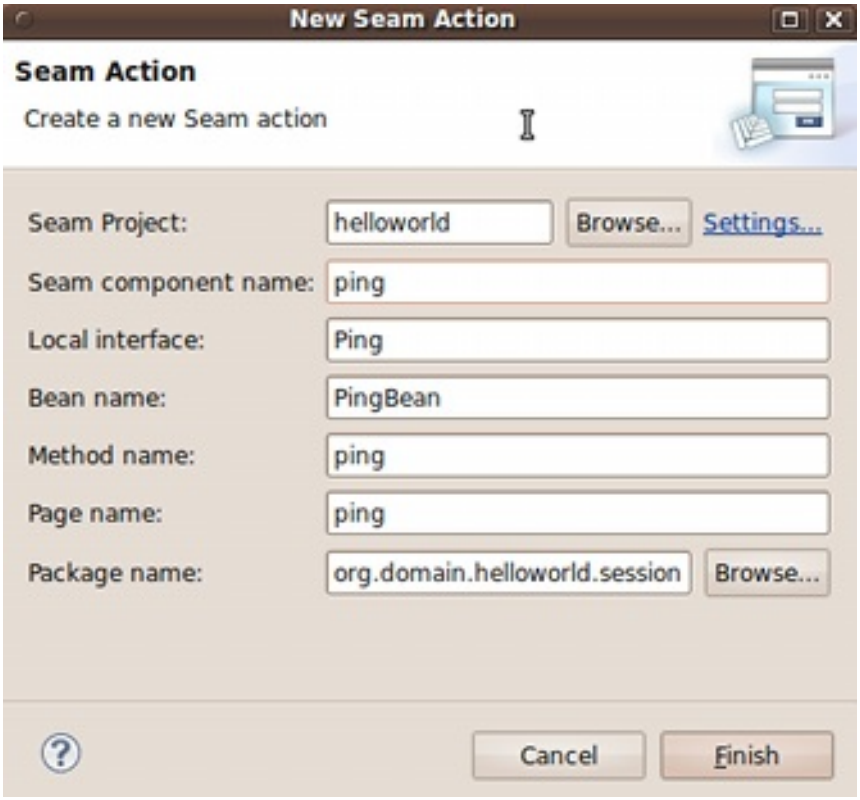
プロジェクトディレクトリに XML 設定ドキュメントが生成されます。 これらのドキュメントは複雑に見えるかもしれませんが、 主に標準 Java EE で構成されるため複数の Seam プロジェクト間であっても変更を行う必要性はほとんどありません。

4.3. 新しいアクションの作成

ステートレスなアクションメソッドを持つシンプルな Web ページを作成するには、**File** → **New** → **Seam Action** の順に選択します。



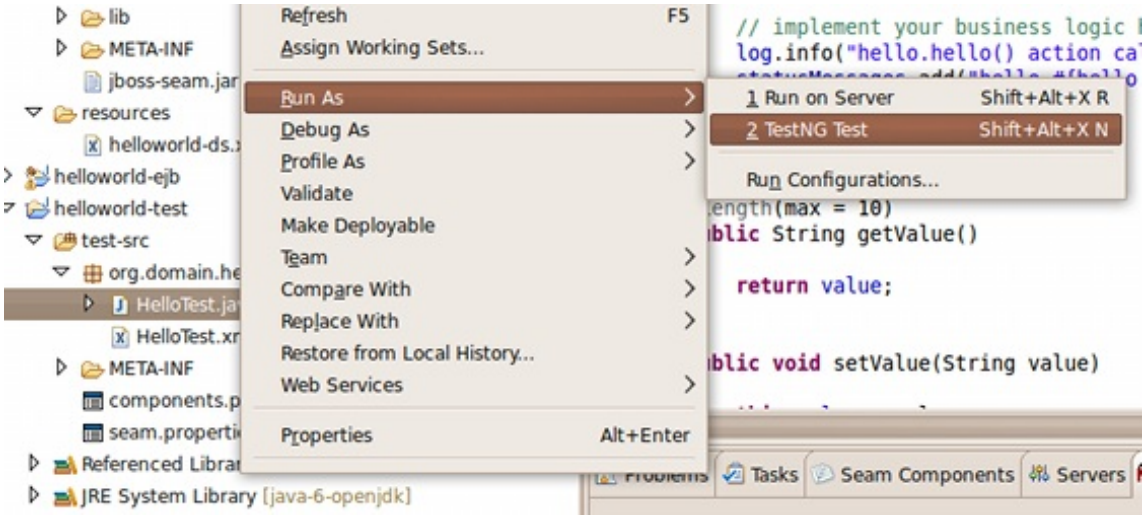
Seam コンポーネントの名前を入力します。他のフィールドには JBoss Developer Studio が適切なデフォルト設定を選択します。



最後に **Finish** を押します。

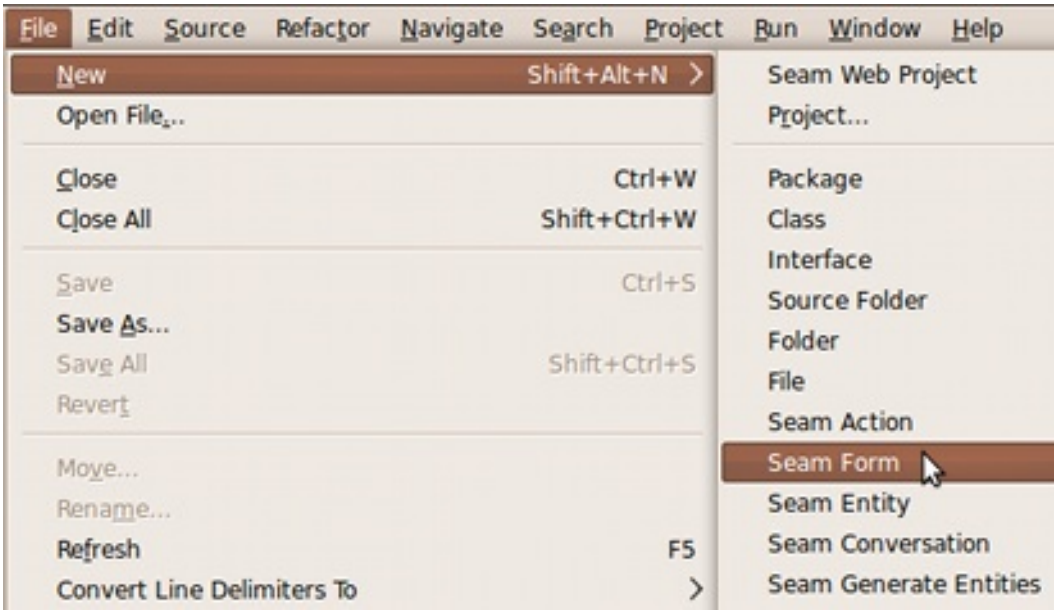
ここで **http://localhost:8080/helloworld/ping.seam** に移動し、ボタンをクリックします。プロジェクトの **src** ディレクトリでこのアクションの背後のコードを見ることができます。**ping()** メソッドにブレークポイントを追加し、再度ボタンをクリックします。

最後に、**helloworld-test** プロジェクトを開き **PingTest** クラスを探し右クリックします。 **Run As -> TestNG Test** の順に選択します。

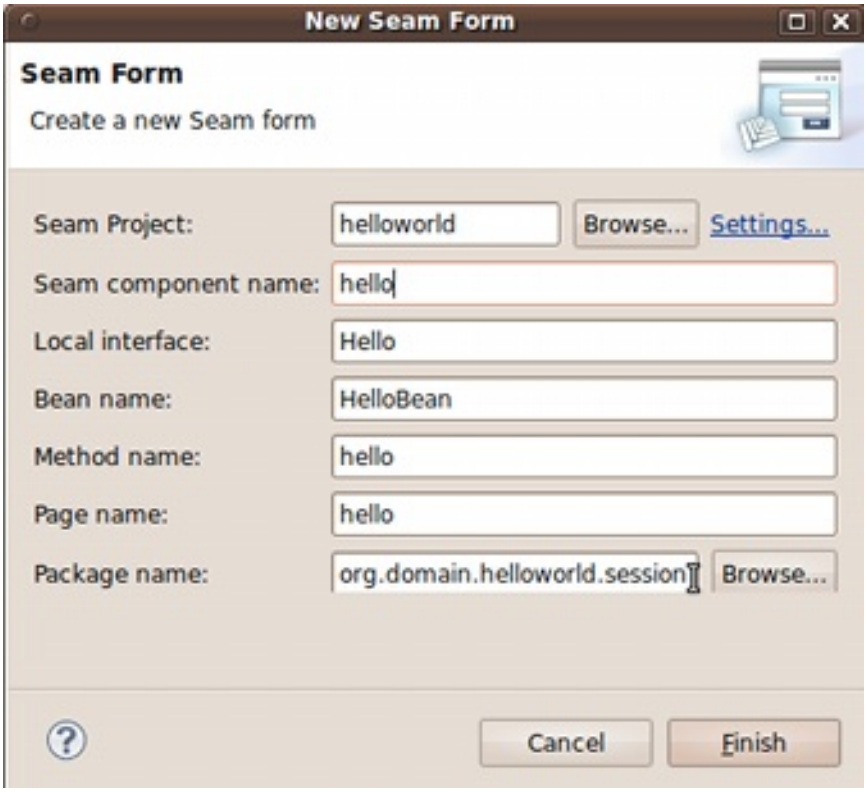


4.4. アクションのあるフォームの作成

最初のステップはフォームの作成です。 **New** → **Seam Form** の順に選択します。



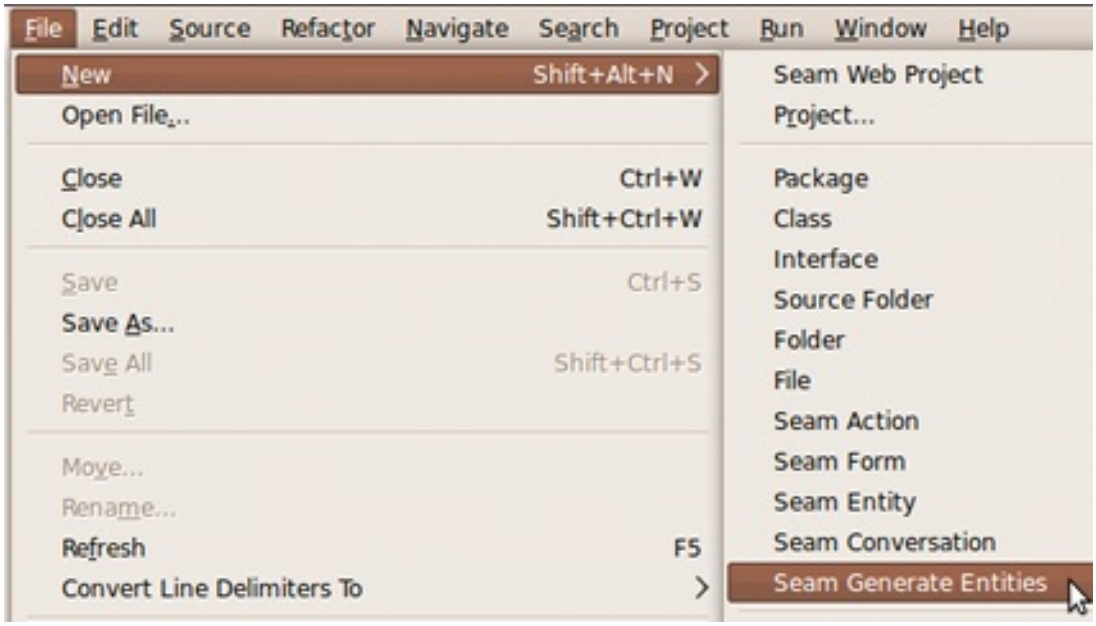
ここで、Seam コンポーネントの名前を入力します。他のフィールドには JBoss Developer Studio が適切なデフォルト設定を選択します。



`http://localhost:8080/helloworld/hello.seam` に移動し、生成されたコードを見てみましょう。テストを実行します。フォームと Seam コンポーネントに新しいフィールドを追加してみてください。Seam がコンポーネントをホットリロードするため、**`src/action`** コードを変更するたびにアプリケーションサーバーを再起動する必要はありません。

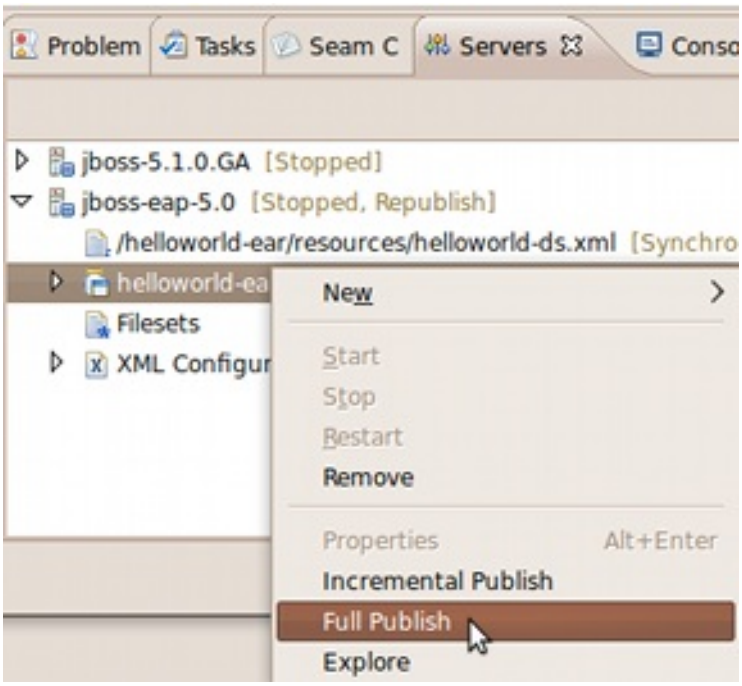
4.5. 既存のデータベースからのアプリケーションの生成

手動でデーターベースにテーブルを作成します (別のデーターベースに切り替えるには新しいプロジェクトを作成してから、その正しいデーターベースを選択します)。次に、**File → New → Seam Generate Entities** の順に選択します。



JBoss Developer Studio では、データーベーススキーマからエンティティ、コンポーネント、ビューのリバースエンジニアリングを行うか、既存の JPA エンティティからコンポーネントとビューのリバースエンジニアリングを行うことができます。このチュートリアルでは、データーベースからのリバースエンジニアリングを行います。

デプロイメントを再実行します。



http://localhost:8080/helloworld に移動します。データベースの閲覧、既存オブジェクトの編集、新規オブジェクトの作成が可能です。生成されたコードは非常にシンプルです。Seam では手作業で簡単にデータアクセスコードが書けるよう設計されています。リバースエンジニアリングも不要です。

4.6. JBOSS DEVELOPER STUDIO を使った SEAM と増分ホットデプロイメント

JBoss Developer Studio は特に設定を行うことなく Facelets のページや **pages.xml** ファイルの増分ホットデプロイメントに対応しています。ただし、Java コードを変更したい場合は **Full Publish** を行うことで、アプリケーションを完全に再起動する必要があります。

Seam は高速の編集 / コンパイル / テストのサイクルを目的とした JavaBean コンポーネントの増分再デプロイメントをサポートしています。この機能を使用するには、JavaBean コンポーネントを **WEB-INF/dev** ディレクトリにデプロイする必要があります。これで、JavaBean コンポーネントは WAR または EAR クラスローダの代わりに特別な Seam クラスローダによってロードされます。

この機能には以下のような制限があります。

- コンポーネントは JavaBean コンポーネントでなければなりません。 EJB3 Bean コンポーネントは使用できません (この制約は修正中です)。
- エンティティはホットデプロイされることはありません。
- **components.xml** でデプロイされたコンポーネントはホットデプロイできない場合があります。
- ホットデプロイ可能なコンポーネントは **WEB-INF/dev** の外部にデプロイされたクラスからは見えません。
- Seam デバッグモードを有効にして **jboss-seam-debug.jar** を **WEB-INF/lib** に配置する必要があります。
- **web.xml** に Seam フィルタをインストールしなければなりません。
- システムに負荷がかかりデバッグが有効な場合は、エラーが表示されることがあります。

JBoss Developer Studio で作成した WAR プロジェクトでは、増分ホットデプロイメントはそのまま使用可能です。ただし、JBoss Developer Studio は EAR プロジェクトの増分ホットデプロイメントには対応していません。

第5章 コンテキスト依存のコンポーネントモデル

Seam における 2 つの中心的概念は、**コンテキスト** と **コンポーネント** です。 コンポーネントはステートフルなオブジェクト、 通常は Enterprise JavaBean (EJB) です。 コンポーネントのインスタンスはコンテキストと関連付けられ、 そのコンテキストで名前が割り当てられます。 **バイジェクション** は、内部のコンポーネント名 (インスタンス変数) をコンテキストの名前にエイリアスできるメカニズムを提供します。これにより、Seam によりコンポーネントツリーの動的な組み立ておよび再組み立てが可能になります。

5.1. SEAM コンテキスト

Seam にはフレームワークによって生成および破棄される組み込みのコンテキストがいくつかあります。 アプリケーションは明示的な Java API 呼び出しによりコンテキスト区分を制御するわけではありません。 通常コンテキストは暗黙的ですが、場合によってはアノテーションで区分されます。

基本的なコンテキストは以下のとおりです。

- ステートレスなコンテキスト
- イベント (例えば 要求) のコンテキスト
- ページのコンテキスト
- 対話のコンテキスト
- セッションのコンテキスト
- ビジネスプロセスのコンテキスト
- アプリケーションのコンテキスト

これらのコンテキストのいくつかは、Servlet や関連する仕様では同じような目的で動作します。 あまり見かけない 2 種類のコンテキストがあります。 **対話コンテキスト** と **ビジネスプロセスコンテキスト** です。 Web アプリケーションで状態管理が非常に脆弱でエラーが発生しやすい理由のひとつは、 3 つの組み込みコンテキスト (要求、 セッション、 アプリケーション) がビジネスロジックに対して特に意味がないためです。 例えば、 アプリケーションのワークフローの観点から見るとユーザーログインセッションは任意の構造です。 このため、 ほとんどの Seam コンポーネントは、 アプリケーションの観点からは最も意味のあるコンテキストとなる対話コンテキストあるいはビジネスプロセスコンテキストにスコープされます。

5.1.1. ステートレスなコンテキスト

本当にステートレスなコンポーネント (主にステートレスセッション Bean) は常にステートレスコンテキストで動作します。 Seam が解決するインスタンスは保存されないためコンテキストがありません。 ステートレスなコンポーネントはオブジェクト指向と言えますが、 定期的に開発されるため Seam アプリケーションの重要な部分を形成します。

5.1.2. イベントのコンテキスト

イベントコンテキストは「最も狭い」ステートフルコンテキストで、 Web 要求の概念を広げて他の種類のイベントも対象とします。最重要なイベントコンテキストの例として、 JSF 要求のライフサイクルと関連付けられたイベントコンテキストがあり、 最もよく利用するコンテキストとなります。イベントコンテキストに関連付けられたコンポーネントは要求終了時に破棄されますが、 その状態は少なくとも要求のライフサイクルの間は使用可能であり明確に定義されます。

RMI または Seam Remoting により Seam コンポーネントを呼び出す場合、 イベントコンテキストはその呼び出しだけのために生成、破棄されます。

5.1.3. ページのコンテキスト

ページコンテキストによりレンダリングされたページの特定のインスタンスと状態を関連付けることができます。 イベントリスナーで状態を初期化する、 またはページレンダリング中にそのページに由来するあらゆるイベントから状態にアクセスが可能です。これは特にサーバー側でのデータ変更で支えられるクリック可能なリストなどの機能に役立ちます。 状態は実際にはクライアントに対してシリアライズされるため、 複数ウィンドウの操作や戻るボタンなどに関して非常に堅固な構造になります。

5.1.4. 対話のコンテキスト

対話コンテキストは Seam で中心となるコンセプトです。**対話** は、ユーザーの観点から見た作業単位です。実際には、複数の要求やデータトランザクションなどユーザーとの複数のやりとりに渡るかもしれません。 しかし、 ユーザーにとっては 1 つの対話が 1 つの問題を解決することになります。 例えば、ホテル予約、契約承認、注文作成などはすべて対話です。 対話を 1 つの「ユースケース」の実装として考えるとわかりやすいかもしれませんが、その関係は必ずしもその通りにはなりません。

対話は現在のウィンドウ内でユーザーの現在のタスクと関連付けられた状態を保持します。1 ユーザーは通常複数のウィンドウにまたがる進行中の対話をいつでも複数持つことができます。 対話コンテキストは異なる対話からの状態が衝突してバグの原因とならないようにします。

単一要求の間しか存続しない対話があります。 複数の要求にまたがる対話は Seam で提供されるアノテーションを付与して区分する必要があります。

タスク にもなる対話があります。 タスクとは長期実行のビジネスプロセスに対して重要な意味を持つ対話であり、 正しく完了するとビジネスプロセスの状態遷移を引き起こすことがあります。 Seam はタスクの区分用に特別なアノテーションのセットを提供します。

対話は **ネストされる** ことが可能なため、ひとつの対話はより広い対話の内部で発生します。これは拡張機能です。

要求間では、 対話状態は通常 Servlet セッション内に保持されます。 Seam は設定可能な **対話タイムアウト** を実装して自動的に非アクティブな対話を破棄し、 ユーザーが対話を中断したときに 1 ユーザーのログインセッションによって保持された状態が増大し続けないようにします。 同じプロセスで Seam は同じ長期実行の対話コンテキスト内の同時要求の処理をシリアライズします。

別の方法として、 クライアントのブラウザに対話の状態を保持するよう Seam を設定することもできます。

5.1.5. セッションのコンテキスト

セッションコンテキストはユーザーログインセッションに関連付けられた状態を保持します。 複数の対話間で状態を共有すると便利なことがありますが、 セッションコンテキストにはログインしたユーザーに関するグローバルな情報以外のコンポーネントは保持しないでください。

JSR-168 ポータル環境では、 セッションコンテキストはポートレットセッションを表します。

5.1.6. ビジネスプロセスのコンテキスト

ビジネスプロセスのコンテキストは長期実行のビジネスプロセスに関連付けられた状態を保持します。この状態は BPM エンジン (この場合は JBoss jBPM) によって管理や永続化が行われます。 ビジネスプロセスは複数ユーザーによる複数のインタラクションに渡ります。 この状態は複数ユーザーの間で明

確な方法で共有されます。 現在のタスクが現在のビジネスプロセスインスタンスを判断し、 ビジネスプロセスのライフサイクルは **プロセス定義の言語** で外部的に定義されるため、 ビジネスプロセス区分のための特別なアノテーションはありません。

5.1.7. アプリケーションのコンテキスト

アプリケーションのコンテキストとは Servlet 仕様の Servlet コンテキストのことです。 アプリケーションのコンテキストは主に設定データ、 参照データ、 メタモデルのような静的情報を保持するために使用されます。 例えば、 Seam はアプリケーションのコンテキスト内に Seam 自体の設定やメタモデルを保管します。

5.1.8. コンテキスト変数

コンテキストは **コンテキスト変数** 一式を使って名前空間を定義します。 これらは Servlet 仕様のセッションや要求属性と同様に機能します。 どのような値でもコンテキスト変数にバインドできますが、 通常 Seam コンポーネントのインスタンスにバインドします。

コンテキスト中のコンポーネントインスタンスはコンテキストの変数名により識別されます (コンテキストの変数名は、通常コンポーネント名に一致します)。 **Contexts** クラスで特定のスコープ内の名前付きコンポーネントインスタンスにプログラム的にアクセスすることができ、これにより **Context** インターフェースのスレッドに結びついた複数のインスタンスへのアクセスを提供します。

```
User user = (User) Contexts.getSessionContext().get("user");
```

名前に関連付けられた値の設定、変更も可能です。

```
Contexts.getSessionContext().set("user", user);
```

ただし、 コンポーネントは通常 **インジェクション** を通じてコンテキストから取得されます。 続いてコンポーネントインスタンスが **アウトジェクション** を通じてコンテキストに与えられます。

5.1.9. コンテキストの検索優先順位

コンポーネントのインスタンスは特定の既知のスコープから取得されることもありますが、 それ以外の場合はすべてのステートフルなスコープは次の優先順位で検索されます。

- イベントのコンテキスト
- ページのコンテキスト
- 対話のコンテキスト
- セッションのコンテキスト
- ビジネスプロセスのコンテキスト
- アプリケーションのコンテキスト

Contexts.lookupInStatefulContexts() を呼び出すことで優先順位の検索を行うことができます。 JSF ページから名前でコンポーネントにアクセスする場合は常に優先順位検索が発生します。

5.1.10. 同時実行モデル

Servlet 仕様も EJB 仕様も同じクライアントからの同時要求を管理する仕組みを定義していません。

Servlet コンテナはスレッドの安全性は確保せずにすべてのスレッドを同時に実行させます。 EJB コンテナによりステートレスなコンポーネント郡の同時アクセスが可能となるため、 複数のスレッドがひとつのステートフルセッション Bean にアクセスすると例外を送出します。 Web アプリケーションベースの詳細な同期要求にはこれで十分です。 ただし、 頻繁に非同期 (AJAX) 要求を使用する最近のアプリケーションの場合は同時実行サポートが不可欠です。 したがって、 Seam は同時実行管理層をそのコンテキストモデルに追加します。

Seam ではセッションおよびアプリケーションのコンテキストはマルチスレッドになり、 複数の同時要求を並行して処理することができます。 イベントとページのコンテキストはシングルスレッドになります。 厳密に言えばビジネスプロセスのコンテキストはマルチスレッドですが、 実際には同時実行は非常に稀なため通常は無視しても構わないでしょう。Seam は対話コンテキストに対し **1 プロセスに対し 1 対話でシングルスレッド** となるモデルを強制するために、1 つの長期実行の対話コンテキストで複数の同時要求をシリアライズします。

セッションコンテキストはマルチスレッドで不安定な状態を含むことが多いため、Seam インターセプタが有効の間はセッションスコープのコンポーネントは常に Seam により同時アクセスから保護されます。 インターセプタが無効の場合、必要なスレッドの安全性に関するあらゆる対策はコンポーネント自体で実装しなければなりません。 Seam はデフォルトで要求をセッションスコープのセッション Bean と JavaBean にシリアライズし、 発生するデッドロックをすべて検出して破棄します。 ただし、アプリケーションによってスコープされたコンポーネントの場合はこれはデフォルトの動作ではありません。 これらのコンポーネントが通常は不安定な状態を保持せず、またグローバルな同期は非常にコスト高となるためです。 シリアライズされたスレッドモデルは **@Synchronized** アノテーションを追加することで、あらゆるセッション Bean や JavaBean コンポーネントに強制できます。

この同時実行モデルは、 開発者側での特別な作業を必要とすることなく、 AJAX クライアントが安全に不安定なセッションや対話状態を使用できることを意味します。

5.2. SEAM コンポーネント

Seam コンポーネントは POJO (Plain Old Java Objects) です。具体的には、JavaBean または Enterprise JavaBean 3.0 (EJB3) エンタープライズ Bean です。 Seam では コンポーネントが EJB である必要はなく、また EJB3 準拠のコンテナがなくても使用できますが、 EJB3 を念頭にして設計され EJB3 と強く統合します。 Seam は以下の **コンポーネントタイプ**をサポートします。

- EJB3 ステートレスセッション Bean
- EJB3 ステートフルセッション Bean
- EJB3 エンティティ Bean (JPA エンティティクラスなど)
- JavaBeans
- EJB3 メッセージ駆動型 Bean
- Spring Bean (26章*Spring Framework 統合* を参照)

5.2.1. ステートレスセッション Bean

ステートレスセッション Bean のコンポーネントは複数の呼び出しに対して状態を保持することができません。 従って、 通常は各種の Seam コンテキスト内の他のコンポーネントの状態に応じて動作します。 JSF のアクションリスナーとして使用できますが、 表示に関しては JSF コンポーネントにプロパティを提供することはできません。

ステートレスセッション Bean は常にステートレスコンテキスト内に存在します。 各要求に対して新しいインスタンスが使用されるため、 ステートレスセッション Bean は同時にアクセスが可能です。 EJB3 コンテナはインスタンスを要求に割り当てます (通常インスタンスは再利用可能なプールから割り

振られるため、 インスタンス変数は以前に使用した Bean からのデータを保持することができます)。

Seam ステートレスセッション Bean コンポーネントは `Component.getInstance()` または `@In(create=true)` のいずれかを使用してインスタンス化されます。 JNDI 検索や`new` 演算子で直接インスタンス化しないでください。

5.2.2. ステートフルセッション Bean

ステートフルセッション Bean コンポーネントは Bean の複数の呼び出しに対して状態を保持できるだけでなく、 複数の要求に対しても状態を保持することができます。 データベースに属さないアプリケーションの状態はすべてステートフルセッション Bean によって保持されるはずです。 これが Seam と他の多くの Web アプリケーションフレームワークとの大きな違いです。 現在の対話データは、 `HttpSession` ではなく対話コンテキストにバインドされたステートフルセッション Bean のインスタンス変数内に格納します。 これにより、 Seam が状態のライフサイクルを管理し、 異なる同時の対話に関する状態間で衝突が起こらないようにします。

ステートフルセッション Bean は JSF アクションリスナーまたはバッキング Bean として表示やフォームのサブミット用にプロパティを JSF コンポーネントに提供するためによく使用されます。

ステートフルセッション Bean はデフォルトで対話コンテキストにバインドされます。 ページもしくはステートレスコンテキストにはバインドできません。

Seam インターセプタが有効の間は、Seam により同時要求はセッションスコープのステートフルセッション Bean にシリアライズされます。

Seam ステートフルセッション Bean コンポーネントは `Component.getInstance()` または `@In(create=true)` のいずれかを使用してインスタンス化されます。 JNDI 検索や`new` 演算子で直接インスタンス化を行わないでください。

5.2.3. エンティティ Bean

エンティティ Bean をコンテキスト変数にバインドさせると Seam コンポーネントとして機能させることができます。 エンティティにはコンテキスト依存識別子に加え永続識別子があるため、 エンティティのインスタンスは Seam によって暗黙的にインスタンス化されるのではなく、 通常は Java コード内で明示的にバインドされます。

エンティティ Bean コンポーネントはバイジェクションやコンテキスト区分に対応しません。 また、 エンティティ Bean が引き起こす検証の呼び出しにも対応していません。

通常、 エンティティ Bean は JSF アクションリスナーとしては使用されず、 表示あるいはフォームのサブミット用に JSF コンポーネントにプロパティを提供するバッキング Bean として機能させることが多くあります。 ステートレスセッション Bean アクションリスナーと合わせてバッキング Bean として使用し create、 update、 delete などのタイプの機能を実装するのが一般的です。

エンティティ Bean はデフォルトで対話コンテキストにバインドされます。 ステートレスコンテキストにはバインドできません。

注記

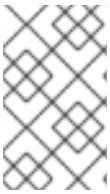
クラスタ化環境では、 エンティティ Bean を直接対話 (またはセッションスコープ Seam コンテキスト変数) にバインドすることは、ステートフルセッション Bean を使って エンティティ Bean を参照することより効率性に欠けます。 このためすべての Seam アプリケーションがエンティティ Bean を Seam コンポーネントとして定義するわけではありません。

Seam エンティティ Bean コンポーネントは **Component.getInstance()** または **@In(create=true)** のいずれかを使用してインスタンス化されるか、**new** 演算子を使用して直接インスタンス化されます。

5.2.4. JavaBeans

JavaBean はステートレスまたはステートフルセッション Bean と同じように使用されます。ただし、宣言的なトランザクション区分、宣言的なセキュリティ、効率的なクラスタ化状態の複製、EJB3 永続性、タイムアウトのメソッドなどの機能は備えていません。

章の後半では、EJB コンテナなしで Seam や Hibernate を使用する方法を見ていきます。この例では、コンポーネントはセッション Bean ではなく JavaBean です。



注記

クラスタ化環境では、対話やセッションスコープの Seam JavaBean コンポーネントのクラスタ化はステートフルセッション Bean コンポーネントのクラスタ化より効率性に欠けます。

JavaBean はデフォルトでイベントコンテキストにバインドされます。

Seam は同時要求を常にセッションスコープの JavaBean にシリアライズします。

Seam JavaBean コンポーネントは **Component.getInstance()** または **@In(create=true)** のいずれかを使用してインスタンス化されます。**new** 演算子で直接インスタンス化しないでください。

5.2.5. メッセージ駆動型 Bean

メッセージ駆動型 Bean は Seam コンポーネントとして機能することができます。しかし、その呼び出しメソッドは他の Seam コンポーネントのそれとは異なります。コンテキスト変数で呼び出されるのではなく、JMS キューまたはトピックに送信されたメッセージを待ち受けます。

メッセージ駆動型 Bean は Seam コンテキストにはバインドできません。また、その **呼び出し元** となるセッションや対話状態にもアクセスできません。ただし、バイジェクションおよび他の Seam のいくつかの機能には対応します。

メッセージ駆動型 Bean はアプリケーションによってインスタンス化されることはありません。メッセージを受信すると EJB コンテナによってインスタンス化されます。

5.2.6. インターセプション

バイジェクション、コンテキスト区分、検証などのアクションを実行するためには、Seam はコンポーネントの呼び出しをインターセプトしなければなりません。JavaBean の場合、Seam がコンポーネントのインスタンス化を完全に制御するため特別な設定は不要です。エンティティ Bean の場合、バイジェクションとコンテキスト区分が定義されていないためインターセプションは不要です。セッション Bean の場合、EJB インターセプタをセッション Bean コンポーネントに登録しなければなりません。以下のようにアノテーションを使用することで行います。

```
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login { ... }
```

しかし、**ejb-jar.xml** もインターセプタを定義する方がよいでしょう。

■

```
  <interceptors>
  <interceptor>
    <interceptor-class>
      org.jboss.seam.ejb.SeamInterceptor
    </interceptor-class>
  </interceptor>
</interceptors>

<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>*</ejb-name>
    <interceptor-class>
      org.jboss.seam.ejb.SeamInterceptor
    </interceptor-class>
  </interceptor-binding>
</assembly-descriptor>
```

5.2.7. コンポーネント名

すべての Seam コンポーネントには名前が必要です。 **@Name** アノテーションを使用してコンポーネントに名前を割り当てます。

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login { ... }
```

これが **Seam コンポーネント名** で、 EJB 仕様で定義された他の名前との関連はありません。 しかし、 Seam コンポーネント名は JSF 管理 Bean 名のような働きをするため、 同一の条件と考えることができます。

@Name だけがコンポーネント名を定義する唯一の方法ではありませんが、 この名前は常に指定しなければなりません。 名前が定義されていないと他の Seam アノテーションが機能しません。

Seam はコンポーネントのインスタンス化を行うときに、 その新しいインスタンスをコンポーネントの設定スコープ内のコンポーネント名と一致する変数にバインドします。 これは Seam により XML ではなくアノテーションでこのマッピングを設定できるという点以外、 JSF 管理 Bean の動作とまったく同じです。 また、 プログラムによってコンポーネントをコンテキスト変数にバインドすることもできます。 これは特定の 1 コンポーネントがシステム内で複数のロールを担っている場合に便利です。 たとえば、 現在の **User** を **currentUser** セッションコンテキスト変数にバインドする一方、管理機能の対象となる **User** も **user** 対話コンテキスト変数にバインドすることがあります。 Seam コンポーネントを参照するコンテキスト変数を上書きすることは可能なため、プログラムによってバインドを行う場合は注意してください。

非常に大規模なアプリケーションの場合や組み込み Seam コンポーネントの場合には、 命名競合を避けるため修飾コンポーネント名がよく使用されます。

```
@Name("com.jboss.myapp.loginAction")
@Stateless
public class LoginAction implements Login { ... }
```

Java コードや JSF の式言語中でも修飾コンポーネント名を使用することができます。

```
<h:commandButton type="submit" value="Login"
   action="#{com.jboss.myapp.loginAction.login}"/>
```

これは混雑しているため、Seam は修飾名を簡易名にエイリアスする機能も提供しています。 以下のような行を **components.xml** ファイルに追加します。

```
<factory name="loginAction" scope="STATELESS"
         value="#{com.jboss.myapp.loginAction}"/>
```

すべての組み込み Seam コンポーネントには修飾名がありますが、 Seam の **名前空間をインポートする** 機能によって非修飾名でもアクセスすることができます。 Seam JAR に含まれる **components.xml** ファイルは以下の名前空間を定義します。

```
<components xmlns="http://jboss.com/products/seam/components">
  <import>org.jboss.seam.core</import>
  <import>org.jboss.seam.cache</import>
  <import>org.jboss.seam.transaction</import>
  <import>org.jboss.seam.framework</import>
  <import>org.jboss.seam.web</import>
  <import>org.jboss.seam.faces</import>
  <import>org.jboss.seam.international</import>
  <import>org.jboss.seam.theme</import>
  <import>org.jboss.seam.pageflow</import>
  <import>org.jboss.seam.bpm</import>
  <import>org.jboss.seam.jms</import>
  <import>org.jboss.seam.mail</import>
  <import>org.jboss.seam.security</import>
  <import>org.jboss.seam.security.management</import>
  <import>org.jboss.seam.security.permission</import>
  <import>org.jboss.seam.captcha</import>
  <import>org.jboss.seam.excel.exporter</import>
  <!-- ... -->
</components>
```

非修飾名の解決を試行すると、 Seam は順にそれぞれの名前空間を調べます。 アプリケーション固有となる追加の名前空間をアプリケーションの **components.xml** ファイルに含めることができます。

5.2.8. コンポーネントスコープの定義

@Scope アノテーションを使用してコンポーネントのスコープ (コンテキスト) を上書きし、 Seam によるインスタンス化のときにコンポーネントインスタンスがバインドされるコンテキストを定義することができます。

```
@Name("user")
@Entity
@Scope(SESSION)
public class User { ... }
```

org.jboss.seam.ScopeType は可能なスコープの列挙を定義します。

5.2.9. 複数のロールを持つコンポーネント

システム内で複数の役割を果たす Seam コンポーネントクラスがあります。 たとえば、 **User** クラスは通常現在のユーザーを表すセッションスコープのコンポーネントですが、 ユーザー管理画面では対話スコープのコンポーネントになります。 **@Role** アノテーションを使用すると、 1 つのコンポーネントに異なるスコープで追加の名前が付いたロールを定義することができます。 これにより、 同じコン

ポーネントクラスを別のコンテキスト変数にバインドさせることができます (いずれの Seam コンポーネントの **インスタンス** も複数のコンテキスト変数にバインド可能ですが、 クラスレベルで行えるため自動インスタンス化を利用することができます)。

```
@Name("user")
@Entity
@Scope(CONVERSATION)
@Role(name="currentUser", scope=SESSION)
public class User { ... }
```

@Roles アノテーションは必要に応じてロールを追加で指定することができます。

```
@Name("user")
@Entity
@Scope(CONVERSATION)
@Roles({ @Role(name="currentUser", scope=SESSION),
         @Role(name="tempUser", scope=EVENT)})
public class User { ... }
```

5.2.10. 組み込みコンポーネント

Seam は組み込みのインターセプタとコンポーネントのセットとして実装されています。 これにより、ランタイムのアプリケーションによる組み込みコンポーネントとの通信、または組み込みコンポーネントをカスタムの実装に置き換えることによる Seam の基本機能のカスタマイズが容易になります。組み込みコンポーネントは Seam の名前空間 **org.jboss.seam.core** および同じ名前の Java パッケージで定義されます。

組み込みコンポーネントは他の Seam コンポーネントと同様にインジェクトすることも可能ですが、**instance()** という便利で静的なメソッドも提供しています。

```
FacesMessages.instance().add("Welcome back, #{user.name}!");
```

5.3. バイジェクション

依存性の注入 または **制御の反転** (IoC) により、 コンテナが setter メソッドあるいはインスタンス変数にコンポーネントを「インジェクト」することで、あるコンポーネントが他のコンポーネントを参照することが可能となります。これまでの依存性の注入の実装では、インジェクションはコンポーネントの構成時に起こるため、 参照はコンポーネントインスタンスのライフタイムの間は変化しませんでした。これはステートレスコンポーネントには理にかなっています。 クライアントの観点から見ると、特定のステートレスなコンポーネントの全インスタンスは交換可能です。一方、 Seam はステートフルなコンポーネントの使用に重点を置いているため、構成としての従来の依存性の注入は有用ではなくなりました。Seam はインジェクションの一般化として **バイジェクション** の概念を導入しています。 インジェクションと対比すると、 バイジェクションは以下のようになります。

コンテキスト依存

バイジェクションは各種のコンテキストからステートフルなコンポーネントを組み立てるために使用されます。 **より広い** コンテキストからのコンポーネントは**より狭い** コンテキストからのコンポーネントへの参照を行うこともできます。

双方向的

値はコンテキスト変数から呼び出されたコンポーネントの属性にインジェクトされ、コンテキストに戻されます (アウトジェクション)。 これによりそれ自体のインスタンス変数を設定するだけで、呼び出されたコンポーネントはコンテキスト依存の変数の値を操作することができます。

動的

コンテキスト依存の変数の値は時間経過で変化し、 Seam のコンポーネントはステートフルであるため、 バイジェクションはコンポーネントが呼び出されるたびに発生します。

要するに、インスタンス変数の値がインジェクトされる、 またはアウトジェクトされる、 あるいはその両方が行われることを指定することで、バイジェクションによりコンテキスト変数をコンポーネントのインスタンス変数にエイリアスできます。アノテーションを使用してバイジェクションを有効にします。

@In アノテーションは値がインスタンス変数または **setter** メソッドにインジェクトされることを指定します。 インスタンス変数の場合、

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  @In User user;
  ...
}
```

setter メソッドの場合、

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  User user;
  @In
  public void setUser(User user) { this.user=user; }
  ...
  }
```

デフォルトでは、 Seam はプロパティ名またはインジェクトされたインスタンス変数名を使用してすべてのコンテキストの優先順位検索を行います。 例えば、 **@In("currentUser")** を使って明示的にコンテキスト変数名を指定したいと思われるかもしれません。

名前付きコンテキスト変数にバインドされた既存のコンポーネントインスタンスが存在しないときに Seam にコンポーネントのインスタンスを作成させたい場合は、 **@In(create=true)** を指定します。値がオプションで (null でも可能) あれば **@In(required=false)** を指定します。

いくつかのコンポーネントでは、 使用されるたびに **@In(create=true)** を指定することは同じ動作の繰り返しとなる場合があります。 このような場合、 コンポーネントに **@AutoCreate** アノテーションを付与します。 これにより **create=true** を明示的に使用しなくても必要なときに常に作成されるようになります。

式値をインジェクトすることも可能です。

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  @In("#{user.username}") String username;
  ...
}
```

インジェクトした値はメソッドの完了とアウトジェクションの直後にディスインジェクトされます (つまり **null** に設定されます)。

(コンポーネントのライフサイクルおよびインジェクションについての詳細は次の章を参照してください。)

@Outアノテーションは属性がインスタンス変数または **getter** メソッドのいずれかからアウトジェクトされることを指定します。 インスタンス変数の場合、

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  @Out User user;
  ...
}
```

getter メソッドの場合、

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  User user;

  @Out
  public User getUser() {
    return user;
  }
  ...
}
```

属性はインジェクト、アウトジェクトされることが可能です。

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  @In
  @Out User user;
  ...
}
```

または

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
  User user;

  @In
  public void setUser(User user) {
    this.user=user;
  }

  @Out
  public User getUser() {
    return user; }
    ...
}
```

5.4. ライフサイクルのメソッド

セッション Bean とエンティティ Bean の Seam コンポーネントは通常の EJB3 ライフサイクルの全コールバック (**@PostConstruct**、 **@PreDestroy** など) をサポートしていますが、 Seam は JavaBean コンポーネントでのこれらコールバックの使用もサポートします。 ただし、 これらのアノテーションは J2EE 環境では有効とならないため Seam は **@PostConstruct** と **@PreDestroy** と等価な 2 つの追加コンポーネントライフサイクルのコールバックを定義します。

@Create メソッドは Seam がコンポーネントをインスタンス化した後に呼び出されます。 コンポーネントが定義できるのは 1 つの **@Create**メソッドのみになります。

@Destroy メソッドは Seam コンポーネントがバインドするコンテキストが終了すると呼び出されます。 コンポーネントが定義できるのは 1 つの **@Destroy** メソッドのみです。

また、 ステートフルセッション Bean コンポーネントは **@Remove** アノテーションが付いたパラメータのないメソッドをひとつ定義 **しなければなりません**。 このメソッドはコンテキストが終了すると Seam により呼び出されます。

最後に、**@Startup** アノテーションはいずれのアプリケーションまたはセッションスコープのコンポーネントにも適用することができます。 **@Startup** アノテーションは、 クライアントによって参照されるのを待たずにコンテキストが開始したら直ちに Seam にコンポーネントをインスタンス化するよう指示します。 **@Startup(depends={....})** を指定するとスタートアップコンポーネントをインスタンス化する順序を制御することができます。

5.5. 条件付きインストール

@Install アノテーションは、特定のデプロイメントシナリオでは必要とされるがそれ以外では必要とされないコンポーネントの条件付きインストールを制御します。これは以下の場合に役立ちます。

- 試験的に基盤となる模擬コンポーネントを作成する
- 特定のデプロイメントシナリオでコンポーネント実装を変更する、または
- コンポーネントの依存部分が利用可能な場合に、 そのコンポーネントをインストールする (フレームワークの作成者に便利)。

@Install で **優先順位** と **依存性** を指定することができます。

コンポーネントの優先順位は、 クラスパスに同じコンポーネント名を持つクラスが複数ある場合にインストールすべきコンポーネントの決定に Seam が使用する番号です。 Seam はより優先順位が高いコンポーネントを選択します。事前定義された優先順位の値があります (昇順)。

1. **BUILT_IN** – 最も優先順位が低いコンポーネントは Seam に組み込まれたコンポーネントです。
2. **FRAMEWORK** – サードパーティのフレームワークによって定義されたコンポーネントは組み込みコンポーネントより優先させることができますが、 アプリケーションコンポーネントにより優先されます。
3. **APPLICATION** – デフォルトの優先順位です。ほとんどのアプリケーションコンポーネントにはこれが適切です。
4. **DEPLOYMENT** – デプロイメント固有のアプリケーションコンポーネント用です。

5. **MOCK** – テストで使用されるモックオブジェクト用です。

JMS キューと対話する **messageSender** という名前のコンポーネントがあるとします。

```
@Name("messageSender")
public class MessageSender {

  public void sendMessage() {
    //do something with JMS
  }
}
```

ユニットテストでは、 有効な JMS キューがないのでこのメソッドをスタブにしたいと思うでしょう。 ユニットテストの実行中にクラスパスには存在しているがアプリケーションでは絶対にデプロイされない **モック** コンポーネントを作成します。

```
@Name("messageSender")
@Install(precedence=MOCK)
public class MockMessageSender extends MessageSender {
  public void sendMessage() {
    //do nothing!
  }
}
```

precedence はクラスパスで両方のコンポーネントを発見したときに Seam が使用するバージョンを決定するときに役立ちます。

クラスパスにどのクラスがあるのかを正確に制御できるなら、これはすばらしいことです。 しかし、多くの依存性を持つ再利用可能なフレームワークを記述している場合、 複数の **jar** 全体に渡りそのフレームワークを分散させたいとは思わないでしょう。 インストール済みの別のコンポーネントやクラスパスにある使用可能なクラスに応じてインストールするコンポーネントを決定する方が好まれるはずです。 **@Install** アノテーションはこの機能も制御します。Seam は多くの組み込みコンポーネントの条件付きインストールを実現するために内部でこのメカニズムを使用します。

5.6. ロギング

Seam の前は最も単純なログメッセージでさえ冗長なコードが必要でした。

```
private static final Log log = LogFactory.getLog(CreateOrderAction.class);

public Order createOrder(User user, Product product, int quantity) {
  if ( log.isDebugEnabled() ) {
    log.debug("Creating new order for user: " + user.username() +
              " product: " + product.name() + " quantity: " + quantity);
  }
  return new Order(user, product, quantity);
}
```

Seam はこうしたコードを大幅に簡素化するロギング API を提供します。

```
@Logger private Log log;

public Order createOrder(User user, Product product, int quantity) {
  log.debug("Creating new order for user: #0 product: #1 quantity: #2",
```

```
            user.username(), product.name(), quantity);

  return new Order(user, product, quantity);
}
```

log 変数を静的に宣言したかどうかに関係なく、 エンティティ Bean コンポーネント (**log** 変数が静的でなければならない) 以外ならいずれでも動作します。

文字列連結は **debug()** メソッド内部で起こるため、 冗長な **if (log.isDebugEnabled())** による保護は不要です。 Seam は **log** のインジェクト先を認識できるため、 通常はログカテゴリを明示的に指定する必要もありません。

User と **Product** が現在のコンテキストで有効な Seam コンポーネントならコードはさらに簡潔になります。

```
@Logger private Log log;
public Order createOrder(User user, Product product, int quantity) {
  log.debug("Creating new order for user: #{user.username}
            product: #{product.name} quantity: #0", quantity);
  return new Order(user, product, quantity);
}
```

Seam ロギングは出力の送信先を log4j または JDK ロギングのどちらにするのかを自動的に選択します。 log4j がクラスパスにある場合はこれが使用されます。そうでない場合は Seam は JDK ロギングを使用します。

5.7. MUTABLE インターフェースと @READONLY

多くのアプリケーションサーバーは **HttpSession** のクラスタリングを備えており、**setAttribute** 明示的に呼び出された場合にのみ、セッションにバインドした可変のオブジェクトの状態への変化が複製されます。これはフェールオーバーが発生する場合にのみ出現するバグを招く可能性があり、 開発時に効果的なテストを行うことができません。 さらに、 複製メッセージ自体はセッション属性にバインドしたシリアライズされたオブジェクトグラフ全体を含むため効率的ではありません。

EJB ステートフルセッション Bean は自動ダーティチェックを行い (つまり、 自動的にオブジェクト状態の変更を検出して更新された状態をデータベースと同期させる必要があります)、 可変状態を複製する必要があります。 洗練された EJB コンテナは属性レベルの複製などの最適化の導入が可能です。 残念ながらすべての Seam ユーザーが EJB3 に対応する環境で作業をしているわけではないので、 Seam はセッションスコープや対話スコープの JavaBean およびエンティティ Bean のコンポーネント用にクラスタセーフな状態管理の追加的なレイヤを提供します。

セッションスコープや対話スコープの JavaBean コンポーネントの場合、 このコンポーネントがアプリケーションにより呼び出されると Seam は各要求ごとに 1 度 **setAttribute()** を呼び出して自動的に複製を強制します。 ただし、 この方法は read-mostly コンポーネントには役立ちません。 **org.jboss.seam.core.Mutable** インターフェースを実装、または **org.jboss.seam.core.AbstractMutable** を拡張して、 コンポーネントの中に独自のダーティチェックのロジックを記述しこの動作を制御します。 以下に例を示します。

```
@Name("account")
public class Account extends AbstractMutable {
  private BigDecimal balance;
  public void setBalance(BigDecimal balance) {
    setDirty(this.balance, balance);
    this.balance = balance;
```

```
  }

  public BigDecimal getBalance() {
    return balance;
  }
  ...
}
```

または、 同様の結果を得るために **@ReadOnly** アノテーションを使用することもできます。

```
@Name("account")
public class Account {
  private BigDecimal balance;
  public void setBalance(BigDecimal balance) {
    this.balance = balance;
  }

  @ReadOnly
  public BigDecimal getBalance() {
    return balance;
  }
  ...
}
```

セッションスコープや対話スコープのエンティティ Bean コンポーネントの場合、 (対話スコープの) エンティティが現在 Seam 管理永続コンテキストに関連付けられているため複製が不要にならない限り、 Seam は各要求ごとに 1度 **setAttribute()** を呼び出して自動的に複製を強制します。 この方法は必ずしも効率的とは限らないので、 セッションや対話スコープのエンティティ Bean は注意して使用してください。 エンティティ Bean インスタンスの「管理」にはステートフルセッション Bean や JavaBean コンポーネントをいつでも記述することができます。 以下に例を示します。

```
@Stateful @Name("account")
public class AccountManager extends AbstractMutable {
  private Account account; // an entity bean
  @Unwrap
  public Account getAccount() {
    return account;
  }
  ...
}
```

Seam Application Framework の **EntityHome** クラスは Seam コンポーネントを使ったエンティティ Bean インスタンスの管理に適した例となる点に留意してください。

5.8. ファクトリとマネージャのコンポーネント

Seam コンポーネントではないオブジェクトを扱わなければならないこともよくありますが、**@In** を使用して Seam コンポーネントにインジェクトし、値メソッドバインディング式およびメソッドバインディング式でそれらを使用して Seam コンテキストのライフサイクルに関連付けたい場合があります (例えば **@Destroy** など)。 このため、 Seam コンテキストは Seam コンポーネントではないオブジェクトを保持することができ、 Seam にはコンテキストにバインドする非コンポーネントオブジェクトとの作業を簡略化する機能が複数備わっています。

ファクトリコンポーネントパターンにより Seam コンポーネントをコンポーネントではないオブジェ

クトに対してインスタンス化を行う機能として動作させることができます。 **ファクトリメソッド** はコンテキスト変数が参照されると呼び出されますが、 バインドされた値は持っていません。 **@Factory** アノテーションを使用してファクトリメソッドを定義します。 ファクトリメソッドは値をコンテキスト変数にバインドし、 バインドした値のスコープを決定します。ファクトリメソッドのスタイルは 2 種類あります。 最初のスタイルは Seam によりコンテキストにバインドされる値を返します。

```
@Factory(scope=CONVERSATION)
public List<Customer> getCustomerList() {
  return ... ;
}
```

2 番目のスタイルは、 値をコンテキスト変数自体にバインドする **void** タイプのメソッドです。

```
@DataModel List<Customer> customerList;
@Factory("customerList")
public void initCustomerList() {
  customerList = ...  ;
}
```

どちらの場合も、 **customerList** コンテキスト変数が参照されその値が null になるとファクトリメソッドが呼び出されます。 ファクトリメソッドはその値のライフサイクルではこれ以上何も持っていません。 さらに強力なパターンは **マネージャコンポーネントパターン**です。 この場合、 コンテキスト変数にバインドする Seam コンポーネントがコンテキスト変数の値を管理し、 残りはクライアントから見えません。

マネージャコンポーネントとは **@Unwrap** メソッドを持つあらゆるコンポーネントです。 このメソッドはクライアント側から見える値を返し、 コンテキスト変数が参照されるたびに呼び出されます。

```
@Name("customerList")
@Scope(CONVERSATION)
public class CustomerListManager {
  ...
  @Unwrap
  public List<Customer> getCustomerList() {
    return ... ;
  }
}
```

マネージャコンポーネントパターンはコンポーネントのライフサイクルにより制御を必要とする場合に特に便利です。 例えば、 コンテキスト終了時にクリーンアップを必要とする重量オブジェクトがある場合、 オブジェクトを **@Unwrap** してマネージャコンポーネントの **@Destroy** メソッドでクリーンアップを実行することが可能です。

```
@Name("hens")
@Scope(APPLICATION)
public class HenHouse {
  Set<Hen> hens;

  @In(required=false) Hen hen;

  @Unwrap
  public List<Hen> getHens()
  {
    if (hens == null) {
```

```
      // Setup our hens }
    return hens;
  }

  @Observer({"chickBorn", "chickenBoughtAtMarket"})
  public addHen() {
    hens.add(hen);
  }

  @Observer("chickenSoldAtMarket")
  public removeHen() {
    hens.remove(hen);
  }

  @Observer("foxGetsIn")
  public removeAllHens() {
    hens.clear();
  }

  ...
}
```

ここでは基礎をなすオブジェクトに変更を加える多くのイベントを管理コンポーネントが監視しています。 コンポーネントはこうした動作自体を管理し、 オブジェクトはアクセスされるたびにアンラップされるため、 一貫性のあるビューが提供されます。

第6章 SEAM コンポーネントの構成

Seam では XML ベースの設定の必要性を最小限に抑えることを目的としています。 ただし、 XML を使って Seam を設定したいという理由はさまざまです。 Java コードからデプロイメント固有の情報を切り離したい、 再利用可能なフレームワークを作成可能にしたい、 Seam 組み込み機能を構成したい等の理由です。 Seam はコンポーネントの設定に対して 2 つのアプローチを提供します。 プロパティファイルまたは **web.xml** でのプロパティ設定によるコンポーネントの設定と**components.xml** によるコンポーネントの設定です。

6.1. プロパティ設定によるコンポーネントの構成

(システムプロパティの) サーブレットコンテキストパラメータ、またはクラスパスのルートにある **seam.properties** プロパティファイルのいずれかを持つ設定プロパティを Seam に与えることができます。

設定可能な Seam コンポーネントは設定可能な属性の JavaBeans スタイルのプロパティ setter メソッドを公開しなければなりません。 つまり、 **com.jboss.myapp.settings** という名前の Seam コンポーネントに **setLocale()** という setter メソッドがある場合、 次のいずれかを与えることができます。

- **seam.properties** ファイル内に **com.jboss.myapp.settings.locale** という名前のプロパティを与えることができます。
- 起動時に **-D** で **org.jboss.seam.properties.com.jboss.myapp.settings.locale** という名前のシステムプロパティを与えることができます。
- または、 サーブレットコンテキストパラメータとして同じシステムプロパティを与えることができます。

これらのいずれもクラスパスのルートで **locale** 属性値を設定します。

同じメカニズムが Seam 自体の設定にも使われます。 たとえば、 対話のタイムアウトを設定するには、 **org.jboss.seam.core.manager.conversationTimeout** の値を **web.xml** または **seam.properties** 内に与えるか、 **org.jboss.seam.properties** が先頭に付いたシステムプロパティで与えます。 (**setConversationTimeout()** という setter メソッドを持つ **org.jboss.seam.core.manager** という名前の組み込み Seam コンポーネントがあります。)

6.2. COMPONENTS.XML によるコンポーネントの設定

components.xml ファイルはプロパティ設定よりパワフルです。次を行うことができます。

- **@Name** アノテーションが付けられ、Seam のデプロイメントスキャナーで検出されたアプリケーションコンポーネントや組み込みコンポーネントなど自動的にインストールされているコンポーネントの設定を行います。
- Seam コンポーネントとして **@Name** アノテーションが付かないクラスをインストールします。別々の名前で複数回インストールが可能なインフラストラクチャコンポーネントに対して最も役立ちます (たとえば、 Seam 管理永続コンテキストなど)。
- **@Name** アノテーションは付いているが、そのコンポーネントはインストールしないことを示す **@Install** アノテーションが付いているためデフォルトではインストールされないコンポーネントをインストールします。
- コンポーネントのスコープを無効にします。

components.xml ファイルは次の 3 つの異なる場所のいずれかに置くことができます。

- **WAR** の **WEB-INF** ディレクトリ
- **JAR** の **META-INF** ディレクトリ
- **@Name** アノテーション付きのクラスを含む任意の**JAR** ディレクトリ

コンポーネントにデフォルトではインストールしないことを示している **@Install** アノテーションがない限り、デプロイメントスキャナーが **seam.properties** ファイルまたは **META-INF/components.xml** ファイルを持つ **@Name** アノテーション付きのクラスを見つけた場合、Seam コンポーネントはインストールされます。**components.xml** ファイルはアノテーションを無効にしなければならない特殊なケースを処理します。

例えば次の **components.xml** ファイルは jBPM をインストールします。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:bpm="http://jboss.com/products/seam/bpm">
  <bpm:jbpm/>
</components>
```

次の例も jBPM をインストールします。

```
<components>
  <component class="org.jboss.seam.bpm.Jbpm"/>
</components>
```

この例は 2 種類の異なる Seam 管理永続コンテキストをインストールして設定します。

```
<components xmlns="http://jboss.com/products/seam/components"

xmlns:persistence="http://jboss.com/products/seam/persistence">

    <persistence:managed-persistence-context name="customerDatabase"
         persistence-unit-jndi-name="java:/customerEntityManagerFactory"/>

    <persistence:managed-persistence-context name="accountingDatabase"
       persistence-unit-jndi-name="java:/accountingEntityManagerFactory"/>

</components>
```

この例も 2 種類の異なる Seam 管理永続コンテキストをインストールして設定します。

```
<components>
  <component name="customerDatabase"

class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">
      java:/customerEntityManagerFactory
    </property>
  </component>

  <component name="accountingDatabase"
```

```
class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">
      java:/accountingEntityManagerFactory
    </property>
  </component>
</components>
```

この例はセッションスコープの Seam 管理永続コンテキストを作成します (実際にはお勧めしません)。

```
<components xmlns="http://jboss.com/products/seam/components"

xmlns:persistence="http://jboss.com/products/seam/persistence">

  <persistence:managed-persistence-context
          name="productDatabase" scope="session"
          persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components>
```

```
<components>

  <component name="productDatabase" scope="session"

class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">
      java:/productEntityManagerFactory
    </property>
  </component>

</components>
```

永続コンテキストなど基盤となるオブジェクトには **auto-create** オプションが一般的に使用され、**@In** アノテーションを使うときに明示的に **create=true** を指定する必要がありません。

```
<components xmlns="http://jboss.com/products/seam/components"

xmlns:persistence="http://jboss.com/products/seam/persistence">

  <persistence:managed-persistence-context
          name="productDatabase" auto-create="true"
          persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components>
```

```
<components>

  <component name="productDatabase"
    auto-create="true"
          class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">
      java:/productEntityManagerFactory
    </property>
```

```
  </component>

</components>
```

<factory> 宣言は値バインディング式もしくはメソッドバインディング式を指定し、 これが最初に参照されたときにコンテキスト変数の値を初期化します。

```
<components>
  <factory name="contact" method="#{contactManager.loadContact}"
           scope="CONVERSATION"/>
</components>
```

次のように Seam コンポーネントの **エイリアス** (別名) が生成可能です。

```
<components>
  <factory name="user" value="#{actor}" scope="STATELESS"/>
</components>
```

よく使用される式に対してもエイリアスを作成できます。

```
<components>
  <factory name="contact" value="#{contactManager.contact}"
           scope="STATELESS"/>
</components>
```

auto-create="true" は **<factory>** 宣言とよく併用されます。

```
<components>
  <factory name="session" value="#{entityManager.delegate}"
           scope="STATELESS" auto-create="true"/>
</components>
```

デプロイメントとテストの両方において同じ **components.xml** ファイルが使用されることがあります (若干の変更あり)。 Seam は **components.xml** 内に **@wildcard@** 形式のワイルドカードを配置することが可能で、 Ant ビルドスクリプトまたはクラスパスに **components.properties** というファイルを与えることによって置き換えることができます (2 番目のアプローチを Seam のサンプルで見ることができます)。

6.3. 細分化した構成ファイル

XML の構成が必要なコンポーネントが大量にある場合は **components.xml** をいくつかの小さいファイルに分割する方が実用的でしょう。 Seam では、 **com.helloworld.Hello** という名前のクラスの設定は **com/helloworld/Hello.component.xml** という名前のリソース内に置くことができます (このパターンは Hibernate でも使われています)。このファイルのルートエレメントは **<components>** または **<component>** エレメントのいずれかが可能です。

<components> ではこのファイル内に複数のコンポーネントを定義することができます。

```
<components>

  <component class="com.helloworld.Hello" name="hello">
    <property name="name">#{user.name}</property>
  </component>
```

```
  <factory name="message" value="#{hello.message}"/>

</components>
```

<component> では 1 つのコンポーネントしか設定できませんが、 冗長性が抑えられます。

```
<component name="hello">
  <property name="name">#{user.name}</property>
</component>
```

2 番目のエレメントにあるクラス名はコンポーネント定義が表れるファイルによって暗示されます。

あるいは、**com/helloworld/components.xml**で **com.helloworld** パッケージ内のすべてのクラスの設定をすることも可能です。

6.4. 設定可能なプロパティのタイプ

文字列、 プリミティブ、 プリミティブラッパータイプのプロパティは次のように設定します。

- org.jboss.seam.core.manager.conversationTimeout 60000
- <core:manager conversation-timeout="60000"/>

```
        <component name="org.jboss.seam.core.manager">
        <property name="conversationTimeout">60000</property>
        </component>
```

文字列またはプリミティブの配列、 セット、 一覧にも対応します。

```
org.jboss.seam.bpm.jbpm.processDefinitions
order.jpdl.xml,
return.jpdl.xml,
inventory.jpdl.xml
```

```
<bpm:jbpm>
  <bpm:process-definitions>
    <value>order.jpdl.xml</value>
    <value>return.jpdl.xml</value>
    <value>inventory.jpdl.xml</value>
  </bpm:process-definitions>
</bpm:jbpm>
```

```
<component name="org.jboss.seam.bpm.jbpm">

  <property name="processDefinitions">
    <value>order.jpdl.xml</value>
    <value>return.jpdl.xml</value>
    <value>inventory.jpdl.xml</value>
  </property>

</component>
```

文字列値のキーと文字列またはプリミティブの値から成るマップでさえもサポートされます。

```
<component name="issueEditor">

  <property name="issueStatuses">
    <key>open</key> <value>open issue</value>
    <key>resolved</key> <value>issue resolved by developer</value>
    <key>closed</key> <value>resolution accepted by user</value>
  </property>

</component>
```

複数の値を持つプロパティを設定する場合、 Seam は **SortedSet/SortedMap** が使用されていない限りデフォルトでは **components.xml** に設定された属性の順序を維持します。 この場合、Seam は **TreeMap/TreeSet** を参照します。 プロパティに具体的なタイプ **(LinkedList** など) がある場合はそのタイプを使用します。

次のように完全修飾クラス名を指定することでそのタイプを上書きすることも可能です。

```
<component name="issueEditor">

  <property name="issueStatusOptions" type="java.util.LinkedHashMap">
    <key>open</key> <value>open issue</value>
    <key>resolved</key> <value>issue resolved by developer</value>
    <key>closed</key> <value>resolution accepted by user</value>
  </property>

</component>
```

最後に、値バインディング式を使ってコンポーネントをリンクさせることができます。 これは呼び出し時ではなくコンポーネントのインスタンス化時に起こるため、 **@In** でのインジェクションとは非常に異なる点に注意してください。 JavaServer Faces (JSF) や Spring などの従来の IoC コンテナによって提供される依存性インジェクションに似ています。

```
<drools:managed-working-memory name="policyPricingWorkingMemory"
                               rule-base="#{policyPricingRules}"/>
```

```
<component name="policyPricingWorkingMemory"
           class="org.jboss.seam.drools.ManagedWorkingMemory">
  <property name="ruleBase">#{policyPricingRules}</property>
</component>
```

Seam はコンポーネントの Bean プロパティへ初期値を代入する前に EL 式の文字列も解決します。 このためコンテキスト依存データの中にはコンポーネントにインジェクトできるものもあります。

```
<component name="greeter" class="com.example.action.Greeter">
  <property name="message">
    Nice to see you, #{identity.username}!
  </property>
</component>
```

ただし、1 つ重要な例外があります。 初期値が Seam の **ValueExpression** または **MethodExpression** のいずれかに割り当てられる場合、 その EL の評価は遅延されて適切な式のラッパーが生成されプロパティに割り当てられます。 Seam Application Framework の **Home** コンポーネントにあるメッセージテンプレートがその一例です。

```
<framework:entity-home name="myEntityHome"
                       class="com.example.action.MyEntityHome"
                       entity-class="com.example.model.MyEntity"
                       created-message="'#{myEntityHome.instance.name}'
                       has been successfully added."/>
```

コンポーネントの内部では、 **ValueExpression** または **MethodExpression** のいずれかで **getExpressionString()** を呼び出すと式の文字列にアクセスすることができます。 プロパティが **ValueExpression** となる場合はその値を **getValue()** で解決します。 プロパティが **MethodExpression** となる場合は **invoke({Object arguments})** でメソッドを呼び出します。 **MethodExpression** プロパティに値を割り当てるには、 初期値全体がひとつの EL式でなければなりません。

6.5. XML 名前空間の使用

前述の例では 2 種類のコンポーネント宣言メソッドを XML 名前空間を使うものと使わないものに修正しています。 以下に名前空間を使用しない典型的な **components.xml** ファイルを示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd">

  <component class="org.jboss.seam.core.init">
    <property name="debug">true</property>
    <property name="jndiPattern">@jndiPattern@</property>
  </component>

</components>
```

ご覧の通りこのコードは冗長です。 さらにコンポーネントと属性の名前がデプロイメント時に確認できません。

名前空間を使用すると、

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/core
               http://jboss.com/products/seam/core-2.2.xsd
               http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd">

  <core:init debug="true" jndi-pattern="@jndiPattern@"/>

</components>
```

スキーマ宣言は冗長ではありますが、 XML の内容自体は簡潔かつ理解しやすいものになります。 このスキーマは各コンポーネントと利用可能な属性に関する詳細情報を与えて、 XML エディタによるインテリジェントな自動補完入力を可能にします。 名前空間付きのエレメントの使用により、適切な **components.xml** ファイルの生成と保守が容易になります。

これは組み込みの Seam コンポーネントに対しては良く機能しますが、 ユーザーのコンポーネントに対してはオプションが 2 つあります。 最初に Seam は両方の混在したモデルに対応することで、ユーザーのコンポーネントには汎用の **<component>** 宣言を使用できるようにし、 また組み込みコンポーネントには名前空間が付いた宣言が使用できるようにしています。 さらに重要な点は、 Seam により独自のコンポーネントに対して簡単に名前空間を宣言できるということです。

いずれの Java パッケージにも **@Namespace** アノテーションをパッケージに付加することによって XML 名前空間を関連付けることができます (パッケージレベルのアノテーションはパッケージディレクトリ内の **package-info.java** という名前のファイルに宣言します)。 seampay デモからの例を示します。

```
@Namespace(value="http://jboss.com/products/seam/examples/ seampay")
package org.jboss.seam.example.seampay; import
org.jboss.seam.annotations.Namespace;
```

components.xml で名前空間のスタイルを使用するのはこんなに簡単です。 次のように記述できます。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:pay="http://jboss.com/products/seam/examples/seampay"
            ... >

  <pay:payment-home new-instance="#{newPayment}"
       created-message="Created a new payment to #{newPayment.payee}" />

  <pay:payment name="newPayment"
               payee="Somebody"
               account="#{selectedAccount}"
               payment-date="#{currentDatetime}"
               created-date="#{currentDatetime}" />
     ...
</components>
```

または、

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:pay="http://jboss.com/products/seam/examples/seampay"
            ... >

  <pay:payment-home>
    <pay:new-instance>"#{newPayment}"</pay:new-instance>
    <pay:created-message>
      Created a new payment to #{newPayment.payee}
    </pay:created-message>
  </pay:payment-home>

  <pay:payment name="newPayment">
    <pay:payee>Somebody"</pay:payee>
    <pay:account>#{selectedAccount}</pay:account>
    <pay:payment-date>#{currentDatetime}</pay:payment-date>
    <pay:created-date>#{currentDatetime}</pay:created-date>
   </pay:payment>
    ...
</components>
```

前述の例では名前空間付きエレメントの 2 種類の使用モデルを説明しています。 最初の宣言では **<pay:payment-home>** が **paymentHome** コンポーネントを参照しています。

```
package org.jboss.seam.example.seampay;
...
@Name("paymentHome")
public class PaymentController extends EntityHome<Payment> {
  ...
}
```

そのエレメント名はコンポーネント名をハイフンで連結した形式です。 そのエレメントの属性はプロパティ名をハイフンで連結した形式です。

2 番目の宣言では、 **<pay:payment>** エレメントが **org.jboss.seam.example.seampay** パッケージにある **Payment** クラスを参照しています。 この場合 **Payment** は Seam コンポーネントとして宣言されているエンティティです。

```
package org.jboss.seam.example.seampay;
...
@Entity
public class Payment implements Serializable {
  ...
}
```

ユーザー定義のコンポーネントに対して妥当性検証と自動補完入力を機能させるにはスキーマが必要になります。 Seam は複数のコンポーネントからなるセットに対してはまだスキーマを自動的には生成できないため、 手動で作成しなければなりません。 参考として標準的な Seam パッケージ用のスキーマ定義を使用できます。

次は Seam によって使用される名前空間です。

- components – **http://jboss.com/products/seam/components**
- core – **http://jboss.com/products/seam/core**
- drools – **http://jboss.com/products/seam/drools**
- framework – **http://jboss.com/products/seam/framework**
- jms – **http://jboss.com/products/seam/jms**
- remoting – **http://jboss.com/products/seam/remoting**
- theme – **http://jboss.com/products/seam/theme**
- security – **http://jboss.com/products/seam/security**
- mail – **http://jboss.com/products/seam/mail**
- web – **http://jboss.com/products/seam/web**
- pdf – **http://jboss.com/products/seam/pdf**
- spring – **http://jboss.com/products/seam/spring**

第7章 イベント、インターセプタ、例外処理

コンテキスト依存コンポーネントモデルを補完するために、Seam アプリケーションの特徴である極度の疎結合を促進させる 2 つの基本概念が存在します。1 つ目の基本概念は、強力なイベントモデルであり、イベントは JavaServer Faces (JSF) のイベントのようなメソッドバインディング式を通じてイベントリスナーにマップされます。2 つ目の概念は、アノテーションやインターセプタを広範囲に使用し、ビジネスロジックを実装するコンポーネントに対して横断的関心事を適用しているということです。

7.1. SEAM イベント

Seam コンポーネントモデルは **イベント駆動アプリケーション**との併用を目的として開発されました。特に、粒度の細かいイベントモデルで、疎結合の粒度の細かいコンポーネントの開発が行えるようになります。 Seam にはイベントのタイプがいくつかあります。

- JSF イベント
- jBPM 遷移イベント
- Seam ページアクション
- Seam コンポーネント駆動イベント
- Seam コンテキスト依存イベント

これらの多様なイベントすべては JSF EL メソッドバインディング式を通じて Seam コンポーネントへマップされます。JSF イベントは、JSF テンプレートで次のように定義されます。

```
<h:commandButton value="Click me!" action="#{helloWorld.sayHello}"/>
```

jBPM 遷移イベントは、jBPM プロセス定義またはページフロー定義で規定されます。

```
<start-page name="hello" view-id="/hello.jsp">
    <transition to="hello">
    <action expression="#{helloWorld.sayHello}"/>
    </transition>
    </start-page>
```

JSF イベントや jPBM イベントの詳細は本書以外にも記載されているため、ここでは Seam によって定義される別の 2 種類のイベントについて見ていきます。

7.2. ページアクション

Seam ページアクションはページのレンダリングの直前に発生するイベントです。 ページアクションは **`WEB-INF/pages.xml`** で宣言します。 特定の JSF ビュー ID に対してページアクションを定義することも可能です。

```
<pages>
    <page view-id="/hello.jsp" action="#{helloWorld.sayHello}"/>
    </pages>
```

あるいは、**`view-id`** へのサフィックスとして **`*`** ワイルドカードを使用し、パターンに一致するすべてのビュー ID に適用するアクションを指定します。

```
<pages>
```

```
    <page view-id="/hello/*" action="#{helloWorld.sayHello}"/>
    </pages>
```

注記

<page> エレメントが細かなページ記述子で定義されている場合は暗黙的に定義されるため **view-id** 属性を省略することができます。

複数のワイルドカード化されたページアクションが現在のビュー ID に一致する場合は、Seam は指定が明確でないアクションから指定が明確なアクションの順ですべてのアクションを呼び出します。

ページアクションのメソッドは JSF の結果を返すことができます。 その結果 が null でなければ、Seam はビューへの移動に定義済みナビゲーションルールを使用します。

<page> エレメントに記載されている ビュー ID は実際の JSP や Facelet ページに対応する必要がありません。 このため、 ページアクションを使って Struts や WebWork のような従来のアクション指向のフレームワーク機能を再生することができます。 HTTP GET など Faces 以外の要求への応答に複雑な動作を行う場合に便利です。

複数または条件付きのページアクションは **<action>** タグを使って指定できます。

```
<pages>
  <page view-id="/hello.jsp">
    <action execute="#{helloWorld.sayHello}"
            if="#{not validation.failed}"/>
    <action execute="#{hitCount.increment}"/>
  </page>
</pages>
```

ページアクションは初期の要求 (Faces 以外) とポストバック (Faces) 要求の両方で実行されます。 ページアクションを使用してデータをロードするとポストバックで実行されている標準の JSF アクションと競合する場合があります。 ページアクションを無効にするひとつの方法として、初期要求でのみ **true** に解決する条件を設定します。

```
<pages>
  <page view-id="/dashboard.xhtml">
    <action execute="#{dashboard.loadData}"
            if="#{not FacesContext.renderKit.responseStateManager
                .isPostback(FacesContext)}"/>
  </page>
</pages>
```

この条件は **ResponseStateManager#isPostback(FacesContext)** を参照して要求がポストバックであるかどうかを判断します。 ResponseStateManager には **FacesContext.getCurrentInstance().getRenderKit(). getResponseStateManager()** を使ってアクセスします。

Seam はこの冗長性の少ない結果を得ることができる組み込みの条件を提供しています。 **on-postback** 属性を **false** に設定するとポストバックでページアクションを無効にすることができます。

```
<pages>
    <page view-id="/dashboard.xhtml">
```

```
    <action execute="#{dashboard.loadData}" on-postback="false"/>
    </page>
    </pages>
```

on-postback 属性はデフォルトでは **true** に設定され後方互換性を維持します。 ただし、 **false** を使用することも多々あります。

7.3. ページパラメータ

Faces 要求 (JSF フォーム送信) は「アクション」 (メソッドバインディング) と「パラメータ」 (入力値バインディング) の両方をカプセル化します。 ページアクションにもパラメータが必要な場合があります。

Faces 以外 (GET) の要求はブックマーク可能なため、ページパラメータはヒューマンリーダブルな要求パラメータとして引き渡されます。

ページパラメータはアクションメソッドを指定してもしなくても使用できます。

7.3.1. 要求パラメータのモデルへのマッピング

Seam により名前付き要求パラメータをモデルオブジェクトの属性にマッピングさせる値バインディングを提供することができます。

```
<pages>
  <page view-id="/hello.jsp" action="#{helloWorld.sayHello}">
    <param name="firstName" value="#{person.firstName}"/>
    <param name="lastName" value="#{person.lastName}"/>
  </page>
</pages>
```

<param> 宣言は JSF 入力の値バインディングと同様に双方向性です。

- ビュー ID に対する Faces 以外 (GET) の要求が発生すると、 Seam は適切なタイプ変換を実行した後、 名前付きパラメータの値をそのモデルオブジェクトに設定します。
- 任意の **<s:link>** や **<s:button>** は透過的に要求パラメータを含みます。 パラメータ値は、レンダリングフェーズの間に (**<s:link>** がレンダリングされるとき) 値バインディングを評価することによって決定されます。
- そのビュー ID への **<redirect/>** を持つナビゲーションルールはすべて要求パラメータを透過的に含みます。 パラメータの値はアプリケーション起動フェーズの最後で値バインディングを評価することにより決定されます。
- その値は特定のビュー ID を持つページの全 JSF フォーム送信で透過的に伝播します。 つまりビューパラメータは Faces 要求の **PAGE**スコープのコンテキスト変数のように動作します。

ただし、値バインディングで参照されるモデル属性の値である **/hello.jsp** に到着し、その値は対話 (または他のサーバー側の状態) を必要とせずにメモリに保持されます。

7.4. 要求パラメータの伝播

name 属性しか指定されていない場合、 要求パラメータは**PAGE** コンテキストを使って伝播されます (つまり、 モデルプロパティへはマッピングされません)。

```
<pages>
  <page view-id="/hello.jsp" action="#{helloWorld.sayHello}">
    <param name="firstName" />
    <param name="lastName" />
  </page>
</pages>
```

ページパラメータの伝播は、マルチレイヤのマスター / 詳細の CRUD ページを作成したいときに特に便利です。 それは (例えば、保存ボタンを押したときの) ビューや編集していたエンティティを「覚えておく」のに使えます。

- 要求パラメータがビューのページパラメータとして記載されていると、**<s:link>** や **<s:button>** はすべて透過的にその要求パラメータを伝播します。
- その値は特定のビュー ID を持つページの全 JSF フォーム送信で透過的に伝播されます (つまり、 ビューパラメータは Faces 要求の **PAGE**スコープのコンテキスト変数のように動作します)。

これらはかなり複雑ですが、 時間をかけてページパラメータを理解することは間違いなく価値があります。 ページパラメータは Faces 以外の要求全体に状態を伝播する最も洗練された方法です。特に次のような状況で役立ちます。たとえば、 検索結果のページをブックマークできる検索画面がある場合、ページパラメータによって同じコードでの POST 要求と GET の要求の処理について記述することになります。ページパラメータを使用するとビュー定義で繰り返し要求パラメータを記載する必要がなく、リダイレクトをもっと簡単にコーディングできます。

7.5. ページパラメータでの URL 書き換え

書き換えは **pages.xml** 内のビューに対して発見されるパターンに応じて発生します。Seam の URL 書き換えは同一のパターンに基づいて受信および発信 URL 書き換えを行います。このプロセスの簡単なパターンを以下に示します。

```
<page view-id="/home.xhtml">
  <rewrite pattern="/home" />
</page>
```

この場合は、 **/home** の着信要求はすべて **/home.xhtml** に送られます。 通常は **/home.seam** をポイントする生成されたリンクは **/home** に書き換えられます。 書き換えパターンはクエリパラメータの前の URL 部分にのみ一致します。 それゆえ、 **/home.seam?conversationId=13** と **/home.seam?color=red** は両方ともこの書き換え規則で一致します。

書き換え規則は、以下の規則に示すようにクエリパラメータを考慮することができます。

```
<page view-id="/home.xhtml">
  <rewrite pattern="/home/{color}" />
  <rewrite pattern="/home" />
</page>
```

この場合、 **/home/red** の着信要求はあたかも **/home.seam?color=red** であるように処理されます。 同様に、 color がページパラメータの場合は **/home.seam?color=blue** と通常表示される発信 URL は代わりに **/home/blue** と出力されます。 規則は順番に処理されるため一般的な規則より先に限定的な規則を記述することが重要です。

デフォルトの Seam クエリパラメータも URL 書き換えを使ってマッピングが可能で、 さらに Seam のフィンガープリントを隠します。 次の例では **/search.seam?conversationId=13** は **/search-13** と書き換えられます。

```
<page view-id="/search.xhtml">
  <rewrite pattern="/search-{conversationId}" />
  <rewrite pattern="/search" />
</page>
```

Seam URL 書き換えによりビュー単位でのシンプルで双方向の書き換えが可能になります。Seam 以外のコンポーネントを対象とするさらに複雑な書き換え規則については、Seam アプリケーションは **org.tuckey.URLRewriteFilter** を使用し続ける、または Web サーバーで書き換え規則を適用させることができます。

URL 書き換えを使用する場合は Seam の **書き換えフィルタ** を有効にする必要があります。 書き換えフィルタについては 「URL の書き換え」 で説明します。

7.6. 変換と妥当性検証

複雑なモデルプロパティに JSF コンバータを次のいずれかの方法で指定することができます。

```
<pages>
  <page view-id="/calculator.jsp" action="#{calculator.calculate}">
    <param name="x" value="#{calculator.lhs}"/>
    <param name="y" value="#{calculator.rhs}"/>
    <param name="op" converterId="com.my.calculator.OperatorConverter"
           value="#{calculator.op}"/>
  </page>
</pages>
```

```
<pages>
  <page view-id="/calculator.jsp" action="#{calculator.calculate}">
    <param name="x" value="#{calculator.lhs}"/>
    <param name="y" value="#{calculator.rhs}"/>
    <param name="op" converter="#{operatorConverter}"
           value="#{calculator.op}"/>
  </page>
</pages>
```

次のいずれかの方法で、JSF バリデータと **required="true"** を使用することもできます。

```
<pages>
  <page view-id="/blog.xhtml">
    <param name="date" value="#{blog.date}"
           validatorId="com.my.blog.PastDate" required="true"/>
  </page>
</pages>
```

```
<pages>
  <page view-id="/blog.xhtml">
    <param name="date" value="#{blog.date}"
```

```
            validator="#{pastDateValidator}" required="true"/>
  </page>
</pages>
```

モデルベースの Hibernate バリデータのアノテーションは自動的に認識され検証されます。 Seam は文字列パラメータ値を日付に変換してまた戻すためにデフォルトの日付コンバータも提供します。

型変換や妥当性検証が失敗すると、 グローバルな **FacesMessage** が **FacesContext** に追加されます。

7.7. ナビゲーション

Seam アプリケーションの **faces-config.xml** で定義された標準の JSF ナビゲーションルールを使用することができます。ただし、このルールには制限がいくつかあります。

- リダイレクト時に要求パラメータの使用は指定できません。
- 規則から対話を開始または終了することはできません。
- 規則はアクションメソッドの戻り値を評価することにより動作するため、 任意の EL 式を評価することはできません。

pages.xml と **faces-config.xml** の間に、「オーケストレーション」ロジックが分散するという別の問題があります。**pages.xml** でこのロジックを統合した方が適切です。

この JSF ナビゲーションルールは

```
<navigation-rule>
  <from-view-id>/editDocument.xhtml</from-view-id>
  <navigation-case>
    <from-action>#{documentEditor.update}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/viewDocument.xhtml</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

次のように書き直すことができます。

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule if-outcome="success">
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

しかし、このメソッドは **DocumentEditor** を文字列の戻り値 (JSF の結果) で汚してしまいます。 代わりに Seam では次のように記述することができます。

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}"
              evaluate="#{documentEditor.errors.size}">
    <rule if-outcome="0">
      <redirect view-id="/viewDocument.xhtml"/>
```

```
    </rule>
  </navigation>
</page>
```

または、次のように記述することもできます。

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

最初の形式は値バインディングを評価して後続のルールにより使用される結果の値を決定します。2 番目の方法は結果を無視してそれぞれ可能なルールに対して値バインディングを評価します。

更新が成功したら、 現在の対話を以下のように終了させることができます。

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <end-conversation/>
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

対話が終了しているため、 後続の要求は関心があるドキュメントを認識しなくなります。要求パラメータとしてドキュメント ID を渡すことができ、これによりビューをブックマーク可能にします。

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <end-conversation/>
      <redirect view-id="/viewDocument.xhtml">
        <param name="documentId" value="#{documentEditor.documentId}"/>
      </redirect>
    </rule>
  </navigation>
</page>
```

結果が null となるのは JSF では特別なケースであり、「そのページを再表示する」という意味に解釈されます。 次のナビゲーションルールは null 以外ならいずれの結果とも一致しますが、 null の結果には **一致しません**。

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <rule>
      <render view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

結果が null の場合にナビゲーションを実行するには次の形式を使います。

```
<page view-id="/editDocument.xhtml">
  <navigation from-action="#{documentEditor.update}">
    <render view-id="/viewDocument.xhtml"/>
  </navigation>
</page>
```

ビュー ID は JSF EL 式となることができます。

```
<page view-id="/editDocument.xhtml">
  <navigation>
    <rule if-outcome="success">
      <redirect view-id="/#{userAgent}/displayDocument.xhtml"/>
    </rule>
  </navigation>
</page>
```

7.8. ナビゲーション、 ページアクション、 パラメータを定義するための詳細に設定されたファイル

異なるページアクションやパラメータが大量にある場合、 または単に大量のナビゲーションルールがある場合、 それらの定義を複数の小さいファイルに分割した方が適切でしょう。 ビュー ID **/calc/calculator.jsp** を持つページのアクションやパラメータは**calc/calculator.page.xml** という名前のリソースに定義できます。 この場合、 **<page>** がルートエレメントであり、 ビュー ID は暗黙的に指定されます。

```
<page action="#{calculator.calculate}">
  <param name="x" value="#{calculator.lhs}"/>
  <param name="y" value="#{calculator.rhs}"/>
  <param name="op" converter="#{operatorConverter}" value="#
{calculator.op}"/>
</page>
```

7.9. コンポーネント駆動イベント

Seam コンポーネント同士は互いのメソッドを呼び出して通信します。 ステートフルコンポーネントは監視側または監視可能パターンを実装することもできます。 ただし、より疎結合な通信を有効にするために Seam には **コンポーネント駆動イベント** が備わっています。

イベントリスナー (監視側) を **components.xml** に指定します。

```
<components>
  <event type="hello">
    <action execute="#{helloListener.sayHelloBack}"/>
    <action execute="#{logger.logHello}"/>
  </event>
</components>
```

ここでは **イベントタイプ** は任意の文字列です。

イベントが発生すると、そのイベント用に登録されたアクションが **components.xml** で出現する順番に従って呼び出されます。イベントを発生させるために Seam は組み込みコンポーネントを提供します。

```
@Name("helloWorld")
public class HelloWorld {
    public void sayHello() {
        FacesMessages.instance().add("Hello World!");
        Events.instance().raiseEvent("hello");
    }
}
```

また、以下のようにアノテーションを使うことも可能です。

```
@Name("helloWorld")
public class HelloWorld {
    @RaiseEvent("hello")
    public void sayHello() {
        FacesMessages.instance().add("Hello World!");
    }
}
```

このイベント生成側はイベント消費側には依存しません。イベントリスナーはまったく生成側に依存関係を持つことなく実装することができます。

```
@Name("helloListener")
public class HelloListener {
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

上記の **components.xml** で定義されたメソッドバインディングはイベントを消費側にマッピングします。アノテーションを使用してこれを行うこともできます。

```
@Name("helloListener")
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

コンポーネント駆動のイベントを知っている方なら、なぜイベントオブジェクトについて今まで言及してこなかったか疑問に思われるかもしれません。Seam ではイベントオブジェクトはイベント生成側とリスナー間で状態を伝播する必要がありません。状態は Seam コンテキストで保持されコンポーネント間で共有されます。ただし、イベントオブジェクトを渡したければ次のようにすることも可能です。

```
@Name("helloWorld")
public class HelloWorld {
    private String name;
    public void sayHello() {
        FacesMessages.instance().add("Hello World, my name is #0.", name);
```

```
        Events.instance().raiseEvent("hello", name);
    }
}
```

```
@Name("helloListener")
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack(String name) {
        FacesMessages.instance().add("Hello #0!", name);
    }
}
```

7.10. コンテキスト依存イベント

Seam は特定の種類のフレームワーク統合にアプリケーションによって使用される組み込みイベントをいくつか定義しています。 次にそのイベントを示します。

表7.1 コンテキスト依存イベント

イベント	詳細
org.jboss.seam.validationFailed	JSFvalidation が失敗すると呼び出されます。
org.jboss.seam.noConversation	長期実行の対話が存在せず長期実行の対話が必要とされる場合に呼び出されます。
org.jboss.seam.preSetVariable.<name>	コンテキスト変数 <name> が設定されると呼び出されます。
org.jboss.seam.postSetVariable.<name>	コンテキスト変数 <name> が設定されると呼び出されます。
org.jboss.seam.preRemoveVariable.<name>	コンテキスト変数 <name> の設定が解除されると呼び出されます。
org.jboss.seam.postRemoveVariable.<name>	コンテキスト変数 <name> の設定が解除されると呼び出されます。
org.jboss.seam.preDestroyContext.<SCOPE>	<SCOPE> コンテキストが破棄される前に呼び出されます。
org.jboss.seam.postDestroyContext.<SCOPE>	<SCOPE> コンテキストが破棄された後に呼び出されます。
org.jboss.seam.beginConversation	長期実行の対話が始まるときに必ず呼び出されます。
org.jboss.seam.endConversation	長期実行の対話が終了するときに必ず呼び出されます。

イベント	詳細
org.jboss.seam.conversationTimeout	対話のタイムアウトが発生すると呼び出されます。対話 ID はパラメータとして渡されます。
org.jboss.seam.beginPageflow	ページフローが開始すると呼び出されます。
org.jboss.seam.beginPageflow.<name>	ページフロー <name> が開始すると呼び出されます。
org.jboss.seam.endPageflow	ページフローが終了すると呼び出されます。
org.jboss.seam.endPageflow.<name>	ページフロー <name> が終了すると呼び出されます。
org.jboss.seam.createProcess.<name>	プロセス <name> が作成されると呼び出されます。
org.jboss.seam.endProcess.<name>	プロセス <name> が終了すると呼び出されます。
org.jboss.seam.initProcess.<name>	プロセス <name> が対話に関連付けられると呼び出されます。
org.jboss.seam.initTask.<name>	タスク <name> が対話に関連付けられると呼び出されます。
org.jboss.seam.startTask.<name>	タスク <name> が開始すると呼び出されます。
org.jboss.seam.endTask.<name>	タスク <name> が終了すると呼び出されます。
org.jboss.seam.postCreate.<name>	コンポーネント <name> が作成されると呼び出されます。
org.jboss.seam.preDestroy.<name>	コンポーネント <name> が破棄されると呼び出されます。
org.jboss.seam.beforePhase	JSF フェーズの開始前に呼び出されます。
org.jboss.seam.afterPhase	JSF フェーズの終了後に呼び出されます。
org.jboss.seam.postInitialization	Seam により全コンポーネントの初期化および起動が終了すると呼び出されます。

イベント	詳細
org.jboss.seam.postReInitialization	再デプロイの後、Seam により全コンポーネントの再初期化および起動が終了すると呼び出されます。
org.jboss.seam.exceptionHandled.<type>	キャッチされない例外が Seam により処理されると呼び出されます。
org.jboss.seam.exceptionHandled	キャッチされない例外が Seam により処理されると呼び出されます。
org.jboss.seam.exceptionNotHandled	キャッチされない例外にハンドラがなかった場合に呼び出されます。
org.jboss.seam.afterTransactionSuccess	Seam Application Framework でトランザクションが成功すると呼び出されます。
org.jboss.seam.afterTransactionSuccess.<name>	エンティティ **<name>** を管理する Seam Application Framework でトランザクションが成功すると呼び出されます。
org.jboss.seam.security.loggedOut	ユーザーがログアウトすると呼び出されます。
org.jboss.seam.security.loginFailed	ユーザー認証が失敗すると呼び出されます。
org.jboss.seam.security.loginSuccessful	ユーザーが正常に認証されると呼び出されます。
org.jboss.seam.security.notAuthorized	承認確認が失敗すると呼び出されます。
org.jboss.seam.security.notLoggedIn	認証されたユーザーがなく、認証が必要な場合に呼び出されます。
org.jboss.seam.security.postAuthenticate	ユーザーが認証された後に呼び出されます。
org.jboss.seam.security.preAuthenticate	ユーザーの認証試行の前に呼び出されます。

Seam コンポーネントは、他のコンポーネント駆動イベントを監視するのと同じようにこれらのイベントを監視します。

7.11. SEAM インターセプタ

EJB3 ではセッション Bean コンポーネントに標準的なインターセプタモデルが導入されました。 Bean にインターセプタを追加するには、 **@AroundInvoke** というアノテーションが付加されたメソッドを

持つクラスを記述して、その Bean にインターセプタのクラス名を指定する **@Interceptors** のアノテーションを付ける必要があります。たとえば、次のインターセプタはアクションリスナーメソッドの呼び出しを許可する前にユーザーがログインされたかを確認します。

```
public class LoggedInInterceptor {
    @AroundInvoke
    public Object checkLoggedIn(InvocationContext invocation)
        throws Exception {
        boolean isLoggedIn = Contexts.getSessionContext()
            .get("loggedIn")!=null;
        if (isLoggedIn) {
            //the user is already logged in return invocation.proceed();
        } else {
            //the user is not logged in, fwd to login page return "login";
        }
    }
}
```

このインターセプタをアクションリスナーとして動作するセッション Bean に適用するには、そのセッション Bean **@Interceptors(LoggedInInterceptor.class)** というアノテーションを付加しなければなりません。ただし、Seam はクラスレベルのインターセプタ (**@Target(TYPE)** アノテーションが付与されたもの) 用にメタアノテーションとして **@Interceptors** を使えるようにすることで、EJB3 でのインターセプタフレームワーク上に構築されます。以下の例では、**@LoggedIn** アノテーションを生成します。

```
@Target(TYPE)
@Retention(RUNTIME)
@Interceptors(LoggedInInterceptor.class)
    public @interface LoggedIn {}
```

これでアクションリスナー Bean に **@LoggedIn** アノテーションを付与しインターセプタを適用することができます。

```
@Stateless
@Name("changePasswordAction")
@LoggedIn
@Interceptors(SeamInterceptor.class)
public class ChangePasswordAction implements ChangePassword {
    ...
        public String changePassword() {
            ...
        }
}
```

インターセプタの順番が重要な場合、インターセプタクラスに **@Interceptor** アノテーションを追加してインターセプタの特定の順序を指定します。

```
@Interceptor(around={BijectionInterceptor.class,
                     ValidationInterceptor.class,
                     ConversationInterceptor.class},
    within=RemoveInterceptor.class)
public class LoggedInInterceptor {
    ...
}
```

組み込み EJB3 の機能に対してクライアント側インターセプタを持たせることもできます。

```
@Interceptor(type=CLIENT)
public class LoggedInInterceptor {
    ...
}
```

EJB インターセプタはステートフルとなるため、 そのライフサイクルはインターセプトするコンポーネントのそれと一致します。 状態を維持する必要がないインターセプタの場合、 Seam によりパフォーマンスの最適化が実現し、**@Interceptor(stateless=true)** が指定されます。

Seam の多くの機能は、前の例で登場したようなインターセプタを含み、組み込みの Seam インターセプタ郡の 1 セットとして実装されます。 これらのインターセプタはインターセプト可能なすべての Seam コンポーネントに対し存在しているため、アノテーションを使って明示的にインターセプタを指定する必要はありません。

Seam のインターセプタは JavaBean コンポーネントと併用することもできます。

EJB はインターセプタを (**@AroundInvoke** を使った) ビジネスメソッドだけでなく、ライフサイクルメソッドの **@PostConstruct**、 **@PreDestroy**、 **@PrePassivate** そして **@PostActive** に対しても定義します。Seam はコンポーネントおよびインターセプタの両方でこれらのライフサイクルのメソッドを EJB3 Bean のみならず JavaBean コンポーネントに対してもサポートします (JavaBean コンポーネントにとって意味のない **@PreDestroy** は除きます)。

7.12. 例外の管理

JSF には例外処理に関して制限があります。 この問題に対処するため、 Seam はアノテーションを付けるか XML ファイルで宣言することで例外クラスの処理を定義することができます。これは EJB3 標準の **@ApplicationException** アノテーションと組み合わされ、例外がトランザクションロールバックの原因となるかどうかを指定します。

7.12.1. 例外およびトランザクション

Bean のビジネスメソッドにより例外が送出されるときに、その例外が現在のトランザクションに直ちにロールバックが必要としてマークするかどうかを制御する明確なルールを EJB は規定します。**システム例外** により常にトランザクションロールバックが発生します。**アプリケーション例外** はデフォルトではロールバックを発生させませんが **@ApplicationException(rollback=true)** が指定されるとロールバックが発生します (アプリケーション例外とはすべてのチェック例外、 または **@ApplicationException** アノテーションが付いたすべての非チェック例外です。 システム例外とは **@ApplicationException** アノテーションがないすべての非チェック例外です)。

注記

ロールバックとしてトランザクションにマークが付けられるのと、実際にトランザクションをロールバックすることは異なります。例外ルールではトランザクションにロールバックのマークが付けられることだけ言及していますが、 例外が送出された後でもそれはアクティブのままである可能性があります。

Seam は EJB3 例外のロールバックルールを Seam JavaBean コンポーネントに対しても適用します。

これらのルールは Seam コンポーネント層内でのみ適用されます。 例外が Seam コンポーネント層の外側で発生すると Seam はアクティブなトランザクションをすべてロールバックします。

7.12.2. Seam の例外処理を有効にする

Seam の例外処理を有効にするには、マスターのサーブレットフィルタを **web.xml** で宣言する必要があります。

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>*.seam</url-pattern>
</filter-mapping>
```

例外ハンドラを実行させるためには、 **web.xml** で Facelets 開発モードを無効にし **components.xml** で Seam デバッグモードを無効にする必要があります。

7.12.3. 例外処理に対するアノテーションの使用

次の例外は Seam コンポーネント層の外部に伝播すると必ず HTTP 404 エラーが発生します。送出されてもすぐに現在のトランザクションをロールバックしませんが、別の Seam コンポーネントによって例外がキャッチされないとこのトランザクションはロールバックされます。

```
@HttpError(errorCode=404)
public class ApplicationException extends Exception {
    ...
}
```

この例外は Seam コンポーネント層の外部に伝播すると必ずブラウザリダイレクトが発生します。また現在の対話も終了させます。 これにより現在のトランザクションを即時ロールバックすることになります。

```
@Redirect(viewId="/failure.xhtml", end=true)
@ApplicationException(rollback=true)
public class UnrecoverableApplicationException extends RuntimeException {
    ...
}
```

注記

Seam は JSF の **RENDER_RESPONSE** フェーズ中に発生する例外は処理できません。一度応答の出力が開始するとリダイレクトを実行することができないからです。

EL を使ってリダイレクト先の **viewId** を指定することも可能です。

この例外が Seam コンポーネント層の外部に伝播すると、 リダイレクトとなりユーザーにメッセージが表示されます。 また、 現在のトランザクションを直ちにロールバックします。

```
@Redirect(viewId="/error.xhtml", message="Unexpected error")
public class SystemException extends RuntimeException {
    ...
}
```

7.12.4. 例外処理に対する XML の使用

すべての例外クラスにアノテーションを付加することは不可能なので、 Seam ではこの機能を **pages.xml** でも指定できるようにしています。

```
<pages>
  <exception class="javax.persistence.EntityNotFoundException">
    <http-error error-code="404"/>
  </exception>

  <exception class="javax.persistence.PersistenceException">
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message>Database access failed</message>
    </redirect>
  </exception>

  <exception>
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message>Unexpected failure</message>
    </redirect>
  </exception>
</pages>
```

最後の **<exception>** 宣言はクラスを指定していないので、 アノテーションまたは**pages.xml** で指定された処理なしですべての例外に対してキャッチオールで動作します。

EL を使ってリダイレクト先の **view-id** を指定することもできます。

EL によって処理された例外インスタンスにアクセスすることもできます。 Seam はそれを対話コンテキストに置きます。 たとえば、 例外のメッセージにアクセスするには次のようにします。

```
...
  throw new AuthorizationException("You are not allowed to do this!");

<pages>
  <exception class="org.jboss.seam.security.AuthorizationException">
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message severity="WARN">
        #{org.jboss.seam.handledException.message}
      </message>
    </redirect>
  </exception>
</pages>
```

org.jboss.seam.handledException は例外ハンドラによって処理されたネストした例外を保持します。 その最も外側の (ラッパーの) 例外は **org.jboss.seam.caughtException** によって取得可能です。

7.12.4.1. 例外のロギングの抑制

pages.xml で定義されている例外ハンドラの場合、 例外がログ記録されるレベルを指定したり、 例外のログ記録を全て抑制することが可能です。 **log** および **log-level** の各属性を使用して例外のロギン

グを制御します。 以下に示すように、 **log="false"** が設定されている場合に指定された例外が発生するとログメッセージは生成されません。

```
<exception class="org.jboss.seam.security.NotLoggedInException"
           log="false">
  <redirect view-id="/register.xhtml">
    <message severity="warn">
      You must be a member to use this feature
    </message>
  </redirect>
</exception>
```

log 属性を指定しないとデフォルトでは **true** に設定されます。 つまり例外はログ記録されます。 代わりに **log-level** を指定して例外がログ記録されるレベルを制御することができます。

```
<exception class="org.jboss.seam.security.NotLoggedInException"
           log-level="info">
  <redirect view-id="/register.xhtml">
    <message severity="warn">
      You must be a member to use this feature
    </message>
  </redirect>
</exception>
```

log-level に指定できる値は、 **fatal**、 **error**、 **warn**、 **info**、 **debug**、 **trace** です。 **log-level** を指定しない、または無効な値を設定した場合は、**log-level** はデフォルトで **error** に設定されます。

7.12.5. 共通の例外

JPA を使用している場合

```
<exception class="javax.persistence.EntityNotFoundException">
  <redirect view-id="/error.xhtml">
    <message>Not found</message>
  </redirect>
</exception>

<exception class="javax.persistence.OptimisticLockException">
  <end-conversation/>
  <redirect view-id="/error.xhtml">
    <message>
      Another user changed the same data, please try again
    </message>
  </redirect>
</exception>
```

Seam Application Framework を使用している場合

```
<exception class="org.jboss.seam.framework.EntityNotFoundException">
  <redirect view-id="/error.xhtml">
    <message>Not found</message>
  </redirect>
</exception>
```

Seam Security を使用している場合

```
<exception class="org.jboss.seam.security.AuthorizationException">
  <redirect>
    <message>You don't have permission to do this</message>
  </redirect>
</exception>

<exception class="org.jboss.seam.security.NotLoggedInException">
  <redirect view-id="/login.xhtml">
    <message>Please log in first</message>
  </redirect>
</exception>
```

そして、 JSF の場合

```
<exception class="javax.Faces.application.ViewExpiredException">
  <redirect view-id="/error.xhtml">
    <message>Your session has timed out, please try again</message>
  </redirect>
</exception>
```

ユーザーがすでにセッションの期限切れとなったページに戻ると **ViewExpiredException** が発生します。 「長期実行の対話の必要」 で説明した **conversation-required** と **no-conversation-view-id** の設定により対話内で使用されたページにアクセスしながら、 セッションの有効期限に対して細かな制御が可能になります。

第8章 対話とワークスペースの管理

本章では Seam の対話モデルについて詳細に説明していきます。

Seam **対話** の概念は、3 つの別々のコンセプトが組み合わさり生まれました。

- **ワークスペース** というコンセプトと、効率的なワークスペースの管理。
- 楽観的なセマンティクスの **アプリケーショントランザクション** というコンセプト。 ステートレスアーキテクチャをベースとする既存のフレームワークでは、拡張永続のコンテキストを効率的に管理することはできませんでした。
- ワークフロー **タスク** というコンセプト。

こうした考えを統一しフレームワークで強力にサポートすることで、 以前よりすっきりしたコードでより豊かで効率的なアプリケーションを可能にするパワフルな構成概念を得ました。

8.1. SEAM の対話モデル

これまで見てきた例は、以下の規則を用いたシンプルな対話モデルで動作します。

- JSF 要求ライフサイクルのレスポンス出力フェーズ、 アプリケーション起動フェーズ、モデル値の更新フェーズ、バリデーション実行フェーズ、リクエスト値の適用フェーズなどの間は対話コンテキストは常にアクティブとなります。
- JSF 要求ライフサイクルのビュー復元フェーズの終了時に、Seam はそれまでの長期実行の全対話コンテキストの復元を試みます。 長期実行の対話コンテキストが存在しない場合は、Seam は一時的な新しい対話コンテキストを作成します。
- **@Begin** メソッドが出てくると、 一時的な対話コンテキストは長期実行の対話に昇格します。
- **@End** メソッドが出てくると、 すべての長期実行の対話コンテキストは一時的な対話に降格されます。
- JSF 要求ライフサイクルであるレスポンス出力フェーズの終わりには、Seam は長期実行の対話コンテキストの内容を記憶するか、 一時的な対話コンテキストの内容を破棄します。
- Faces 要求 (JSF ポストバック) はすべて対話コンテキストを伝播します。 デフォルトでは、Faces ではない要求 (GET 要求など) は対話コンテキストを伝播しません。
- JSF 要求のライフサイクルがリダイレクトで短縮される場合、 対話が既に **@End(beforeRedirect=true)** で終了されていない限り Seam は透過的に現在の対話コンテキストを格納して復元します。

Seam は透過的に対話コンテキスト (一時的な対話コンテキストを含む) を JSF ポストバックおよびリダイレクト全体に伝播します。 特に何も付けなければ **Faces でない要求** (GET 要求など) は対話コンテキストを伝播せず新たな一時対話内で処理されます。常にではありませんが、これが通常求められる動作です。

Faces でない要求全体に Seam の対話を伝播させたい場合、 要求パラメータとして Seam **対話 ID** を明示的にコード化する必要があります。

```
<a href="main.jsf?#{manager.conversationIdParameter}=#{conversation.id}">
  Continue
</a>
```

または、 JSFの場合

```
<h:outputLink value="main.jsf">
  <f:param name="#{manager.conversationIdParameter}"
     value="#{conversation.id}"/>
  <h:outputText value="Continue"/>
</h:outputLink>
```

Seam タグライブラリを使用する場合、 以下は等価です。

```
<h:outputLink value="main.jsf">
  <s:conversationId/>
  <h:outputText value="Continue"/>
</h:outputLink>
```

ポストバック用の対話コンテキストの伝播を無効にするコード例を以下に示します。

```
<h:commandLink action="main" value="Exit">
  <f:param name="conversationPropagation" value="none"/>
</h:commandLink>
```

以下は Seam タグライブラリと同等です。

```
<h:commandLink action="main" value="Exit">
  <s:conversationPropagation type="none"/>
</h:commandLink>
```

注記

対話コンテキストの伝播を無効にすることと、対話を終了することは同じでは**ありません**。

conversationPropagation 要求パラメータまたは **<s:conversationPropagation>** タグを使って対話の開始と終了を行う、またはネストされた対話を開始することができます。

```
<h:commandLink action="main" value="Exit">
  <s:conversationPropagation type="end"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Child">
  <s:conversationPropagation type="nested"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Hotel">
  <s:conversationPropagation type="begin"/>
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Hotel">
  <s:conversationPropagation type="join"/>
</h:commandLink>
```

この対話モデルにより、 マルチウィンドウ操作に正常に動作するアプリケーションの構築が容易になります。多くのアプリケーションに必要なのはこれだけです。 複雑なアプリケーションの中には以下の追加要件の両方あるいはどちらかを必要とするものがあります。

- 対話には、 連続的に実行したり同時に実行する多くの小さな単位のユーザーの操作も含まれます。より小さい **ネストされた対話** には単独の対話状態セットがあり、また外側の対話状態へのアクセスもあります。

- ユーザーは同じブラウザのウィンドウ内でいくつもの対話を切り換えることができます。 この機能は **ワークスペース管理** と呼ばれます。

8.2. ネストされた対話

ネストされた対話は、既存の対話のスコープ内で **`@Begin(nested=true)`** とマークされたメソッドを呼び出すことにより作成されます。 ネストされた対話にはそれ自体の対話コンテキストがありますが、外側の対話のコンテキストから値を読み取ることができます。外側の対話のコンテキストはネストされた対話内では読み取り専用ですが、オブジェクトは参照により取得されるため、オブジェクト自体への変更はその外側のコンテキストに反映されます。

- 対話をネストするとオリジナルの対話または外側の対話のコンテキストに積み重ねられるコンテキストを初期化します。 外側の対話が親とみなされます。

- ネストされた対話のコンテキストに直接設定されるまたはアウトジェクトされる値はすべて親となる対話のコンテキストでアクセス可能なオブジェクトに影響は与えません。

- 対話コンテキストからのコンテキスト検索やインジェクションはまず現在の対話コンテキストにある値を検索します。値が見付からないと対話がネストされている場合はその対話スタックまで続きます。この動作は上書き可能です。

その後 **`@End`** が出てくると、 ネストされた対話は破棄されて外側の対話が開始し、 対話スタックを **ポップ** します。 対話は任意の深さにネストすることができます。

特定のユーザーアクティビティ (ワークスペース管理や戻るボタン) により、 内側の対話が終了する前に外側の対話が開始されることがあります。 この場合、 同じ外側の対話に属する同時のネストされた対話を複数持つことが可能です。ネストされた対話が終了する前に外側の対話が終了すると、 Seam はネストされた対話コンテキストを外側のコンテキストと共にすべて破棄します。

対話スタックの最下位にある対話がルートの対話です。 この対話を破棄すると派生した対話はすべて常に破棄されます。 **`@End(root=true)`** を指定すると宣言的にこれを行うことができます。

対話は **継続可能な状態** と考えることができます。 ネストされた対話により、 ユーザーの操作のさまざまなポイントにおいてアプリケーションは一貫した継続可能な状態を捕らえることができます。 これにより、 戻るボタンを押したときやワークスペースの管理に対して正しく動作するようにします。

前述した通り、 現在ネストされている対話の親となる対話にコンポーネントが存在する場合、このネストされている対話は同じインスタンスを使用します。ネストされるそれぞれの対話内に別々のインスタンスを持たせると、親となる対話のコンポーネントインスタンスがその子となる対話からは見えなくなるため、時には便利な場合があります。これを行うには、コンポーネントに **`@PerNestedConversation`** アノテーションを付けます。

8.3. GET 要求を使った対話の開始

ページが Faces でない要求 (HTTP GET 要求など) 経由でアクセスされる場合、 JSF はトリガされるアクションリスナーを定義しません。これはユーザーがページをブックマークする、または **`<h:outputLink>`** からそのページに移動する場合に発生します。

ページがアクセスされたら直ちに対話を開始したい場合があります。 JSF アクションメソッドがないため、アクションに **@Begin** アノテーションを付けることはできません。

このページが状態をコンテキスト変数にフェッチする必要がある場合、 さらなる問題が発生します。すでに、この問題を解決する 2 つの方法を見てきました。 Seam コンポーネントにその状態が保持される場合、 **@Create** メソッドでその状態をフェッチできます。 状態が保持されていなければ、 コンテキスト変数に対して **@Factory** メソッドを定義することができます。

いずれの方法もうまくいかない場合、 Seam では **pages.xml** ファイルに **ページアクション** を定義することができます。

```
<pages>
  <page view-id="/messageList.jsp" action="#{messageManager.list}"/>
  ...
</pages>
```

レスポンス出力フェーズの始め、つまりページのレンダリング開始直前にこのアクションメソッドが呼び出されます。 ページアクションが null 以外の結果を返す場合、 Seam は適切な JSF および Seam ナビゲーションルールを処理するため、 まったく異なるページがレンダリングされることがあります。

ページのレンダリング前にしたいことが対話の開始 **だけ** の場合、組み込みアクションメソッドを次のように使用できます。

```
<pages>
  <page view-id="/messageList.jsp" action="#{conversation.begin}"/>
  ...
</pages>
```

また、 この組み込みアクションは JSF コントロールから呼び出すこともでき、 同様に **#{conversation.end}** を使って対話を終了します。

以下のように既存の対話への参加、 ネストした対話やページフロー、 アトミックな対話の開始などの制御に **<begin-conversation>** エレメントを使用することができます。

```
<pages>
  <page view-id="/messageList.jsp">
    <begin-conversation nested="true" pageflow="AddItem"/>
  <page>
  ...
</pages>
```

また、 **<end-conversation>** エレメントもあります。

```
<pages>
  <page view-id="/home.jsp">
    <end-conversation/>
  <page>
  ...
</pages>
```

これでページがアクセスされた直後に対話を開始できるオプションは 5 種類になりました。

- **@Create** メソッドに **@Begin** アノテーションを追加する

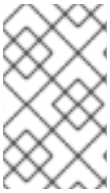
- **@Factory** メソッドに **@Begin** アノテーションを追加する
- Seam ページアクションメソッドに **@Begin** アノテーションを追加する
- **pages.xml** で **<begin-conversation>** を使用する
- **#{conversation.begin}** を Seam ページアクションメソッドとして使用する

8.4. 長期実行の対話の必要

ページの中には長期実行の対話のコンテキストにのみ関連している特定のページがあります。 このようなページへのアクセスを制限する方法のひとつとして、長期実行の対話の存在がレンダリングされているページの必須条件とする方法があります。

Seam のページ記述子には **conversation-required** 属性があり、 ページのレンダリングが行われるには現在の対話が長期実行で (またはネストされている) なければならないことを示すことができます。

```
<page view-id="/book.xhtml" conversation-required="true"/>
```



注記

現時点では、どの長期実行の対話が必要かを示すことはできませんが、ページアクション内の対話に特定の値が存在しているかどうかを確認することで基本的な承認を構築することができます。

長期実行の対話が存在しないがページが要求されたことを Seam が確定すると次のアクションが実行されます。

- **org.jboss.seam.noConversation** というコンテキスト依存イベントを発生させます。
- **org.jboss.seam.NoConversation** バンドルキーを持つ警告ステータスのメッセージを登録します。
- 次のように、 **no-conversation-view-id** 属性で定義されている場合はユーザーを代替となるページにリダイレクトします。

  ```
  <pages no-conversation-view-id="/main.xhtml"/>
  ```

  このページはアプリケーション全体で使用されます。 現在、 代替ページを複数定義することはできません。

8.5. <S:LINK> と <S:BUTTON> の使用

JSF コマンドリンクは常に JavaScript でフォームサブミットを行います。 これによりウェブブラウザの「新しいウィンドウで開く」または「新しいタブで開く」機能が動作しない問題が発生します。純粋な JSF でこの機能が必要な場合は、 **<h:outputLink>** を使用する必要があります。 ただし、 このメソッドには重要な制限が 2 つあります。

- JSF にはアクションリスナーを **<h:outputLink>** につなげる方法は備わっていません。
- また、実際のフォームサブミットがないため JSF は選択された **DataModel** の行を伝播しません。

Seam は **ページアクション** という概念で 1 番目の問題を解決しますが、 2 番目の問題は解決しません。 要求パラメータを渡しサーバー側で選択されたオブジェクトを再度クエリすることでこの問題に対処することは可能です。いくつかのケースでは (Seam ブログのサンプルアプリケーションなど) これが最善策となります。 これは RESTful でありサーバー側の状態を必要としないためブックマーク機能に対応します。ブックマークを必要としない他のケースでは **@DataModel** と **@DataModelSelection** が透過的かつ便利です。

この機能を補ってさらに対話伝播をより簡略化するために、 Seam は **<s:link>** JSF タグを提供します。

このリンクは JSF ID だけ指定できます。

```
<s:link view="/login.xhtml" value="Login"/>
```

また、 アクションメソッドを指定することもできます。 この場合アクションの結果は最終的なページを確定します。

```
<s:link action="#{login.logout}" value="Logout"/>
```

JSF ビュー ID とアクションメソッドの両方を指定すると、 アクションメソッドが null 以外の結果を返さない限りそのビューが使用されます。

```
<s:link view="/loggedOut.xhtml"  action="#{login.logout}" value="Logout"/>
```

リンクは **<h:dataTable>** 内で使用する **DataModel** の選択された行を自動的に伝播します。

```
<s:link view="/hotel.xhtml" action="#{hotelSearch.selectHotel}"
   value="#{hotel.name}"/>
```

既存の対話のスコープを残しておくことができます。

```
<s:link view="/main.xhtml" propagation="none"/>
```

対話を開始、 終了、 またはネストすることができます。

```
<s:link action="#{issueEditor.viewComment}" propagation="nest"/>
```

リンクが対話を開始すると、ページプローの使用を指定することができます。

```
<s:link action="#{documentEditor.getDocument}" propagation="begin"
   pageflow="EditDocument"/>
```

以下のように **taskInstance** 属性は jBPM タスクリストで使用します。例は 「Seam と jBPM を使ったアプリケーションの全容 : DVD ストアサンプル」 を参照してください。

```
<s:link action="#{documentApproval.approveOrReject}"
   taskInstance="#{task}"/>
```

最後に「リンク」をボタンとしてレンダリングさせたい場合は **<s:button>** を使用します。

```
<s:button action="#{login.logout}" value="Logout"/>
```

8.6. 成功のメッセージ

動作が成功したか失敗したかをユーザーに知らせるために、通常メッセージが表示されます。この機能には、 JSF **FacesMessage** を使うと便利です。ただし、成功のアクションは多くの場合ブラウザリダイレクトを必要とします。JSF はリダイレクト全体に Faces のメッセージを伝播しないため、純粋な JSF で成功のメッセージを表示するのは困難です。

組み込み対話のスコープされた Seam コンポーネントである **facesMessages** がこの問題を解決します (これには Seam リダイレクトフィルタが必要です)。

```
@Name("editDocumentAction")
@Stateless
    public class EditDocumentBean implements EditDocument {
    @In EntityManager em;
    @In Document document;
    @In FacesMessages facesMessages;

    public String update() {
        em.merge(document);
        facesMessages.add("Document updated");
    }
}
```

メッセージが **facesMessages** に追加されると、次のレスポンス出力フェーズで現在の対話に対して使用されます。Seam はリダイレクト全体で一時的な対話コンテキストも維持するため、長期実行の対話がなくても機能します。

JSF EL 式を Faces メッセージサマリーに含めることもできます。

```
facesMessages.add("Document #{document.title} was updated");
```

メッセージは通常通りに表示されます。

```
<h:messages globalOnly="true"/>
```

8.7. ナチュラル対話の ID

永続オブジェクトを処理する対話を作業する場合に、 標準の「サロゲート」対話 ID ではなくそのオブジェクトのナチュラルビジネスキーを使用するのにはいくつか理由があります。

既存の対話に容易にリダイレクトできる

ユーザーが同じ動作を 2 度要求した場合、 既存の対話にリダイレクトさせると便利なことがあります。 たとえば次のような状況の場合です。

ebay で両親へのクリスマスプレゼントを購入しようとしているとします。これを両親に直接郵送しようと思っています。支払い詳細は入力しましたが両親の住所を思い出すことができません。 住所を探している間に誤って同じブラウザウィンドウを使ってしまいました。 もう一度先ほどのプレゼントの支払いの場所に戻る必要があります。

ナチュラル対話を使うと、 ユーザーは前回の対話に参加して中断したところから簡単に始めることができます。 この例の場合、 対話 ID の itemId を持つ payForItem 対話に再度参加できます。

わかりやすい URL

わかりやすい URL は重要であり (ページの内容を ID 番号を使わずにわかりやすく参照可能)、 編集可能な階層型になっています (ユーザーは URL を編集して目的のページに行くことが可能)。

ナチュラル対話では、 アプリケーションに複雑で長い URL を生成させながら URLRewrite を使ってユーザーにはシンプルで憶えやすい URL を表示することができます。 ホテル予約の例の場合、**http://seam-hotels/book.seam?hotel=BestWesternAntwerpen** は **http://seam-hotels/book/BestWesternAntwerpen** と書き換えられるため、 非常にわかりやすくなります。 URLRewrite はパラメータに依存する点に注意してください。 前のサンプルの **hotel** はドメインモデルで一意のパラメータにマッピングを行わなければなりません。

8.8. ナチュラル対話の作成

ナチュラル対話は **pages.xml** で定義されます。

```
<conversation name="PlaceBid" parameter-name="auctionId"
              parameter-value="#{auction.auctionId}"/>
```

上記の定義でまず注意する点は対話の名前です。 この場合は **PlaceBid** です。 対話名は一意的にこの特定の名前が付いた対話を識別し、 **page** 定義を使用して参加する名前の付いた対話を識別します。

属性 **parameter-name** はナチュラル対話 ID を保持してデフォルトの対話 ID パラメータを置換する要求パラメータを定義します。 この例では **parameter-name** は **auctionId** です。 つまり、 ページの URL 内に **cid=123** のような対話パラメータではなく **auctionId=765432** を含むようになります。

最後の属性 **parameter-value** は対話 ID として使用するナチュラルビジネスキーの値の評価に使用される EL 式を定義します。 この例では対話 ID が現在スコープ内にある **auction** インスタンスの主キーの値になります。

次に、名前の付いた対話に参加しているページを定義します。 **page** 定義の **conversation** 属性を指定することで実行できます。

```
<page view-id="/bid.xhtml" conversation="PlaceBid" login-required="true">
  <navigation from-action="#{bidAction.confirmBid}">
    <rule if-outcome="success">
      <redirect view-id="/auction.xhtml">
        <param name="id" value="#{bidAction.bid.auction.auctionId}"/>
      </redirect>
    </rule>
  </navigation>
</page>
```

8.9. ナチュラル対話へのリダイレクト

ナチュラル対話を開始またはリダイレクトする場合、 ナチュラル対話名を指定する方法はいくつかあります。まずは次のページ定義を見てみましょう。

```
<page view-id="/auction.xhtml">
  <param name="id" value="#{auctionDetail.selectedAuctionId}"/>
  <navigation from-action="#{bidAction.placeBid}">
    <redirect view-id="/bid.xhtml"/>
  </navigation>
</page>
```

ここでは、 **#{bidAction.placeBid}** を呼び出すことによりナチュラル対話 ID **PlaceBid** で設定された **/bid.xhtml** にリダイレクトされるのがわかります。 アクションメソッドの宣言は以下のようになります。

```
@Begin(join = true)
public void placeBid()
```

名前が付いた対話が **<page/>** エレメントで指定されると、 その名前が付いた対話へのリダイレクトはアクションメソッドの呼び出しに続いてナビゲーションルールの一部として発生します。既存の対話にリダイレクトする場合は、これが問題となることがあります。アクションメソッドが呼び出される前にリダイレクトが発生する必要があるためです。したがってアクションが呼び出される前に対話名を指定する必要があります。これを行う方法のひとつとして **s:conversationName** タグの使用があります。

```
<h:commandButton id="placeBidWithAmount" styleClass="placeBid"
   action="#{bidAction.placeBid}">
  <s:conversationName value="PlaceBid"/>
</h:commandButton>
```

また、 **s:link** または **s:button** のいずれかに **conversationName** 属性を指定することもできます。

```
<s:link value="Place Bid" action="#{bidAction.placeBid}"
   conversationName="PlaceBid"/>
```

8.10. ワークスペースの管理

ワークスペース管理では、1つのウィンドウで複数の対話を「切り換える」ことができます。 Seam のワークスペース管理は Java レベルで完全に透過的です。 次のようにしてワークスペース管理を有効にします。

- それぞれのビュー ID (JSF または Seam ナビゲーションルールを使用する場合) またはページノード (jPDL ページフローを使用する場合) に **記述** テキストを入力します。ワークスペースを切り替えることで、この記述テキストをユーザーに表示します。
- ページの中に1つ以上のワークスペース切り替え JSP または Facelets の一部を含ませます。標準の断片はドロップダウンメニュー、 対話のリスト、「ブレッドクラム」を通じてワークスペース管理をサポートします。

8.10.1. ワークスペース管理と JSF ナビゲーション

JSF または Seam ナビゲーションルールが使用される場合、 Seam は対話の現在の **view-id** を復元してその対話に切り替えます。 ワークスペースの記述テキストは **pages.xml** と呼ばれるファイルで定義され、 Seam はこのファイルが **WEB-INF** ディレクトリ内に **faces-config.xml** と共に配置されているようにします。

```
<pages>
  <page view-id="/main.xhtml">
    <description>Search hotels: #{hotelBooking.searchString}</description>
  </page>
  <page view-id="/hotel.xhtml">
    <description>View hotel: #{hotel.name}</description>
  </page>
```

```
  <page view-id="/book.xhtml">
    <description>Book hotel: #{hotel.name}</description>
  </page>
  <page view-id="/confirm.xhtml">
    <description>Confirm: #{booking.description}</description>
  </page>
</pages>
```

注記

Seam アプリケーションはこのファイルがなくても動作しますが、 ワークスペースの切り替えは利用できなくなります。

8.10.2. ワークスペース管理と jPDL ページフロー

jPDL ページフロー定義を使う場合、 Seam は現在の jBPM のプロセス状態を復元することによって特定の対話に切り替えます。 同じ **view-id** に現在の **<page>** ノードに応じて異なる記述を持つことができるためより柔軟なモデルとなります。記述テキストは **<page>** ノードで定義されます。

```
<pageflow-definition name="shopping">
  <start-state name="start">
    <transition to="browse"/>
  </start-state>
  <page name="browse" view-id="/browse.xhtml">
    <description>DVD Search: #{search.searchPattern}</description>
    <transition to="browse"/>
    <transition name="checkout" to="checkout"/>
  </page>
  <page name="checkout" view-id="/checkout.xhtml">
    <description>Purchase: $#{cart.total}</description>
    <transition to="checkout"/>
    <transition name="complete" to="complete"/>
  </page>
  <page name="complete" view-id="/complete.xhtml">
    <end-conversation />
  </page>
</pageflow-definition>
```

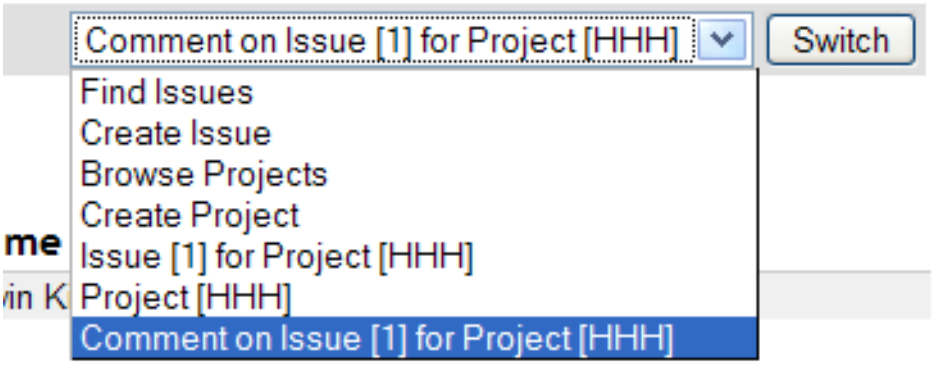
8.10.3. 対話切り替え

次の一部を JSP または Facelets のページに含めることで、 現在の対話またはアプリケーションの他のページに切り替えられるドロップダウンメニューを取得します。

```
<h:selectOneMenu value="#{switcher.conversationIdOrOutcome}">
  <f:selectItem itemLabel="Find Issues" itemValue="findIssue"/>
  <f:selectItem itemLabel="Create Issue" itemValue="editIssue"/>
  <f:selectItems value="#{switcher.selectItems}"/>
</h:selectOneMenu>
<h:commandButton action="#{switcher.select}" value="Switch"/>
```

この例には、 各対話に 1 アイテムを含むメニューに加えて、 ユーザーに別の対話を開始させる 2 つの追加アイテムがあります。

詳細が書かれた対話 (**pages.xml** で指定) のみがドロップダウンメニューに含まれます。



8.10.4. 対話一覧

対話一覧は対話切り替えに似ていますが、 表形式で表示される点が異なります。

```
<h:dataTable value="#{conversationList}" var="entry"
   rendered="#{not empty conversationList}">
  <h:column>
    <f:facet name="header">Workspace</f:facet>
    <h:commandLink action="#{entry.select}" value="#{entry.description}"/>
    <h:outputText value="[current]" rendered="#{entry.current}"/>
  </h:column>
  <h:column>
    <f:facet name="header">Activity</f:facet>
    <h:outputText value="#{entry.startDatetime}">
      <f:convertDateTime type="time" pattern="hh:mm a"/>
    </h:outputText>
    <h:outputText value=" - "/>
    <h:outputText value="#{entry.lastDatetime}">
      <f:convertDateTime type="time" pattern="hh:mm a"/>
    </h:outputText>
  </h:column>
  <h:column>
    <f:facet name="header">Action</f:facet>
    <h:commandButton action="#{entry.select}" value="#{msg.Switch}"/>
    <h:commandButton action="#{entry.destroy}" value="#{msg.Destroy}"/>
  </h:column>
</h:dataTable>
```

ご使用のアプリケーションに合うようカスタマイズ可能です。

Workspace	Workspace activity	Action
Comment on Issue [1] for Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy
Issue [1] for Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy
Project [HHH]	01:18 PM - 01:18 PM	Switch Destroy

詳細を持つ対話のみがこの一覧に含まれます。

対話一覧によりユーザーはワークスペースを破棄できる点に注意してください。

8.10.5. ブレッドクラム

ブレッドクラムは、 現在の対話スタック内の対話へのリンクの一覧です。 ネストされた対話モデルを使うアプリケーションで役立ちます。

```
<ui:repeat value="#{conversationStack}" var="entry">
  <h:outputText value=" | "/>
  <h:commandLink value="#{entry.description}" action="#{entry.select}"/>
</ui:repeat>
```

8.11. 対話型コンポーネントと JSF コンポーネントのバインディング

対話型コンポーネントには、JSF コンポーネントへのバインディングの保持には使用できないという小さな制限があります (一般的には、 アプリケーションロジックからビューに強い依存関係を作成するため、必ず必要でない限りこの JSF の機能は使用しないことが一般的に推奨されます)。ポストバック要求では、 Seam 対話コンテキストが復元される前、ビュー復元フェーズ中にコンポーネントのバインディングが更新されます。

これに対処するには、 イベントスコープのコンポーネントを使ってコンポーネントバインディングを格納し、必要とする対話スコープのコンポーネントにそれをインジェクトします。

```
@Name("grid")
@Scope(ScopeType.EVENT)
public class Grid {
    private HtmlPanelGrid htmlPanelGrid; // getters and setters
    ...
}
```

```
@Name("gridEditor")
@Scope(ScopeType.CONVERSATION)
public class GridEditor {
    @In(required=false)
    private Grid grid;
    ...
}
```

また別の制限として、対話スコープのコンポーネントをバインドされた JSF コントロールでイベントスコープのコンポーネントにインジェクトできない点があります。これには **facesMessages** のような Seam の組み込みコンポーネントが含まれます。

暗黙の **uiComponent** ハンドルで JSF コンポーネントツリーにアクセスすることもできます。 次の例では反復中にデータテーブルを支える **UIData** コンポーネントの **getRowIndex()** にアクセスし、 現在の行番号を表示します。

```
<h:dataTable id="lineItemTable" var="lineItem"
   value="#{orderHome.lineItems}">
  <h:column>
    Row: #{uiComponent[&#39;lineItemTable&#39;].rowIndex}
```

```
    </h:column>
    ...
</h:dataTable>
```

このマップでは、JSF UI のコンポーネントはクライアント識別子で使用可能です。

8.12. 対話型コンポーネントへの同時呼び出し

Seam コンポーネントへの同時呼び出しに関する一般的な説明は 「同時実行モデル」 でご覧ください。本項では、同時実行が発生する最も一般的な状況について説明します (AJAX 要求から対話型コンポーネントにアクセスする場合)。また、クライアントで発生したイベントの制御に Ajax クライアントライブラリが提供するオプションについて説明してから RichFaces で提供されるオプションについて見ていきます。

対話型コンポーネントでは実際の同時アクセスは許可されないため、 Seam は各要求を連続的に処理するようそれぞれを待ち行列に入れます。 これにより各要求は確定的に実行されます。 ただし、シンプルなキューにはいくつか制限があります。なんらかの理由でメソッドが完了するまでに時間がかかる場合、 クライアントが要求を生成するたびにそれを実行すると DoS 攻撃を招く恐れがあります。多くの場合、AJAX を使用してユーザーにステータスのクィックアップデートを提供するため、アクションを長時間実行し続けるのは実用的ではありません。

したがって、 長期実行の対話の内側で作業する場合は Seam は一定期間アクションイベントを待ち行列に入れます (同時要求タイムアウト)。 タイムアウトまでに Seam がイベントを処理できないと一時的な対話を作成してユーザーにタイムアウトを知らせるメッセージを表示します。このため、 AJAX イベントでサーバーを溢れさせないようにすることが重要です。

components.xml で同時要求のタイムアウトに (ミリ秒単位で) 適切なデフォルトを設定することができます。

```
<core:manager concurrent-request-timeout="500" />
```

また、 ページごとに同時要求のタイムアウトを調整することもできます。

```
<page view-id="/book.xhtml" conversation-required="true"
      login-required="true" concurrent-request-timeout="2000" />
```

ここまではユーザーに対して連続的に出現する AJAX 要求について説明してきました。 クライアントはサーバーにイベントが発生したことを伝え、 その結果に応じてページの一部を再レンダリングします。 この方法は AJAX 要求が軽量である場合は十分ですが (1 列内の数字の合計を計算するなど呼び出されるメソッドがシンプルである場合)、 計算が複雑となる場合には別の方法が必要です。

クライアントがサーバーに AJAX 要求を送信しこれによりサーバーで非同期にアクションが直ちに開始するような場合にはポーリングベースの方法を使用してください。 アクションが実行されている間、クライアントは更新に対しサーバーをポーリングします。 長期実行のアクションの連続でいずれのアクションもタイムアウトさせないようにすることが重要な場合はこの方法を使用した方が賢明です。

8.12.1. 対話型 AJAX アプリケーションを設計する方法

まず、より簡単な「連続」要求の方法とポーリングの方法のどちらを使用するのかを決める必要があります。

連続 要求を選択する場合は、 要求が完了するまでに要される時間を推測する必要があります。前項で説明したようにこのページに対する同時要求のタイムアウトを変更する必要があるかもしれません。要求がサーバーを溢れさせないようにするためサーバー側での行列待ちがおそらく必要となります。 イベ

ントが頻繁に発生し (入力フィールドの **keypress** や **onblur** など) クライアントの即時更新が優先事項ではない場合は、クライアント側で要求の遅延を設定してください。 要求遅延の作業を行う場合、サーバー側でも行列待ちできることを考慮に入れてください。

最後に、クライアントライブラリは未完了の重複要求を最新のものは後に残してすべて停止するオプションを備えています。

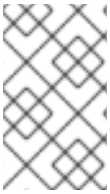
ポーリングの方法を使用する場合は細かな調整はあまり必要ありません。 アクションメソッド **@Asynchronous** をマークしてポーリングの間隔を決定するだけです。

```
int total;

// This method is called when an event occurs on the client
// It takes a really long time to execute
@Asynchronous
public void calculateTotal() {
    total = someReallyComplicatedCalculation();
}

// This method is called as the result of the poll
// It's very quick to execute
public int getTotal() {
    return total;
}
```

8.12.2. エラー処理

同時要求を対話型コンポーネントに対して行列待ちに入れるよう十分注意をしてアプリケーションを設計しても、 サーバーがオーバーロードとなる危険性はあります。 オーバーロードが発生すると **concurrent-request-timeout** の期限が切れるまでに全要求は処理されなくなります。 こうした場合、 Seam は **ConcurrentRequestTimeoutException** を送出します。これは **pages.xml** で処理されます。 HTTP 503 エラーの送信が推奨されます。

```
<exception class="org.jboss.seam.ConcurrentRequestTimeoutException"
           log-level="trace">
  <http-error error-code="503" />
</exception>
```



注記

現在サーバーは一時的な過負荷またはサーバーメンテナンスのため要求を処理することができません。これは一時的な状態で、しばらく待つと緩和されることを意味しています。

代わりにエラーページにリダイレクトすることができます。

```
<exception class="org.jboss.seam.ConcurrentRequestTimeoutException"
           log-level="trace">
  <end-conversation/>
  <redirect view-id="/error.xhtml">
    <message>
      The server is too busy to process your request,
      please try again later
```

```
    </message>
  </redirect>
</exception>
```

ICEfaces、 RichFaces Ajax、 Seam Remoting はいずれも HTTP エラーコードを処理することができます。 Seam Remoting は HTTP エラーを表示するダイアログボックスを出現させます。ICEfaces はエラーをその接続状態のコンポーネント内に表示します。 RichFaces はユーザー定義が可能なコールバックで HTTP エラーを処理するのに最も完全な対応を行います。 たとえば、 エラーメッセージをユーザーに表示するには以下のようにします。

```
<script type="text/javascript">
  A4J.AJAX.onError = function(req,status,message) {
    alert("An error occurred");
  };
</script>
```

エラーコード以外で、 セッションがタイムアウトしたことによりビューの有効期限切れをサーバーが報告する場合は RichFaces で別のコールバック機能を使用します。

```
<script type="text/javascript">
  A4J.AJAX.onExpired = function(loc,message) {
    alert("View expired");
  };
</script>
```

別の方法として、RichFaces にエラーを処理させることもできます。この場合、 ユーザーは「ビューの状態を復元できませんでした、ページを再ロードしますか?」というプロンプトを受け取ります。 このメッセージをグローバルにカスタマイズするには、アプリケーションのリソースバンドルで次のメッセージキーを設定します。

```
AJAX_VIEW_EXPIRED=View expired. Please reload the page.
```

8.12.3. RichFaces (Ajax4jsf)

RichFaces (Ajax4jsf) は Seam で最もよく使用される AJAX ライブラリで、前項で説明したすべての制御を提供します。

`eventsQueue`

イベントが置かれる待ち行列を提供します。 イベントはすべて待ち行列に入れられ要求がサーバーに連続的に送られます。サーバーが溢れるのを防ぐため、サーバーに対する要求の実行に一定の時間がかかる場合に便利です (重量のある計算で低速のソースから情報を取得する場合など)。

`ignoreDupResponses`

より新しい「同様な」要求がすでに行列待ちにある場合はこの要求によって生成された応答を無視します。 **`ignoreDupResponses="true"`** はサーバー側の要求処理を取り消す **わけではありません**。 クライアント側での不要な更新を防ぐだけです。

このオプションは複数の同時要求を許可するため、 Seam の対話で使用する場合は注意が必要です。

`requestDelay`

要求がキューに残っている時間をミリ秒単位で定義します。要求がこの時点で処理されていない場合は、要求は送信されるか (応答が受信されているかどうかにかかわらず) 破棄 (より新しい「同様」

のイベントがキューにある場合) されます。

このオプションは複数の同時要求を許可するため、 Seam の対話で使用する場合は注意が必要です。 アクションが実行に要する時間より設定する遅延 (同時要求のタイムアウトと併用) の方が長くなければなりません。

```
<a:poll reRender="total" interval="1000" />
```

必要に応じてサーバーにポーリングし、エリアを再レンダリングします。

第9章 ページフローとビジネスプロセス

JBoss jBPM は Java SE や EE 環境向けのビジネスプロセス管理エンジンです。 jBPM はビジネスプロセスやユーザーインタラクションを待ち状態やデシジョン、 タスク、 Web ページなどを表すノードのグラフとして表現します。 グラフは jPDL と呼ばれるシンプルで非常に読みやすい XML 言語で定義され、 Eclipse プラグインを利用してグラフィカルに編集や表示を行うことができます。 jPDL は拡張可能な言語であり、 WEB アプリケーションのページフローの定義から典型的なワークフローの管理、 また SOA 環境におけるサービスのオーケストレーションまで幅広く対応します。

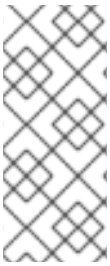
Seam アプリケーションは、複雑なユーザーインタラクションでページフローの定義を行う、 また包括的なビジネスプロセスの定義を行うという 2 種類の異なる問題に jBPM を使用します。

前者の場合、 jPDL プロセス定義は単一の対話に対してページフローを定義するのに対し、Seam の対話は 1 ユーザーとの比較的実行期間が短いインタラクションであると考えられます。

後者の場合、 ビジネスプロセスは複数ユーザーの複数の対話にまたがることがあります。 その状態は jBPM データベースでは永続であるため、長期実行とみなされます。 複数ユーザーのアクティビティの調整は、 単一ユーザーとのインタラクションについて動作を記述するより複雑となるため、 jBPM は複数の並列実行パスやタスクを管理するための高度な機能を提供します。

注記

ページフローと包括的なビジネスプロセスは混同しないようにしてください。 異なる粒度レベルで動作します。 ページフロー、 対話、 タスクはすべて単一ユーザーとの単一の操作となります。 ビジネスプロセスは多くのタスクにまたがります。 また、 この 2 つの jBPM アプリケーションは互いに依存性がないので、 一緒に使用することも別々に使用することもでき、 まったく使用しなくても構いません。



注記

Seam を使用する上で jPDL の知識は必要ありません。 JSF や Seam ナビゲーションルールでページフローを定義し、 アプリケーションがプロセス駆動というよりデータ駆動となる場合は恐らく jBPM は必要ありません。 ただし、 明確に定義されたグラフィカルな表現という観点からユーザーによる操作を考えるとより堅牢なアプリケーションの構築に役立ちます。

9.1. SEAM のページフロー

Seam にはページフローを定義する 2 つの方法があります。

- JSF あるいは Seam ナビゲーションルールを使用します - **ステートレスなナビゲーションモデル**
- jPDL を使用します - **ステートフルなナビゲーションモデル**

簡単なアプリケーションではステートレスなナビゲーションモデルで十分です。 複雑なアプリケーションになる場合は両方を組み合わせて使用します。 各モデルはそれぞれに長所と短所があるのでそれらを考慮に入れ実装を行ってください。

9.1.1. 2 種類のナビゲーションモデル

ステートレスなモデルは一組の名前の付いた論理的なイベントの結果から直接、結果として生じるビューページへのマッピングを定義します。 ナビゲーションルールはイベントの発生源となるページ以外、 アプリケーションで保持される状態はすべて無視します。 したがって、 アクションリスナーの

メソッドしかアプリケーションの現在の状態にアクセスできないため、このアクションリスナーのメソッドがページフローに関する決定を行わなければならない場合があります。

これは JSF ナビゲーションルールを使用したページフロー定義の例です。

```
<navigation-rule>
  <from-view-id>/numberGuess.jsp</from-view-id>

  <navigation-case>
    <from-outcome>guess</from-outcome>
    <to-view-id>/numberGuess.jsp</to-view-id>
    <redirect/>
  </navigation-case>

  <navigation-case>
    <from-outcome>win</from-outcome>
    <to-view-id>/win.jsp</to-view-id>
    <redirect/>
  </navigation-case>

  <navigation-case>
    <from-outcome>lose</from-outcome>
    <to-view-id>/lose.jsp</to-view-id>
    <redirect/>
  </navigation-case>

</navigation-rule>
```

これは Seam ナビゲーションルールを使用した同じページフロー定義の例です。

```
<page view-id="/numberGuess.jsp">
  <navigation>

    <rule if-outcome="guess">
      <redirect view-id="/numberGuess.jsp"/>
    </rule>

    <rule if-outcome="win">
      <redirect view-id="/win.jsp"/>
    </rule>

    <rule if-outcome="lose">
      <redirect view-id="/lose.jsp"/>
    </rule>

  </navigation>
</page>
```

ナビゲーションルールが冗長過ぎると感じる場合は、アクションリスナーのメソッドから直接ビュー ID を返すことができます。

```
public String guess() {
  if (guess==randomNumber) return "/win.jsp";
  if (++guessCount==maxGuesses) return "/lose.jsp";
```

```
  return null;
}
```

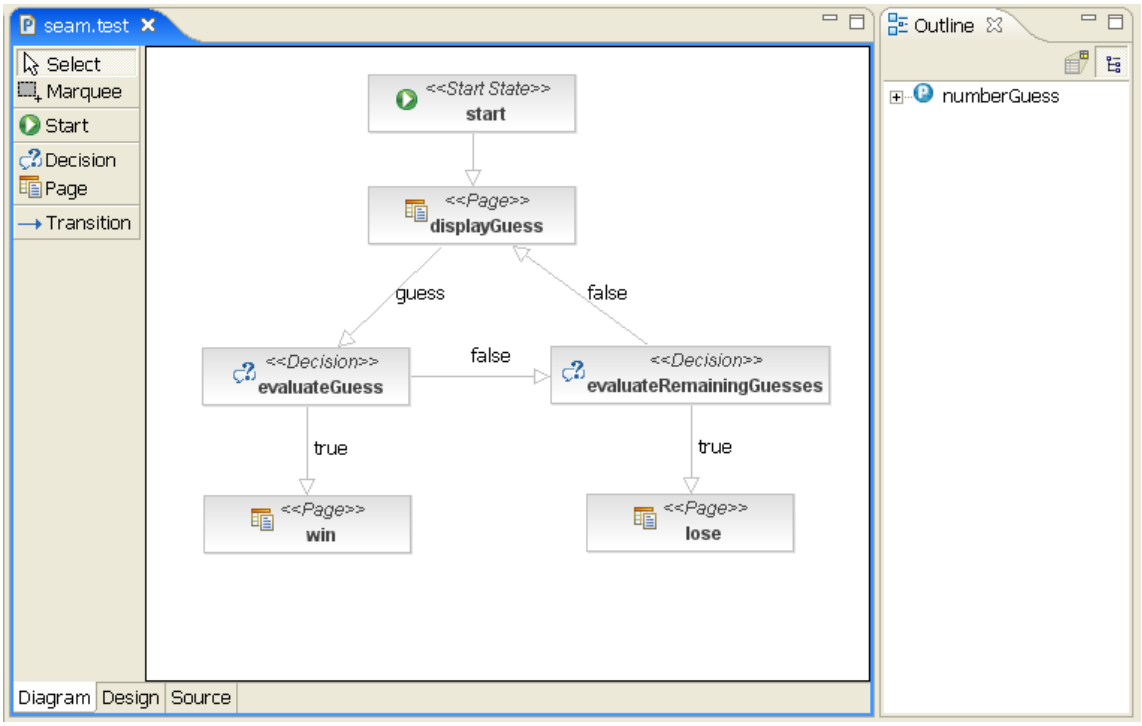
これはリダイレクトとなるので注意してください。 リダイレクトで使用するパラメータを指定することもできます。

```
public String search() {
  return "/searchResults.jsp?searchPattern=#{searchAction.searchPattern}";
}
```

ステートフルなモデルは名前の付いた論理的なアプリケーションの状態間で起こる遷移の一式を定義します。 このモデルでは jPDL ページフロー定義中にあらゆるユーザー操作のフロー表現が可能となるため、 インタラクションのフローを全く認識しないアクションリスナーのメソッドを記述することができます。

これは jPDL を使用したページフロー定義の例です。

```
<pageflow-definition name="numberGuess">
  <start-page name="displayGuess" view-id="/numberGuess.jsp">
    <redirect/>
    <transition name="guess" to="evaluateGuess">
      <action expression="#{numberGuess.guess}" />
    </transition>
  </start-page>

  <decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
    <transition name="true" to="win"/>
    <transition name="false" to="evaluateRemainingGuesses"/>
  </decision>

  <decision name="evaluateRemainingGuesses"
            expression="#{numberGuess.lastGuess}">
    <transition name="true" to="lose"/>
    <transition name="false" to="displayGuess"/>
  </decision>

  <page name="win" view-id="/win.jsp">
    <redirect/>
    <end-conversation />
  </page>

  <page name="lose" view-id="/lose.jsp">
    <redirect/>
    <end-conversation />
  </page>

</pageflow-definition>
```



ここで、 すぐ気付くことが 2 つあります。

- JSF と Seam ナビゲーションルールは非常にシンプルです。 (しかし、 根底となる Java コードはより複雑であるという事実を隠しています。)
- jPDL によりユーザー操作がとたんに理解しやすくなるため、 JSP や Java コードを見る必要性がなくなります。

また、 ステートフルモデルはさらに制約的です。 それぞれの論理的な状態 (ページフローの各ステップ) に対して他の状態に遷移可能な制約された一式があります。 ステートレスモデルは **アドホックな** モデルとなるため、 アプリケーションではなくユーザーが次に行きたいところを決めるような比較的制約のない自由な操作に適しています。

ステートフルとステートレスのナビゲーションの違いは、 モーダルのビューと非モーダルのビューによく似ています。 Seam のアプリケーションは単純な意味では通常はモーダルではありません。 実際、 モーダルな動作を回避するために対話を使用します。 ただし、 Seam アプリケーションは任意の対話のレベルではモーダルとなる場合があり、 また頻繁にモーダルとなります。 ユーザーが行う操作の順番を予測することは非常に困難であるためモーダルな動作は避けるのが最適ですが、 ステートフルモデルでは存在意義があります。

2 つのモデルの最大の違いは戻るボタンの動作です。

9.1.2. Seam と戻るボタン

JSF あるいは Seam ナビゲーションルールが使用される場合、 ユーザーは戻るボタン、 進むボタン、 更新ボタンを使って自由な操作を行うことができます。 内部的な対話状態の一貫性を保持するのがアプリケーションの役割です。 開発者は Web アプリケーションのフレームワークやステートレスなコンポーネントのモデルを扱う経験を通してこれがいかに困難であるかを学んできました。 ステートフルなセッション Bean で支えられる明確に定義された対話モデルとなるような Seam ではこれが非常に簡単になります。 通常、 アクションリスナーのメソッドの冒頭で **`no-conversation-view-id`** を null チェックと組合わせるだけです。 大抵望まれるものは自由なナビゲーションへの対応となります。

この場合、 **`no-conversation-view-id`** の宣言は **`pages.xml`** で行います。 現在は存在していない対話で表示されたページからの要求の場合には別のページにリダイレクトを行うよう指示します。

```
<page view-id="/checkout.xhtml" no-conversation-view-id="/main.xhtml"/>
```

一方、ステートフルモデルでは戻るボタンを押すと前の状態に戻る未定義の遷移と解釈されます。ステートフルモデルは現在の状態から定義された遷移セットを強制実行するため、戻るボタンはステートフルモデルではデフォルトで許可されません。Seam は透過的に戻るボタンの使用を検出して前の「古い」ページのアクション試行をブロックして、ユーザーを「現在の」ページにリダイレクトします (そして Faces メッセージを表示)。開発者にとってはこれはステートフルモデルの特長になりますが、ユーザーにとってはイライラさせられることがあります。**back="enabled"** を設定すると特定のページノードからの戻るボタン操作を許可することができます。

```
<page name="checkout" view-id="/checkout.xhtml" back="enabled">
  <redirect/>
  <transition to="checkout"/>
  <transition name="complete" to="complete"/>
</page>
```

これは、**checkout** 状態から以前のどの状態にでも戻るボタンでの移動が可能です。

注記

遷移の後にリダイレクトを行うようページ設定されている場合、流れの後半のページで戻るを有効にしていても戻るボタンでユーザーがそのページに戻ることはできません。Seam はページスコープにページフローに関する情報を格納し、戻るボタンは復元されるその情報の POST にならなければならないためです (Faces 要求を通じてなど)。リダイレクトはこのリンクを提供します。

ページフローの間にレンダリングされたページから要求が発生し、そのページフローを持つ対話がすでに存在していない場合は、何が起こるかを定義しなければなりません。この場合 **no-conversation-view-id** 宣言はページフロー定義で行います。

```
<page name="checkout" view-id="/checkout.xhtml" back="enabled"
      no-conversation-view-id="/main.xhtml">
  <redirect/>
  <transition to="checkout"/>
  <transition name="complete" to="complete"/>
</page>
```

実際にはいずれのナビゲーションモデルとも使い道があります。どんなときにどちらのモデルの方が適切かを理解するために、これから簡単に学んでいきます。

9.2. JPDL ページフローの使用

9.2.1. ページフローのインストール

Seam の jBPM 関連のコンポーネントをインストールし、Seam アーカイブ (**seam.properties** ファイルを含むアーカイブ) の中にページフローの定義 (標準の **.jpdl.xml** 拡張を使用) を配置する必要があります。

```
<bpm:jbpm />
```

ページフロー定義を検索する場所を明示的に Seam に指示することもできます。ここでは **components.xml** で指定します。

```
<bpm:jbpm>
  <bpm:pageflow-definitions>
    <value>pageflow.jpdl.xml</value>
  </bpm:pageflow-definitions>
</bpm:jbpm>
```

9.2.2. ページフローの開始

@Begin、 **@BeginTask** または **@StartTask** のアノテーションを使用してプロセス定義の名前を指定し jPDL ベースのページフローを「開始」します。

```
@Begin(pageflow="numberguess") public void begin() { ... }
```

もしくは、 **pages.xml** を使用してページフローを開始できます。

```
<page>
  <begin-conversation pageflow="numberguess"/>
</page>
```

RENDER_RESPONSE フェーズ中 – 例えば、 **@Factory** または **@Create** メソッド中 – にページフローを開始している場合は、ページは既にレンダリングされたとして考え **<start-page>** ノードを上記の例のようにページフロー内の最初のノードとして使用します。

ただし、 ページフローがアクションリスナー呼び出しの結果として開始される場合、 アクションリスナーの結果は最初のページをレンダリングすると決定します。 この場合、 **<start-state>** をページフローの最初のノードとして使用し、 可能性のある結果それぞれに対して遷移を宣言します。

```
<pageflow-definition name="viewEditDocument">
  <start-state name="start">
    <transition name="documentFound" to="displayDocument"/>
    <transition name="documentNotFound" to="notFound"/>
  </start-state>

  <page name="displayDocument" view-id="/document.jsp">
    <transition name="edit" to="editDocument"/>
    <transition name="done" to="main"/>
  </page>

  ...

  <page name="notFound" view-id="/404.jsp">
    <end-conversation/>
  </page>

</pageflow-definition>
```

9.2.3. ページノードと遷移

各 **<page>** ノードは、システムがユーザー入力を待っている状態を表します。

```
<page name="displayGuess" view-id="/numberGuess.jsp">
  <redirect/>
  <transition name="guess" to="evaluateGuess">
```

```
    <action expression="#{numberGuess.guess}" />
  </transition>
</page>
```

view-id は JSF のビュー ID です。 **<redirect/>** エレメントは JSF ナビゲーションルールにある **<redirect/>** と同じ効果をもたらします。 つまり、 post-then-redirect の動作をしてブラウザの更新ボタンに関する問題に対応します。 (Seam はこれらブラウザのリダイレクト全体に対話コンテキストを伝播するため、 Ruby on Rails 系の **フラッシュ** 構造は必要としません。)

遷移名は **numberGuess.jsp** のコマンドボタンまたはコマンドリンクをクリックすると起こる JSF 結果の名前です。

```
<h:commandButton type="submit" value="Guess" action="guess"/>
```

このボタンをクリックして遷移が起こると、 jBPM は **numberGuess** コンポーネントの **guess ()** メソッドを呼び出して遷移のアクションを起動します。 jPDL でアクションの指定に使用される構文は普通の JSF EL 式であり、 遷移のハンドラは現在の Seam コンテキストにある Seam コンポーネントのメソッドになります。 従って、 JSF イベント用のイベントモデルと同じイベントモデルを jBPM イベント用に持つことになります。これは Seam で従うべき原則の 1 つです。

結果が null になる場合 (**action** が定義されていないコマンドボタンなど)、 Seam は名前のない遷移のサインがあればそれを送信します。 すべての遷移に名前が付けられている場合には単純にそのページを再表示します。 したがって次のようにこのボタンとページフローを単純化することができます。

```
<h:commandButton type="submit" value="Guess"/>
```

次のような名前のない遷移を実行します。

```
<page name="displayGuess" view-id="/numberGuess.jsp">
  <redirect/>
  <transition to="evaluateGuess">
    <action expression="#{numberGuess.guess}" />
  </transition>
</page>
```

ボタンにアクションメソッドを呼び出すことも可能です。 この場合アクションの結果が行われる遷移を決定します。

```
<h:commandButton type="submit" value="Guess"
   action="#{numberGuess.guess}"/>
```

```
<page name="displayGuess" view-id="/numberGuess.jsp">
  <transition name="correctGuess" to="win"/>
  <transition name="incorrectGuess" to="evaluateGuess"/>
</page>
```

ただし、 これはフロー制御をページフロー定義以外に移行して他のコンポーネントに戻しているため質の悪いスタイルだと考えられます。 フロー制御関連はページフロー自体に集中させる方がよいでしょう。

9.2.4. フローの制御

一般的にページフローを定義するときには jPDL にこれ以上強力な機能は必要とされません。 ただし **<decision>** ノードは必要になります。

```
<decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
  <transition name="true" to="win"/>
  <transition name="false" to="evaluateRemainingGuesses"/>
</decision>
```

デシジョンは Seam コンテキスト内で JSF EL 式を評価することにより決定されます。

9.2.5. フローの終了

<end-conversation> または **@End** を使用して対話を終了します。 読みやすいよう両方とも使用した方がいいでしょう。

```
<page name="win" view-id="/win.jsp">
  <redirect/>
  <end-conversation/>
</page>
```

オプションとしてタスクを終了するか、 jBPM の **transition** 名を指定することができます。 この場合 Seam は包括的なビジネスプロセスで現在のタスクの終了サインを送信します。

```
<page name="win" view-id="/win.jsp">
  <redirect/>
  <end-task transition="success"/>
</page>
```

9.2.6. ページフローの構成

ページフローを構成することは可能です。 これにより別のページフローが実行している間に任意のページフローだけを中断させることができます。 **<process-state>** ノードは外部のページフローを中断させてから指定ページフローの実行を開始します。

```
<process-state name="cheat">
  <sub-process name="cheat"/>
  <transition to="displayGuess"/>
</process-state>
```

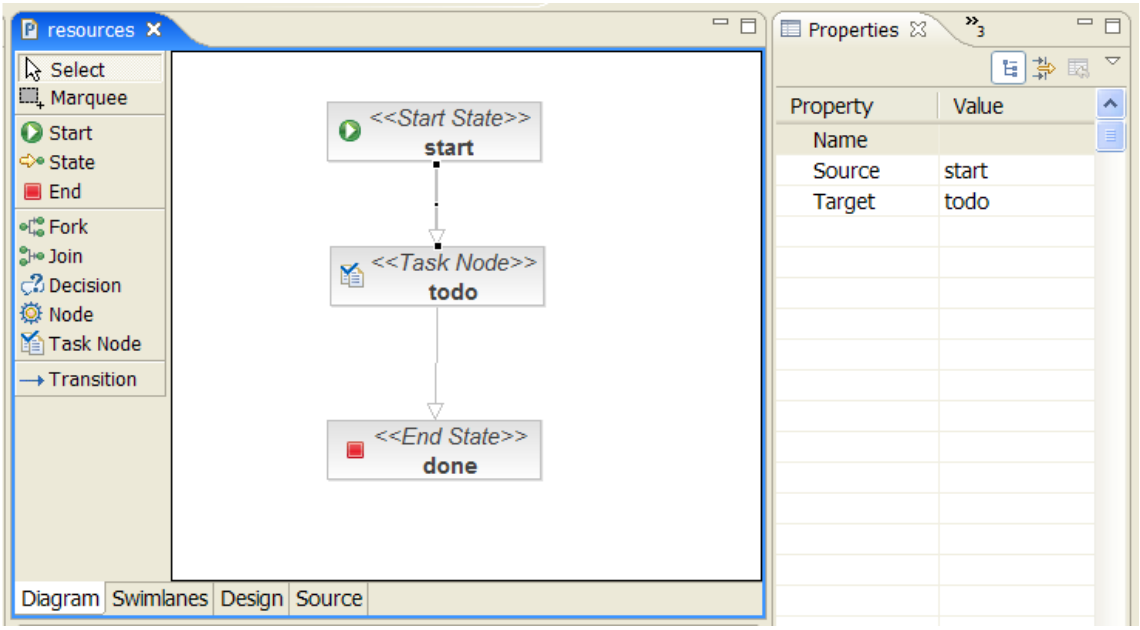
<start-state> ノードで内側のフローは開始します。 **<end-state>** ノードの到着すると、内側のフローは終了し、外側のフローは **<process-state>** 要素で定義された遷移から再開されます。

9.3. SEAM のビジネスプロセス管理

任意のタスクを行う実行者、およびそのタスクを実行するタイミングに関して明確に定義したルールに従い、ユーザーまたはソフトウェアシステムにより実行する必要がある一連のタスクがビジネスプロセスです。 Seam の jBPM 統合によりユーザーによるタスク一覧の表示や管理が容易になります。 またビジネスプロセスに関連付けられた状態を **BUSINESS_PROCESS** コンテキストに格納し、 jBPM 変数でその状態を永続にすることもできます。

<page> ノードではなく **<task-node>** ノードを使用するという点以外、 シンプルなビジネスプロセス定義はページフロー定義によく似ています。 長期実行のビジネスプロセスでは、 ユーザーによりログインが行われタスクが実行されるのをシステムが待っている場合に待ち状態が起こります。

```
<process-definition name="todo">
  <start-state name="start">
    <transition to="todo"/>
  </start-state>

  <task-node name="todo">
    <task name="todo" description="#{todoList.description}">
      <assignment actor-id="#{actor.id}"/>
    </task>
    <transition to="done"/>
  </task-node>

  <end-state name="done"/>

</process-definition>
```



jPDL ビジネスプロセス定義と jPDL ページフロー定義は同じプロジェクトで使用することができます。この場合、 ビジネスプロセスにある 1 つの **<task>** は **<pageflow-definition>** ページフロー全体に該当します。

9.4. JPDL ビジネスプロセス定義の使用

9.4.1. プロセス定義のインストール

まず、 jBPM をインストールしてビジネスプロセス定義の場所を指示する必要があります。

```
<bpm:jbpm>
  <bpm:process-definitions>
    <value>todo.jpdl.xml</value>
  </bpm:process-definitions>
</bpm:jbpm>
```

jBPM プロセスはアプリケーションの再起動行っても存続するため、 実稼働環境で Seam を使用する場合はアプリケーション起動のたびにプロセス定義をインストールす必要はありません。 したがってプロセスは Seam の外側にある jBPM にデプロイする必要があります。 **components.xml** からプロセス定義をインストールする必要があるのはアプリケーション開発時のみです。

9.4.2. actor ID の初期化

現在ログインしているユーザーを常に把握しておく必要があります。 jBPM はユーザーの **actor ID** と **group actor ID** でユーザーを識別します。 **actor** と呼ばれる組み込み Seam コンポーネントを使用して現在の actor ID を指定します。

```
@In Actor actor;

public String login() {
  ...
  actor.setId( user.getUserName() );
  actor.getGroupActorIds().addAll( user.getGroupNames() );
  ...
}
```

9.4.3. ビジネスプロセスの初期化

ビジネスプロセスインスタンスを初期化するためには **@CreateProcess** アノテーションを使用します。

```
@CreateProcess(definition="todo")
public void createTodo() { ... }
```

もしくは、 **pages.xml** を使用してビジネスプロセスを初期化することもできます。

```
<page>
  <create-process definition="todo" />
</page>
```

9.4.4. タスクの割り当て

プロセスがタスクノードに到達するとタスクのインスタンスが作成されます。 このタスクインスタンスは、ユーザーまたはユーザーグループに割り当てられなければなりません。actor ID をハードコード化するか、 Seam コンポーネントに委譲することができます。

```
<task name="todo" description="#{todoList.description}">
  <assignment actor-id="#{actor.id}"/>
</task>
```

この場合、 単純に現在のユーザーにタスクを割り当てます。 タスクをプールに割り当てることもできます。

```
<task name="todo" description="#{todoList.description}">
  <assignment pooled-actors="employees"/>
</task>
```

9.4.5. タスクリスト

いくつかの組み込み Seam コンポーネントによりタスクリストの表示が容易になります。**pooledTaskInstanceList** は ユーザーが自分自身に割り当てることができるプールされたタスクのリストです。

```
<h:dataTable value="#{pooledTaskInstanceList}" var="task">
  <h:column>
    <f:facet name="header">Description</f:facet>
    <h:outputText value="#{task.description}"/>
  </h:column>
  <h:column>
    <s:link action="#{pooledTask.assignToCurrentActor}"
            value="Assign" taskInstance="#{task}"/>
  </h:column>
</h:dataTable>
```

<s:link> の代わりに普通の JSF **<h:commandLink>** を使用することもできます。

```
<h:commandLink action="#{pooledTask.assignToCurrentActor}">
  <f:param name="taskId" value="#{task.id}"/>
</h:commandLink>
```

pooledTask コンポーネントは、単純にタスクを現在のユーザーに割り当てる組み込みコンポーネントです。

taskInstanceListForType コンポーネントは、 現在のユーザーに割り当てられた特定タイプのタスクを含んでいます。

```
<h:dataTable value="#{taskInstanceListForType['todo']}" var="task">
  <h:column>
    <f:facet name="header">Description</f:facet>
    <h:outputText value="#{task.description}"/>
  </h:column>
  <h:column>
    <s:link action="#{todoList.start}"
            value="Start Work" taskInstance="#{task}"/>
  </h:column>
</h:dataTable>
```

9.4.6. タスクの実行

タスクの作業を開始させるために、リスナーメソッドに **@StartTask** または **@BeginTask** を使用します。

```
@StartTask public String start() { ... }
```

もしくは、**pages.xml** を使用してタスクを開始することもできます。

```
<page>
  <start-task />
</page>
```

これらのアノテーションは包括的なビジネスプロセスという点において重要となる特殊な種類の対話を開始します。 この対話による処理はビジネスプロセスコンテキスト内に保持される状態へのアクセスを有します。

@EndTask で対話を終了すると Seam はタスクの完了サインを送信します。

```
@EndTask(transition="completed")
public String completed() { ... }
```

代わりに、 **pages.xml** を使用することもできます。

```
<page>
  <end-task transition="completed" />
</page>
```

EL を使用して **pages.xml** に遷移を指定することもできます。

この時点では、 jBPM はビジネスプロセス定義の実行を継続します (さらに複雑なプロセスでは、 プロセスの実行が再開される前にいくつかのタスクを完了する必要があるかもしれません)。

複雑なビジネスプロセスの管理を実現するための jBPM が提供する高度な機能の全体的な概要については jBPM ドキュメントを参照してください。

第10章 SEAM とオブジェクト / リレーショナルマッピング

Seam は Enterprise JavaBeans 3.0 (EJB3) で導入される Java Persistence API および Hibernate の 2 つの最も一般的な Java 用永続アーキテクチャに対して広範なサポートを提供します。 Seam 固有の状態管理アーキテクチャにより、 あらゆる Web アプリケーションフレームワークからも高度な ORM 統合を実現します。

10.1. はじめに

Seam は Java アプリケーションアーキテクチャの旧世代の典型的なステートレス性による煩わしさから生まれました。当初 Seam の状態管理のアーキテクチャは永続性、 特に **楽観的トランザクションの処理** に関する問題を解決する目的で設計されていました。 スケーラブルなオンラインアプリケーションは常に楽観的トランザクションを使用します。 アプリケーションがごく少数の同時クライアントに対応するよう設計されていない限り、 アトミック (データベース／JTA) レベルのトランザクションはユーザー操作にはスパンしないはずです。 しかし、ほとんどすべての作業は最初にユーザーにデータを表示してからそのデータを更新することです。このため、 Hibernate は楽観的トランザクションをスパンした永続コンテキストの概念に対応するよう設計されました。

残念ながら、 Seam や EJB 3.0 より以前の「ステートレス」と呼ばれるアーキテクチャには楽観的なトランザクションを表現するための構成概念がありませんでした。 このため、 代わりにアトミックなトランザクションに対してスコープされる永続コンテキストを提供していました。当然これはユーザーにとって多くの問題を引き起こし、 また Hibernate に関するユーザーからの最大の苦情、 `LazyInitializationException` の原因となっています。 ここで必要なのはアプリケーション層で楽観的トランザクションを表現する構成概念です。

EJB 3 はこの問題を認識し、 コンポーネントの寿命に対してスコープされる **拡張永続コンテキスト** を持ったステートフルなコンポーネント (ステートフルセッション Bean) というアイデアを導入します。これは問題を解決する完全なソリューションではなく (それ自体は便利な構成です)、 まだこの方法には 2 つの問題が残っています。

- ステートフルセッション Bean の寿命はウェブ層でコードにより手作業で管理されなければなりません。
- 同じ楽観的トランザクション内のステートフルコンポーネント間での永続コンテキストの伝播は可能ですが非常に複雑です。

Seam は対話を提供することにより 1 番目の問題を解決し、 対話に対してステートフルセッションの Bean コンポーネントをスコープします (ほとんどの対話は実際にはデータ層で楽観的トランザクションを表示します)。Seam 予約サンプルなどの永続コンテキストの伝播を必要としない多くのシンプルなアプリケーションにはこれで十分です。 各対話内で疎に作用しあっているコンポーネントを多く持つもう少し複雑なアプリケーションの場合、 コンポーネント群全体への永続コンテキストの伝播は重要な問題となります。 このため Seam は EJB3 の永続コンテキスト管理モデルを拡張して、対話スコープの拡張永続コンテキストを提供します。

10.2. SEAM 管理トランザクション

EJB セッション Bean は宣言型トランザクション管理を特長としています。 EJB コンテナは Bean が呼び出されると透過的にトランザクションを起動し、 呼出しが終了するとトランザクションも終了させることが可能です。 JSF アクションリスナーとして動作するセッション Bean メソッドを記述する場合、 そのアクションに関連するすべての作業を 1 つのトランザクションとして行うことができ、 アクションの処理が完全に終了したらコミットまたはロールバックされるようにできます。これは便利な機能であり、いくつかの Seam アプリケーションに必要なのはこれだけです。

ただしこの方法には問題が 1 つあります。1 つのセッション Bean への単一のメソッド呼び出しの 1 要求で、 Seam のアプリケーションは全データアクセスを行うわけではありません。

- リクエストがいくつかの疎結合コンポーネントによる処理を必要とし、それぞれのコンポーネントが Web 層から個別に呼び出される場合です。 Seam でリクエストごと Web 層から EJB コンポーネントへの呼び出しが複数あるのはよく見られることです。
- ビューのレンダリングに関連の遅延フェッチが必要な場合です。

要求ごとに存在するトランザクション数が多くなると、 使用しているアプリケーションが多くの同時要求を処理している間にそれだけ多くのアトミック性と独立性の問題に遭遇する可能性が高くなります。 書き込み動作はすべて同じトランザクション内で起こらなければならないからです。

Hibernate ユーザーはこの問題に対処するため **open session in view** パターンを開発しました。 Spring のようなフレームワークはトランザクションスコープの永続コンテキストを使用しているのでこれも重要です。 これによりフェッチされない関連がアクセスされると **`LazyInitializationException`** が引き起こされました。

Open session in view は要求全体に広がる単一トランザクションとして通常は実装されます。 この実装で最も深刻な問題は、コミットするまでトランザクションが成功したかがわからない点です。 ただし、 トランザクションがコミットされるとビューは完全にレンダリングされ、 レンダリングされた応答はすでにクライアントを同期している可能性があります。 このため、 ユーザーにトランザクションが成功しなかったことを伝える手段がありません。

Seam はトランザクション独立性の問題と関連フェッチの問題を解決しながら、**open session in view** の主要な不備な点にも対処します。 変更点が 2 つあります。

- Seam はトランザクションではなく対話に対してスコープされる拡張永続コンテキストを使用します。
- Seam は要求ごとに 2 つのトランザクションを使用します。 1 番目はビュー復元フェーズの開始からアプリケーション起動フェーズの終わりまでに渡り、2 番目はレスポンス出力フェーズまでに渡ります (アプリケーションの中には、 最初のフェーズがこれより後のリクエスト値の適用フェーズの開始時に始まるものもあります)。

次項では、対話スコープの永続コンテキストの設定方法について説明していきますが、その前に Seam トランザクション管理を有効にします。 Seam トランザクション管理なしで対話スコープの永続コンテキストを使用することができます。また、Seam 管理永続コンテキストがなくても Seam トランザクション管理を利用すると便利です。ただし、 この 2 つの機能は併用する方が効果的です。

10.2.1. Seam 管理トランザクションを無効にする

Seam トランザクション管理はデフォルトではすべての JSF 要求に有効ですが、**`components.xml`** で無効にすることができます。

```
<core:init transaction-management-enabled="false"/>

<transaction:no-transaction />
```

10.2.2. Seamトランザクションマネージャの設定

Seam はトランザクションでの開始、 コミット、 ロールバック、 同期などの動作にトランザクション管理の抽象化を提供します。 デフォルトでは Seam はコンテナ管理やプログラムでの EJB トランザクションを統合する JTA トランザクションコンポーネントを使用します。 Java EE 5 の環境で作業している場合は **`components.xml`** に EJB 同期化コンポーネントをインストールしてください。

```
<transaction:ejb-transaction />
```

ただし、 EE 5 コンテナ以外で作業している場合は Seam が適切なトランザクション同期化メカニズムの自動検出を試行します。 Seam が正しいメカニズムを検出できない場合は、次のいずれかを設定する必要があるかもしれません。

- **javax.persistence.EntityTransaction** インターフェースで JPA RESOURCE_LOCAL 管理のトランザクションを設定します。 **EntityTransaction** はリクエスト値の適用フェーズの開始時にトランザクションを開始します。
- **org.hibernate.Transaction** インターフェースで Hibernate 管理のトランザクションを設定します。 **HibernateTransaction** はリクエスト値の適用フェーズの開始時にトランザクションを開始します。
- **org.springframework.transaction.PlatformTransactionManager** インターフェースで Spring 管理トランザクションを設定します。 Spring の **PlatformTransactionManagement** マネージャは **userConversationContext** 属性を設定するとリクエスト値の適用フェーズの開始時にトランザクションを開始することができます。
- Seam 管理トランザクションを明示的に無効にします。

components.xml に次を追加して JPA RESOURCE_LOCAL トランザクション管理を設定します。 **#{em}** は **persistence:managed-persistence-context** コンポーネント名です。 管理永続コンテキスト名が **entityManager** なら **entity-manager** 属性を省略することができます (詳細は 「Seam 管理永続コンテキスト」 を参照)。

```
<transaction:entity-transaction entity-manager="#{em}"/>
```

Hibernate 管理トランザクションを設定するには **components.xml** で次を宣言します。**#{hibernateSession}** はプロジェクトの **persistence:managed-hibernate-session** コンポーネント名です。 管理 Hibernate セッション名が **session** なら **session** 属性を省略することができます (詳細は 「Seam 管理永続コンテキスト」を参照)。

```
<transaction:hibernate-transaction session="#{hibernateSession}"/>
```

Seam 管理トランザクションを明示的に無効にするには次を **components.xml** で宣言します。

```
<transaction:no-transaction />
```

Spring 管理トランザクションの設定方法については 「Spring の PlatformTransactionManagement の使用」 を参照してください。

10.2.3. トランザクションの同期化

トランザクションの同期化は **beforeCompletion()** や **afterCompletion()** などトランザクション関連のイベントにコールバックを提供します。 デフォルトでは Seam はそれ自体のトランザクション同期化コンポーネントを使用します。これには同期化のコールバックが必ず正しく実行されるようトランザクションをコミットするときに Seam トランザクションコンポーネントを明示的に使用することが必要です。 Java EE 5 環境では **<transaction:ejb-transaction/>** を **components.xml** で宣言し、 コンテナが Seam の認識範囲外にトランザクションをコミットする場合に Seam 同期化のコールバックが正しく呼び出されるようにしてください。

10.3. SEAM 管理永続コンテキスト

Seam を Java EE 5 環境外で使用している場合、 コンテナによる永続コンテキストのライフサイクルの管理は期待できません。EE 5 環境であっても、複雑なアプリケーションの疎結合コンポーネント間で永続コンテキストを伝播することは容易ではなく、エラーが発生しやすいことがあります。

この場合、 コンポーネントで **管理永続コンテキスト** (JPA 用) または **管理セッション** (Hibernate 用) のいずれかを使用する必要があります。 Seam 管理永続コンテキストは単に対話コンテキストの **EntityManager** または **Session** のインスタンスを管理する組み込みの Seam コンポーネントです。 **@In** でインジェクトすることができます。

Seam 管理永続コンテキストはクラスタ環境で非常に効率的です。Seam は EJB3 の仕様ではできないコンテナ管理永続コンテキストの最適化を実行することができます。 ノード間の永続コンテキストの状態を複製することなく拡張永続コンテキストの透過的なフェールオーバーをサポートします (この点については、EJB 仕様の次回のリビジョンで修正したいと考えています)。

10.3.1. JPA での Seam 管理永続コンテキストの使用

管理永続コンテキストの設定は簡単です。 **components.xml** 内に次のように記述します。

```
<persistence:managed-persistence-context name="bookingDatabase"
   auto-create="true"
   persistence-unit-jndi-name="java:/EntityManagerFactories/bookingData"/>
```

この設定により対話スコープの **bookingDatabase** という名前の Seam コンポーネントが作成され、JNDI 名 **java:/EntityManagerFactories/bookingData** を持つ永続ユニット (**EntityManagerFactory** インスタンス) の **EntityManager** インスタンスの寿命を管理します。

EntityManagerFactory を JNDI にバインドする必要があります。これを行うには JBoss では、 次のプロパティ設定を **persistence.xml** に追加します。

```
<property name="jboss.entity.manager.factory.jndi.name"
          value="java:/EntityManagerFactories/bookingData"/>
```

これで以下のように **EntityManager** をインジェクトできます。

```
@In EntityManager bookingDatabase;
```

EJB 3 を使用していてクラスまたはメソッドに **@TransactionAttribute(REQUIRES_NEW)** をマークすると、トランザクションと永続コンテキストはこのオブジェクトでのメソッド呼び出しには伝播されないはずです。 ただし、 Seam 管理永続コンテキストは対話内のいずれのコンポーネントにも伝播されるため **REQUIRES_NEW** とマークされたメソッドに伝播されます。 したがって、 メソッドに **REQUIRES_NEW** をマークする場合は **@PersistenceContext** を使ってエンティティマネージャにアクセスしてください。

10.3.2. Seam 管理の Hibernate セッションの使用

Seam 管理の Hibernate セッションも同様に**components.xml** で次のように記述することができます。

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>

<persistence:managed-hibernate-session name="bookingDatabase"
            auto-create="true"
            session-factory-jndi-name="java:/bookingSessionFactory"/>
```

java:/bookingSessionFactory は **hibernate.cfg.xml** で指定されるセッションファクトリ名です。

```
<session-factory name="java:/bookingSessionFactory">
  <property name="transaction.flush_before_completion">true</property>
  <property name="connection.release_mode">after_statement</property>
  <property name="transaction.manager_lookup_class">
    org.hibernate.transaction.JBossTransactionManagerLookup
  </property>
  <property name="transaction.factory_class">
    org.hibernate.transaction.JTATransactionFactory
  </property>
  <property name="connection.datasource">
    java:/bookingDatasource
  </property>
  ...
</session-factory>
```

注記

Seam はデータベースでセッションを同期しないので、**hibernate.transaction.flush_before_completion** を常に有効にしてセッションが JTA トランザクションのコミットより先に自動的に同期されるようにしてください。

これで、 次のコードを使って JavaBean コンポーネントに管理 Hibernate **Session** をインジェクトできます。

```
@In Session bookingDatabase;
```

10.3.3. Seam 管理永続コンテキストとアトミックな対話

merge() を使用したり、 各要求の開始時にデータを再ロードしたり、 例外と格闘しなくとも (**LazyInitializationException** や **NonUniqueObjectException**)、 対話にスコープされる永続コンテキストにより複数のサーバー要求にまたがる楽観的なトランザクションをプログラムすることができます。

楽観的ロックを使ってトランザクションの分離と一貫性を実現します。 Hibernate および EJB3 はいずれも **@Version** アノテーションを付与することで楽観的ロックの使用を容易にします。

デフォルトでは、 永続コンテキストは各トランザクションの終わりでデータベースと同期 (フラッシュ) されます。 これが目的の動作である場合もありますが、 すべての変更はメモリに保持され対話が正常に終了したときにのみデータベースに書き込まれる動作を期待することの方が多いでしょう。 これにより EJB3 永続との真にアトミックな対話を可能にします。 ただし、 Hibernate では仕様により定義される **FlushModeType** に対するベンダー拡張としてこの機能を提供しています。他のベンダーもすぐに同様の拡張機能を提供することが予想されます。

Seam では対話の開始時に **FlushModeType.MANUAL** を指定することができます。 現在は Hibernate が永続を実現する構成要素である場合にのみ機能しますが、 他のベンダーによる同等の拡張機能もサポートする予定です。

```
@In EntityManager em; //a Seam-managed persistence context
@Begin(flushMode=MANUAL)
```

```
public void beginClaimWizard() {
  claim = em.find(Claim.class, claimId);
}
```

これで **claim** オブジェクトは全対話の間、永続コンテキストによって管理され続けます。この **claim** に変更を加えることができます。

```
public void addPartyToClaim() {
  Party party = ....;
  claim.addParty(party);
}
```

ただし、これらの変更は明示的に同期が発生するよう強制するまではデータベースに対してフラッシュされません。

```
@End public void commitClaim() {
  em.flush();
}
```

pages.xml から **flushMode** を **MANUAL** に設定することもできます。たとえばナビゲーションルールでは以下のようになります。

```
<begin-conversation flush-mode="MANUAL" />
```

いずれの **Seam** 管理永続コンテキストに対しても手動によるフラッシュモードを使用するよう設定できます。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core">
  <core:manager conversation-timeout="120000"
        default-flush-mode="manual" />
</components>
```

10.4. JPA「DELEGATE」の使用

EntityManager インターフェースにより **getDelegate()** メソッドを通じてベンダー固有の **API** にアクセスすることができます。**Hibernate** をベンダーとして使用し、**org.hibernate.Session** を **delegate** インターフェースとして使用することをお勧めしますが、別の **JPA** プロバイダを使用する必要がある場合は 「代替の JPA プロバイダの使用」で詳細をご確認ください。

どのベンダーを使用するかに関わらず、**Seam** コンポーネントで **delegate** を使用する方法はいくつかあります。以下にその一例を示します。

```
@In EntityManager entityManager;
@Create public void init() {
  ((Session)entityManager.getDelegate() ).enableFilter("currentVersions");
}
```

ほとんどの **Java** ユーザーと同様、型キャストの使用を避けたい場合は、次の行を **components.xml** に追加することで **delegate** にアクセスすることもできます。

```
<factory name="session" scope="STATELESS" auto-create="true"
         value="#{entityManager.delegate}"/>
```

これでセッションを直接インジェクトできるようになります。

```
@In Session session;

@Create
public void init() {
  session.enableFilter("currentVersions");
}
```

10.5. EJB-QL/HQL での EL の使用

Seam 管理永続コンテキストを使用する場合や **@PersistenceContext** でコンテナ管理永続コンテキストをインジェクトする場合は、常に Seam は **EntityManager** または **Session** オブジェクトをプロキシします。 これにより、 EL 式をクエリ文字列内で問題なく効果的に使用することができます。 たとえば、 次を見てください。

```
User user = em.createQuery("from User where username=#{user.username}")
                              .getSingleResult();
```

上記の例は、 以下の例と同等です。

```
User user = em.createQuery("from User where username=:username")
                              .setParameter("username", user.getUsername())
                              .getSingleResult();
```

警告

以下の形式は使用しないでください。SQL インジェクション攻撃に脆弱性があり、非効率でもあります。

```
User user = em.createQuery("from User where username=" +
user.getUsername()).getSingleResult(); //BAD!
```

10.6. HIBERNATE フィルタの使用

フィルタ は Hibernate 固有の最も便利な機能です。 フィルタによりデータベース内のデータ表示に制限を与えることができます。 フィルタについては Hibernate のドキュメントで詳細に説明されています。本項ではフィルタを Seam に統合する簡単で効果的な方法を記載します。

Seam 管理永続コンテキストには定義したフィルタの一覧を持たせることができます。 **EntityManager** や Hibernate **Session** が初めて作成されたときは常に有効になります (Hibernate が永続を実現する構成要素である場合にのみ使用できます)。

```
<persistence:filter name="regionFilter">
```

```
  <persistence:name>region</persistence:name>
  <persistence:parameters>
    <key>regionCode</key>
    <value>#{region.code}</value>
  </persistence:parameters>
</persistence:filter>

<persistence:filter name="currentFilter">
  <persistence:name>current</persistence:name>
  <persistence:parameters>
    <key>date</key>
    <value>#{currentDate}</value>
  </persistence:parameters>
</persistence:filter>

<persistence:managed-persistence-context name="personDatabase"
  persistence-unit-jndi-
name="java:/EntityManagerFactories/personDatabase">
  <persistence:filters>
    <value>#{regionFilter}</value>
    <value>#{currentFilter}</value>
  </persistence:filters>
</persistence:managed-persistence-context>
```

第11章 SEAM での JSF フォーム検証

純粋な JSF では検証はビューで定義されます。

```
<h:form>
  <h:messages/>

  <div>
    Country:
    <h:inputText value="#{location.country}" required="true">
      <my:validateCountry/>
    </h:inputText>
  </div>

  <div>
    Zip code:
    <h:inputText value="#{location.zip}" required="true">
      <my:validateZip/>
    </h:inputText>
  </div>

  <h:commandButton/>
</h:form>
```

実際にはこの方法は通常 DRY に違反しています。データモデルの一部であり、データベーススキーマの定義全体にわたって存在する制約をほとんどの「検証」が実際は強制実行するためです。Seam は Hibernate Validator を使って定義されるモデルベースの制約に対するサポートを提供しています。

Location クラスで制約を定義するところから始めましょう。

```
public class Location {
    private String country;
    private String zip;

    @NotNull
    @Length(max=30)
    public String getCountry() { return country; }
    public void setCountry(String c) { country = c; }

    @NotNull
    @Length(max=6)
    @Pattern("^\d*$")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

実際には Hibernate Validator に組み込みされたものではなく、カスタムな制約を使う方がスマートかもしれません。

```
public class Location {
    private String country;
    private String zip;

    @NotNull
    @Country
```

```
    public String getCountry() { return country; }
    public void setCountry(String c) { country = c; }

    @NotNull
    @ZipCode
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

いずれの方法を取るにしても、JSF ページ内で使用される検証のタイプを指定する必要はありません。代わりに、**<s:validate>** を使ってモデルオブジェクトで定義される制約に対して検証を行います。

```
<h:form>
  <h:messages/>

  <div>
    Country:
    <h:inputText value="#{location.country}" required="true">
      <s:validate/>
    </h:inputText>
  </div>

  <div>
    Zip code:
    <h:inputText value="#{location.zip}" required="true">
      <s:validate/>
    </h:inputText>
  </div>

  <h:commandButton/>

</h:form>
```

注記

このモデルで **@NotNull** を指定してもコントロールに出現させるのに **required="true"** が必要なくなるというわけではありません。これは JSF 検証アーキテクチャの限界によるものです。

この方法によりモデルで制約を定義して、ビューで制約違反を提示します。

デザインはよくなりましたが、最初のデザインと比べてそれほど冗長性が軽減されているわけではありません。**<s:validateAll>** を使ってみます。

```
<h:form>

  <h:messages/>

  <s:validateAll>

  <div>
    Country:
    <h:inputText value="#{location.country}" required="true"/>
  </div>
```

```
  <div>
    Zip code:
    <h:inputText value="#{location.zip}" required="true"/>
  </div>

  <h:commandButton/>

  </s:validateAll>

</h:form>
```

このタグは **<s:validate>** をフォーム内のすべての入力に追加します。 フォームが大きくなる場合は、 入力の手間をかなり省くことができます。

次に、 検証が失敗した場合にユーザーにフィードバックを表示させる必要があります。 現在すべてのメッセージはフォームの冒頭に表示されます。 メッセージと入力を関連付けられるようにするためには、 入力コンポーネントで標準の **label** 属性を使いラベルを定義する必要があります。

```
<h:inputText value="#{location.zip}" required="true" label="Zip:">
  <s:validate/>
</h:inputText>
```

プレースホルダーの {0} (Hiberate Validator の制約用に JSF メッセージに渡される最初で唯一のパラメータ) を使ってこの値をメッセージ文字列にインジェクトします。これらのメッセージを定義する場所の詳細は「国際化」の項をご覧ください。

注記

validator.length={0} length must be between {min} and {max}

エラーがあるフィールドの隣にメッセージを表示させ、 フィールドとラベルをハイライトさせて、 フィールドの隣にイメージを表示させたいとします。純粋な JSF で可能なのは最初のメッセージの表示のみです。 また、 必須フォームの各フィールドにはラベルの隣に色の付いたアスタリスクを表示させたいとします。

これは各フィールドにとって多くの機能となります。 フォームにあるすべてのフィールドそれぞれに対してイメージ、 メッセージ、 入力フィールドのレイアウトやハイライトを指定したいとは思わないでしょうから、Facelets テンプレートにそのレイアウトを指定します。

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:f="http://java.sun.com/jsf/core"
                xmlns:s="http://jboss.com/products/seam/taglib">
  <div>

    <s:label styleClass="#{invalid?'error':''}">
      <ui:insert name="label"/>
      <s:span styleClass="required" rendered="#{required}">*</s:span>
    </s:label>

    <span class="#{invalid?'error':''}">
      <h:graphicImage value="/img/error.gif" rendered="#{invalid}"/>
      <s:validateAll>
```

```
        <ui:insert/>
      </s:validateAll>
    </span>

    <s:message styleClass="error"/>

  </div>

</ui:composition>
```

<s:decorate> を使って各フォームフィールドにこのテンプレートを含ませることができます。

```
<h:form>
  <h:messages globalOnly="true"/>

    <s:decorate template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText value="#{location.country}" required="true"/>
    </s:decorate>

    <s:decorate template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText value="#{location.zip}" required="true"/>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

最後に、ユーザーがフォーム内を移動しながら RichFaces Ajax を使って検証メッセージを表示させることができます。

```
<h:form>
  <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText value="#{location.country}" required="true">
            <a:support event="onblur" reRender="countryDecoration"
                       bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <s:decorate id="zipDecoration" template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText value="#{location.zip}" required="true">
            <a:support event="onblur" reRender="zipDecoration"
                       bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

重要なページのコントロールには明示的な ID を定義すると便利なスタイルになります。 特に UI 用の自動テストを行いたい場合などに適しています。 明示的な ID を与えないと、 JSF はそれらを生成しますがページ上で変更があると静的なままにはなりません。

```
<h:form id="form">
  <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
    <ui:define name="label">Country:</ui:define>
      <h:inputText id="country" value="#{location.country}"
                   required="true">
        <a:support event="onblur" reRender="countryDecoration"
                   bypassUpdates="true"/>
      </h:inputText>
    </s:decorate>

    <s:decorate id="zipDecoration" template="edit.xhtml">
      <ui:define name="label">Zip code:</ui:define>
      <h:inputText id="zip" value="#{location.zip}" required="true">
        <a:support event="onblur" reRender="zipDecoration"
                   bypassUpdates="true"/>
      </h:inputText>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

検証が失敗したときに表示させるメッセージを変えたい場合、 Seam メッセージバンドルを Hibernate Validator で使用することができます。

```
public class Location {
    private String name;
    private String zip;

    // Getters and setters for name

    @NotNull
    @Length(max=6)
    @ZipCode(message="#{messages['location.zipCode.invalid']}")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

```
 location.zipCode.invalid = The zip code is not valid for #{location.name}
```

第12章 GROOVY 統合

Seam は Rapid Application Development (RAD) に関する素晴らしい機能を備えています。Seam により 標準 Java API での互換性を維持しながらも既存のプラットフォームで動的な言語を利用できます。静的言語と動的言語が統合されるためコンテキスト切り替えが不要となり、 通常の Seam コンポーネントと同様に同じアノテーションと API を使用して動的な Seam コンポーネントの記述ができるようになります。

12.1. はじめに

Groovy は使い勝手の軽快な Java ベースの動的な言語で Python、 Ruby、 Smalltalk に基づいた機能を備えています。 Java ベースであり Java オブジェクトとクラスを持つため Groovy は理解しやすく、また既存の Java ライブラリやフレームワークとのシームレスな統合が容易となります。

12.2. GROOVY による SEAM アプリケーションの記述

Groovy オブジェクトは Java オブジェクトであるため、 Seam コンポーネントを Groovy で記述しデプロイすることができます。 また、 同じアプリケーションで Groovy のクラスと Java のクラスを一緒に使うこともできます。

12.2.1. Groovy コンポーネントの記述

アノテーションに対応させるには Groovy 1.1 またはそれ以降を使用する必要があります。次項では Seam アプリケーションでの Groovy の使用方法について示しています。

12.2.1.1. エンティティ

例12.1 Seam アプリケーションでの Groovy の使用

```
@Entity
@Name("hotel")
class Hotel implements Serializable {
    @Id @GeneratedValue
    Long id

        @Length(max=50) @NotNull
        String name

        @Length(max=100) @NotNull
        String address

        @Length(max=40) @NotNull
        String city

        @Length(min=2, max=10) @NotNull
        String state

        @Length(min=4, max=6) @NotNull
        String zip

        @Length(min=2, max=40) @NotNull
        String country

        @Column(precision=6, scale=2)
```

```
        BigDecimal price

        @Override
        String toString(){
        return "Hotel(${name},${address},${city},${zip})"
    }
}
```

Groovy ではプロパティをサポートしているため、 冗長な getter や setter を明示的に記述する必要がありません。 上記の例では、 hotel クラスは Java から **hotel.getCity()** でアクセスできます。 この getter や setter は Groovy のコンパイラが生成したものです。 こうした簡略構文を使えば、 エンティティコードは非常に簡潔になります。

12.2.2. Seam コンポーネント

Seam コンポーネントを Groovy で記述するのは Java で記述するのと同じで、 Seam コンポーネントとしてクラスに目印をつけるためにアノテーションを使います。

例12.2 Groovy による Seam コンポーネントの記述

```
@Scope(ScopeType.SESSION)
@Name("bookingList")
class BookingListAction implements Serializable
{
    @In EntityManager em
    @In User user
    @DataModel List<Booking> bookings
    @DataModelSelection Booking booking
    @Logger Log log

    @Factory
    public void getBookings()
    {
        bookings = em.createQuery('''
        select b from Booking b
        where b.user.username = :username
        order by b.checkinDate''').
            setParameter("username", user.username).
            getResultList()
    }

    public void cancel()
    {
        log.info("Cancel booking: #{bookingList.booking.id}
                 for #{user.username}")
        Booking cancelled = em.find(Booking.class, booking.id)
        if (cancelled != null) em.remove( cancelled )
           getBookings()
           FacesMessages.instance().add("Booking cancelled for
confirmation
                                        number #
{bookingList.booking.id}",
```

```
                                                    new Object[0])
        }
    }
```

12.2.3. seam-gen

seam-gen は Groovy と透過的に作用し合います。 seam-gen で生成されるプロジェクトでは開発環境を一切追加することなく Groovy コードを記述できます。 Groovy エンティティを記述する場合には **.groovy** ファイルを **src/main** に置くだけです。 アクションを記述する場合には **.groovy** ファイルを **src/hot** に置くだけです。

12.3. デプロイ

Groovy クラスのデプロイは Java クラスのデプロイと同様です。 JavaBean コンポーネントのクラスと同様に、Seam では GroovyBean コンポーネントのクラスをアプリケーションを再起動することなく再デプロイすることができます。

12.3.1. Groovy コードのデプロイ

Groovy のエンティティ、 セッション Bean、 コンポーネントはすべてデプロイにコンパイルを必要とします。 **groovyc** ant タスクを使用します。 コンパイルされると、 Groovy のクラスは Java クラスとまったく同じになるため、 アプリケーションサーバーはどちらも同等に扱います。これにより Groovy と Java のコードをシームレスに融合させることができます。

12.3.2. 開発時のネイティブ .groovy ファイルのデプロイ

Seam では増分ホットデプロイメントモードでの **.groovy** ファイルのホットデプロイをサポートしています (コンパイルしないでデプロイを行う)。 このモードは開発のみとなり、 修正 / テストの高速周期を実現します。 「Seam と増分ホットデプロイメント」の設定方法に従って **.groovy** ホットデプロイメントを設定します。 Groovy コード (**.groovy** ファイル) を **WEB-INF/dev** ディレクトリにデプロイします。 アプリケーションまたはアプリケーションサーバーを再起動することなく、 GroovyBean コンポーネントは増加的にデプロイを行うようになります。

注記

ネイティブ **.groovy** ファイルのデプロイメントには通常の Seam ホットデプロイメントと同じ制約があります。

- コンポーネントは JavaBeans または GroovyBeans でなければなりません。 EJB3 Bean は使用できません。
- エンティティはホットデプロイできません。
- ホットデプロイ可能なコンポーネントは **WEB-INF/dev** の外部にデプロイされたクラスからは見えません。
- Seam デバックモードを有効にしなければなりません。

12.3.3. seam-gen

Seam-gen は Groovy ファイルのデプロイおよびコンパイルを透過的にサポートしています。 これには開発時に使用可能なネイティブ **.groovy** ファイルのホットデプロイメントも含まれます。 WAR タイ

プのプロジェクトでは **src/hot** の Java と Groovy のクラス群は自動的に増分ホットデプロイメントの対象となります。 実稼働モードでは Groovy ファイルはデプロイメントの前にコンパイルされます。

examples/groovybooking には、 完全に Groovy で記述された増分ホットデプロイメントに対応する Booking デモがあります。

第13章 SEAM アプリケーションフレームワーク

Seam はアノテートされた純粋な Java クラスで簡単にアプリケーションを作成することができます。設定や拡張により再利用できる既成のコンポーネントのセットを提供することで一般的なプログラミングタスクをさらに簡単にできます。

Seam Application Framework では Web アプリケーションで基本的なデータベースアクセスを行う場合に Hibernate または JPA を使用することにより記述しなければならないコード量を少なくすることができます。 フレームワークには理解しやすく必要に応じ拡張しやすいシンプルなクラスの一部が含まれています。

13.1. はじめに

Seam Application Framework が提供するコンポーネントは、2 つの方法いずれかで利用することができます。1 つ目の方法は、 他の Seam の組み込みコンポーネントと同様に、**components.xml** でコンポーネントのインスタンスをインストールし、設定する方法です。 たとえば、 以下の **components.xml** 設定の一部では **Person** エンティティに対する基本的な CRUD 操作を実行する 1 コンポーネントをインストールします。

```
<framework:entity-home name="personHome" entity-class="eg.Person"
            entity-manager="#{personDatabase}">
<framework:id>#{param.personId}</framework:id>
</framework:entity-home>
```

あまり XML に依存したくない場合は拡張により行うこともできます。

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
@In EntityManager personDatabase;
  public EntityManager getEntityManager() {
    return personDatabase;
  }
}
```

2 つ目の方法の主要な利点はフレームワークのクラスが機能拡張やカスタマイズ向けに設計されているという点であり、 これにより機能の追加や組み込み機能の無効化が容易になります。

もうひとつの利点として、 クラスとして EJB ステートフルセッション Bean (または 純粋の JavaBean コンポーネント) を使うオプションもあります。

```
@Stateful
@Name("personHome")
public class PersonHome extends EntityHome<Person>
                        implements LocalPersonHome { }
```

クラスをステートレスセッション Bean にすることもできます。 この場合、 その名前が **entityManager** であってもインジェクションを使って永続コンテキストを提供**しなければなりません**。

```
@Stateless
@Name("personHome")
public class PersonHome extends EntityHome<Person>
                        implements LocalPersonHome {
  @In EntityManager entityManager;
```

```
  public EntityManager getPersistenceContext() {
    entityManager;
  }
}
```

現時点で、Seam Application Framework は CRUD 用に **EntityHome** と **HibernateEntityHome**、またクエリ用に**EntityQuery** と **HibernateEntityQuery** の 4 つの主要な組み込みコンポーネントを提供しています。

Home と Query コンポーネントはセッションスコープ、イベントスコープ、または対話スコープで機能するように作成されています。 どのスコープを使用するかは、アプリケーションで使用する状態モデルによります。

Seam Application Framework は Seam 管理永続コンテキストでのみ動作します。 デフォルトでは、コンポーネントは **entityManager** という名前の永続コンテキストを期待します。

13.2. HOME オブジェクト

Home オブジェクトは特定のエンティティクラスに永続性操作を行います。 **Person** クラスについて考えてみましょう。

```
@Entity
public class Person {
  @Id private Long id;
  private String firstName;
  private String lastName;
  private Country nationality;
  //getters and setters...
}
```

personHome コンポーネントを定義するには、以下のように設定を通じて行うか

```
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

拡張で行います。

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {}
```

Home オブジェクトは **persist()**、 **remove()**、 **update()**、 **getInstance()** のような動作を提供します。 **remove()** または **update()** を呼び出す前に、**setId()** メソッドを用いて対象のオブジェクトの識別子を設定する必要があります。

たとえば、 Home を JSF ページから直接利用することができます。

```
<h1>Create Person</h1>
<h:form>
  <div>
    First name: <h:inputText value="#{personHome.instance.firstName}"/>
  </div>
  <div>
    Last name:  <h:inputText value="#{personHome.instance.lastName}"/>
  </div>
  <div>
```

```
    <h:commandButton value="Create Person"
       action="#{personHome.persist}"/>
  </div>
</h:form>
```

Person は **person** で参照できると便利なため、**components.xml** にその行を加えます (設定を使用している場合)。

```
<factory name="person" value="#{personHome.instance}"/>
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

拡張を使用している場合は、 **PersonHome** に **@Factory** メソッドを追加できます。

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
  @Factory("person")
  public Person initPerson() {
    return getInstance();
  }
}
```

これで以下のように JSF ページの記述が簡単になります。

```
<h1>Create Person</h1>
<h:form>
  <div>
    First name: <h:inputText value="#{person.firstName}"/>
  </div>
  <div>
    Last name: <h:inputText value="#{person.lastName}"/>
  </div>
  <div>
    <h:commandButton value="Create Person"
       action="#{personHome.persist}"/>
  </div>
</h:form>
```

これが新しい **Person** のエントリを作成するために必要なすべてのコードです。 データベースの既存の **Person** エントリを表示、更新、削除できるようにしたい場合は**PersonHome** にそのエントリの識別子を渡すことが必要です。 これを行うにはページパラメータの使用が適しています。

```
<pages>
  <page view-id="/editPerson.jsp">
    <param name="personId" value="#{personHome.id}"/>
  </page>
</pages>
```

これで JSF ページにこれらの機能を追加することができます。

```
<h1>
  <h:outputText rendered="#{!personHome.managed}" value="Create Person"/>
  <h:outputText rendered="#{personHome.managed}" value="Edit Person"/>
</h1>
<h:form>
```

```
  <div>
    First name: <h:inputText value="#{person.firstName}"/>
  </div>
  <div>
    Last name: <h:inputText value="#{person.lastName}"/>
  </div>
  <div>
    <h:commandButton value="Create Person" action="#{personHome.persist}"
       rendered="#{!personHome.managed}"/>
    <h:commandButton value="Update Person" action="#{personHome.update}"
       rendered="#{personHome.managed}"/>
    <h:commandButton value="Delete Person" action="#{personHome.remove}"
       rendered="#{personHome.managed}"/>
  </div>
</h:form>
```

要求パラメータがないページにリンクする場合、**Create Person** としてページが表示されます。**personId** 要求パラメータに値を与えると **Edit Person** ページが表示されます。

nationality を初期化して **Person** エントリを作成しなければならない場合を考えてみましょう。これも簡単にできます。設定を使う場合は、以下のとおりです。

```
<factory name="person" value="#{personHome.instance}"/>
<framework:entity-home name="personHome" entity-class="eg.Person"
           new-instance="#{newPerson}"/>
<component name="newPerson" class="eg.Person">
  <property name="nationality">#{country}</property>
</component>
```

また、拡張で行うには次のようにします。

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
  @In Country country;
  @Factory("person")
  public Person initPerson() {
    return getInstance();
  }
  protected Person createInstance() {
    return new Person(country);
  }
}
```

Country は、例えば、**CountryHome** という別の Home オブジェクトの管理下のオブジェクトとすることもできます。

アソシエーションの管理など、より洗練された操作を実現するのも **PersonHome** にメソッドを追加するだけでできるようになります。

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
  @In Country country;
  @Factory("person")
  public Person initPerson() {
    return getInstance();
```

```
  }
  protected Person createInstance() {
    return new Person(country);
  }
  public void migrate() {
    getInstance().setCountry(country);
    update();
  }
}
```

トランザクションが成功すると (**persist()**、 **update()**、 または **remove()** への呼び出し)、 Home オブジェクトは **org.jboss.seam.afterTransactionSuccess** イベントを引き起こします。 このイベントを監視することで元になるエンティティが変更された場合にクエリをリフレッシュすることができます。 任意のエンティティの永続化、 更新、 削除が行われたときに特定のクエリをリフレッシュしたいだけの場合は **org.jboss.seam.afterTransactionSuccess.<name>** を監視します (**<name>** がそのエンティティ名です)。

Home オブジェクトは操作が成功すると自動的に Faces のメッセージを表示します。 メッセージをカスタマイズする場合も次のように設定を使用します。

```
<factory name="person" value="#{personHome.instance}"/>
<framework:entity-home name="personHome" entity-class="eg.Person"
           new-instance="#{newPerson}">
  <framework:created-message>
    New person #{person.firstName} #{person.lastName} created
  </framework:created-message>
  <framework:deleted-message>
    Person #{person.firstName} #{person.lastName} deleted
  </framework:deleted-message>
  <framework:updated-message>
    Person #{person.firstName} #{person.lastName} updated
  </framework:updated-message>
</framework:entity-home>
<component name="newPerson" class="eg.Person">
  <property name="nationality">#{country}</property>
</component>
```

拡張で行うには以下のようにします。

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {
  @In Country country;
  @Factory("person")
  public Person initPerson() {
    return getInstance();
  }
  protected Person createInstance() {
    return new Person(country);
  }
  protected String getCreatedMessage() {
    return createValueExpression("New person #{person.firstName}
                                          #{person.lastName}
created");
  }
  protected String getUpdatedMessage() {
```

```
    return createValueExpression("Person #{person.firstName}
                                        #{person.lastName} updated");
  }
  protected String getDeletedMessage() {
  return createValueExpression("Person #{person.firstName}
                                       #{person.lastName} deleted");
  }
}
```

メッセージ定義における最良の方法は Seam に対して既知のリソースバンドル (デフォルトでは **messages** という名前のバンドル) にそのメッセージを定義することです。

```
Person_created=New person #{person.firstName} #{person.lastName} created
Person_deleted=Person #{person.firstName} #{person.lastName} deleted
Person_updated=Person #{person.firstName} #{person.lastName} updated
```

この方法により国際化に対応でき、コードや設定ファイルの外観を良い状態に保ちます。

13.3. QUERY オブジェクト

データベース内の **Person** インスタンスの全一覧が必要な場合は Query オブジェクトを以下のように使うことができます。

```
<framework:entity-query name="people" ejbql="select p from Person p"/>
```

JSF ページからそれを使うことができます。

```
<h1>List of people</h1>
<h:dataTable value="#{people.resultList}" var="person">
  <h:column>
    <s:link view="/editPerson.jsp"
            value="#{person.firstName} #{person.lastName}">
      <f:param name="personId" value="#{person.id}"/>
    </s:link>
  </h:column>
</h:dataTable>
```

ページネーションに対応させる必要がある場合は

```
<framework:entity-query name="people" ejbql="select p from Person p"
           order="lastName" max-results="20"/>
```

表示するページを決めるページパラメータを使います。

```
<pages>
  <page view-id="/searchPerson.jsp">
    <param name="firstResult" value="#{people.firstResult}"/>
  </page>
</pages>
```

ページネーションを管理する JSF のコードは若干繁雑ですが許容範囲内です。

```
<h1>Search for people</h1>
```

```
<h:dataTable value="#{people.resultList}" var="person">
  <h:column>

    <s:link view="/editPerson.jsp"
            value="#{person.firstName} #{person.lastName}">
      <f:param name="personId" value="#{person.id}"/>
    </s:link>

  </h:column>

</h:dataTable>

<s:link view="/search.xhtml" rendered="#{people.previousExists}"
        value="First Page">
  <f:param name="firstResult" value="0"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.previousExists}"
        value="Previous Page">
  <f:param name="firstResult" value="#{people.previousFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}"
        value="Next Page">
  <f:param name="firstResult" value="#{people.nextFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}"
        value="Last Page">
  <f:param name="firstResult" value="#{people.lastFirstResult}"/>
</s:link>
```

実際の検索画面ではユーザーはオプションで検索基準を入力でき、検索結果を絞りこむことができます。**Query** オブジェクトを使うとこのユースケースに対応するオプションの制約を指定できます。

```
<component name="examplePerson" class="Person"/>
<framework:entity-query name="people" ejbql="select p from Person p"
          order="lastName" max-results="20">
  <framework:restrictions>

    <value>
      lower(firstName) like lower(concat(#{examplePerson.firstName},'%&'))
    </value>

    <value>
      lower(lastName) like lower(concat(#{examplePerson.lastName},'%&'))
    </value>

  </framework:restrictions>
</framework:entity-query>
```

上記の例では「**example**」オブジェクトの使用について留意してください。

```
<h1>Search for people</h1>
<h:form>
```

```
  <div>
    First name: <h:inputText value="#{examplePerson.firstName}"/>
  </div>

  <div>
    Last name: <h:inputText value="#{examplePerson.lastName}"/>
  </div>

  <div>
    <h:commandButton value="Search" action="/search.jsp"/>
  </div>

</h:form>

<h:dataTable value="#{people.resultList}" var="person">
  <h:column>
    <s:link view="/editPerson.jsp"
       value="#{person.firstName} #{person.lastName}">
      <f:param name="personId" value="#{person.id}"/>
    </s:link>
  </h:column>
</h:dataTable>
```

元となるエンティティが変更されたときにそのクエリをリフレッシュするには **org.jboss.seam.afterTransactionSuccess** イベントを監視します。

```
<event type="org.jboss.seam.afterTransactionSuccess">
  <action execute="#{people.refresh}" />
</event>
```

または、 **PersonHome** で person エンティティの永続化、 更新、 または削除が行われた場合にクエリをリフレッシュする場合は次のようにします。

```
<event type="org.jboss.seam.afterTransactionSuccess.Person">
  <action execute="#{people.refresh}" />
</event>
```

残念ながら、Query オブジェクトは **join fetch** のクエリとはうまく動作しません。これらのクエリでのページネーションの使用は推奨しません。 **getCountEjbql()** を無効化して、結果の合計数を計算する独自の方法を実装する必要があります。

本項の例はすべて設定による再利用を示していますが、拡張により再利用を行うことも同様に可能です。

13.4. CONTROLLER オブジェクト

Controller クラスとそのサブクラス (**EntityController**、 **HibernateEntityController**、 **BusinessProcessController**) は Seam Application Framework のオプションとなる部分です。よく使用される組み込みのコンポーネントやコンポーネントメソッドへの便利なアクセス方法を提供します。このクラスを使用するとキーストロークを節約でき、また新しいユーザーにとっては Seam に組み込まれている豊富な機能を探る素晴らしい足掛かりとなります。

たとえば、 **RegisterAction** (Seam 登録のサンプルより) は次のようになります。

```
@Stateless
@Name("register")
public class RegisterAction extends EntityController implements Register {
  @In private User user;
  public String register() {
    List existing = createQuery("select u.username from
                                 User u where u.username=:username").
                                 setParameter("username",
                                 user.getUsername()).getResultList();
    if ( existing.size()==0 ) {
      persist(user);
      info("Registered new user #{user.username}");
      return "/registered.jspx";
    } else {
      addFacesMessage("User #{user.username} already exists");
      return null;
    }
  }
}
```

第14章 SEAM と JBOSS RULES

Seam では、Seam コンポーネントまたは jBPM プロセス定義から JBoss Rules (Drools) の ルールベースの呼び出しが容易になります。

14.1. ルールをインストールする

最初のステップは、 Seam コンテキスト変数で **org.drools.RuleBase** のインスタンスを使用可能にすることです。 テスト目的で、 Seam はクラスパスから静的なルール一式をコンパイルする組み込みコンポーネントを提供しています。 このコンポーネントは **components.xml** を使ってインストールできます。

```
<drools:rule-base name="policyPricingRules">
  <drools:rule-files>
    <value>policyPricingRules.drl</value>
  </drools:rule-files>
</drools:rule-base>
```

このコンポーネントは、 DRL (**.drl**) 一式、 またはデシジョンテーブル (**.xls**) ファイルからルールをコンパイルし、 Seam **APPLICATION** コンテキストに **org.drools.RuleBase** のインスタンスをキャッシュします。 ルール駆動型アプリケーションには複数のルールベースのインストールが必要となる可能性が高いので留意してください。

Drools DSL を使用したい場合は DSL 定義も指定する必要があります。

```
<drools:rule-base name="policyPricingRules" dsl-file="policyPricing.dsl">
  <drools:rule-files>
    <value>policyPricingRules.drl</value>
  </drools:rule-files>
</drools:rule-base>
```

RuleBaseConfiguration でカスタムの結果例外ハンドラを登録する場合はそのハンドラを記述する必要があります。 次のサンプルで示します。

```
@Scope(ScopeType.APPLICATION)
@Startup
@Name("myConsequenceExceptionHandler")
public class MyConsequenceExceptionHandler
       implements ConsequenceExceptionHandler, Externalizable {
  public void readExternal(ObjectInput in) throws IOException,
                                          ClassNotFoundException { }

  public void writeExternal(ObjectOutput out) throws IOException { }

  public void handleException(Activation activation,
                              WorkingMemory workingMemory,
                              Exception exception) {
    throw new ConsequenceException( exception, activation.getRule() );
  }
}
```

次にこれを登録します。

```
<drools:rule-base name="policyPricingRules"
```

```
                    dsl-file="policyPricing.dsl"
                    consequence-exception-handler=
                      "#{myConsequenceExceptionHandler}">
    <drools:rule-files>
      <value>policyPricingRules.drl</value>
    </drools:rule-files>
  </drools:rule-base>
```

ほとんどのルール駆動型アプリケーションでは、 ルールは動的にデプロイ可能でなければなりません。 RuleBase の管理に Drools RuleAgent を使用すると便利です。 RuleAgent は Drools ルールサーバー (BRMS) またはローカルファイルレポジトリにあるホットデプロイルールのパッケージに接続することができます。 RulesAgent 管理の RuleBase も **components.xml** で設定が可能です。

```
<drools:rule-agent name="insuranceRules"
        configurationFile="/WEB-INF/deployedrules.properties" />
```

プロパティファイルにはその RulesAgent に固有のプロパティが含まれます。 Drools サンプルディストリビューションからの設定ファイルの例を示します。

```
newInstance=true
url=http://localhost:8080/drools-jbrms/org.drools.brms.JBRMS/package/
           org.acme.insurance/fmeyer
localCacheDir=/Users/fernandomeyer/projects/jbossrules/drools-examples/
               drools-examples-brms/cache
poll=30
name=insuranceconfig
```

また、 設定ファイルを避けコンポーネントで直接オプションを設定することも可能です。

```
<drools:rule-agent name="insuranceRules"
        url="http://localhost:8080/drools-jbrms/org.drools.brms.JBRMS/
                    package/org.acme.insurance/fmeyer"
        local-cache-dir="/Users/fernandomeyer/projects/jbossrules/
                         drools-examples/drools-examples-brms/cache"
        poll="30"
 configuration-name="insuranceconfig" />
```

次に、 各対話に対して **org.drools.WorkingMemory** インスタンスを使用可能にする必要があります (各 **WorkingMemory** は現在の対話に関連する fact を蓄積します)。

```
<drools:managed-working-memory name="policyPricingWorkingMemory"
        auto-create="true" rule-base="#{policyPricingRules}"/>
```

ruleBase 設定プロパティでルールベースに対して **policyPricingWorkingMemory** を参照している点に注目してください。

イベントリスナーを WorkingMemory に追加することで、ルール実行やアサートされるオブジェクトなど、ルールエンジンのイベントを通知する方法を追加することもできます。

```
<drools:managed-working-memory name="policyPricingWorkingMemory"
                               auto-create="true"
                               rule-base="#{policyPricingRules}">
  <drools:event-listeners>
    <value>org.drools.event.DebugWorkingMemoryEventListener</value>
```

```
    <value>org.drools.event.DebugAgendaEventListener</value>
  </drools:event-listeners>
</drools:managed-working-memory>
```

14.2. SEAM コンポーネントからのルールの使用

これで **WorkingMemory** を任意の Seam コンポーネントにインジェクトし、 fact をアサートしてルールを実行することができます。

```
@In WorkingMemory policyPricingWorkingMemory;
@In Policy policy;
@In Customer customer;

public void pricePolicy() throws FactException {
  policyPricingWorkingMemory.insert(policy);
  policyPricingWorkingMemory.insert(customer);
  policyPricingWorkingMemory.fireAllRules();
}
```

14.3. JBPM プロセス定義からのルールの使用

ルールベースは、ページプローまたはビジネスプロセス定義のいずれかで jBPM アクションハンドラ、決定ハンドラまたはアサイメントハンドラとして動作することができます。

```
<decision name="approval">
  <handler class="org.jboss.seam.drools.DroolsDecisionHandler">
    <workingMemoryName>orderApprovalRulesWorkingMemory</workingMemoryName>
    <assertObjects>
      <element>#{customer}</element>
      <element>#{order}</element>
      <element>#{order.lineItems}</element>
    </assertObjects>
  </handler>

  <transition name="approved" to="ship">
    <action class="org.jboss.seam.drools.DroolsActionHandler">
      <workingMemoryName>shippingRulesWorkingMemory</workingMemoryName>
      <assertObjects>
        <element>#{customer}</element>
        <element>#{order}</element>
        <element>#{order.lineItems}</element>
      </assertObjects>
    </action>
  </transition>
  <transition name="rejected" to="cancelled"/>
</decision>
```

<assertObjects> エレメントは **WorkingMemory** に fact としてアサートされるオブジェクトの集合または 1 オブジェクトを返す EL 式を指定します。

jBPM タスク割り当てに対する Drools の使用もサポートされます。

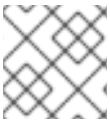
```
<task-node name="review">
```

```
  <task name="review" description="Review Order">
    <assignment handler="org.jboss.seam.drools.DroolsAssignmentHandler">
      <workingMemoryName>
        orderApprovalRulesWorkingMemory
      </workingMemoryName>
      <assertObjects>
        <element>#{actor}</element>
        <element>#{customer}</element>
        <element>#{order}</element>
        <element>#{order.lineItems}</element>
      </assertObjects>
    </assignment>
  </task>
  <transition name="rejected" to="cancelled"/>
  <transition name="approved" to="approved"/>
</task-node>
```

特定のオブジェクトが Drools グローバルとして使用可能です。 jBPM **Assignable** は **assignable** として、 Seam **Decision** オブジェクトは **decision** として使用可能です。 decision を処理するルールは **decision.setOutcome"result")** を呼び出して決定結果を確定するはずです。assignment を実行するルールは **Assignable** を持つ actor IDを設定するはずです。

```
package org.jboss.seam.examples.shop
import org.jboss.seam.drools.Decision
global Decision decision
rule "Approve Order For Loyal Customer"
  when
    Customer( loyaltyStatus == "GOLD" )
    Order( totalAmount <= 10000 )
  then
    decision.setOutcome("approved");
end
```

```
package org.jboss.seam.examples.shop
import org.jbpm.taskmgmt.exe.Assignable
global Assignable assignable
rule "Assign Review For Small Order"
  when
    Order( totalAmount <= 100 )
  then
    assignable.setPooledActors( new String[] {"reviewers"} );
end
```



注記

Drools についての詳細は http://www.drools.org を参照してください。

重要

Seam はシンプルなルールを実装するのには十分な Drools の依存性を同梱しています。機能を追加する場合は、Drools の完全ディストリビューションをダウンロードしてから必要に応じて依存性を追加してください。

第15章 セキュリティ

15.1. 概要

Seam Security API は Seam ベースのアプリケーションに以下のような数多くのセキュリティ関連機能を提供します。

- 認証 – あらゆるセキュリティプロバイダに対してユーザー認証を可能にする、拡張可能な JAAS ベースの認証層を提供します。
- アイデンティティ管理 – ランタイムに Seam アプリケーションのユーザーとロールを管理する API を提供します。
- 承認 – ユーザーのロール、 永続的なルールベースのパーミッション、 カスタマイズされたセキュリティロジックを簡単に実装できるプラグイン可能なパーミッションリゾルバをサポートする非常に包括的な承認フレームワークを提供します。
- パーミッション管理 – アプリケーションのセキュリティポリシーの管理を容易にする組み込みの Seam コンポーネント一式を提供します。
- CAPCHA サポート – Seam ベースのサイトに有害となる自動化ソフトウェア / スクリプトから保護するためのサポートを提供します。

本章ではこれらの機能について詳しく説明していきます。

15.2. セキュリティを無効にする

Seam Security を無効にする必要がある状況が発生することがあります (例えばユニットテスト中やネイティブ JAAS などの別のセキュリティ手段を使用する場合など)。 セキュリティのインフラストラクチャを無効にするには、 静的メソッドとなる **`Identity.setSecurityEnabled(false)`** を呼び出します。 ただし、 アプリケーションを設定したい場合は代わりに次の設定を **`components.xml`** で制御する方が便利でしょう。

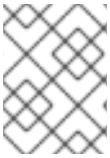
- エンティティのセキュリティ
- Hibernate セキュリティインターセプタ
- Seam Security インターセプタ
- ページ単位の制約
- サーブレット API セキュリティ統合

本章ではユーザーのアイデンティティ (認証) の確立とアクセス制約 (承認) の規定に使用できる非常に多くのオプションについて説明します。 セキュリティモデルの基礎となる認証から見ていくことにします。

15.3. 認証

Seam Security は JAAS (Java Authentication and Authorization Service) をベースにした承認機能を提供し、ユーザー認証の処理に堅牢で高度に設定可能な API を提供しています。求めている認証がこれほど複雑ではない場合は、Seam には単純化された認証メソッドも備わっています。

15.3.1. 認証コンポーネントの設定



注記

Seam のアイデンティティ管理機能を使用する場合は、 認証コンポーネントの作成は不要となるので本章を省略しても構いません。

Seam の単純化された認証メソッドでは、独自の Seam コンポーネントのひとつに対して認証を委譲する組み込みの JAAS ログインモジュール (**SeamLoginModule**) を使用します (このモジュールは余分な設定ファイルを必要とせず、 Seam 内で事前設定され同梱されています)。これにより、 使用するアプリケーションで提供されるエンティティクラスで認証メソッドを記述する、 または別のサードパーティのプロバイダ経由で認証を行うことができます。 この単純化された認証を設定するには **components.xml** で下記のように **identity** コンポーネントを設定する必要があります。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:security="http://jboss.com/products/seam/security"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd
               http://jboss.com/products/seam/security
               http://jboss.com/products/seam/security-2.2.xsd">

<security:identity authenticate-method="#{authenticator.authenticate}"/>

</components>
```

#{authenticator.authenticate} は **authenticator** コンポーネントの **authenticate** メソッドがユーザーの認証に使用されることを示すメソッドバインディングです。

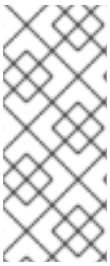
15.3.2. 認証メソッドの記述

components.xml 内の **identity** に対して指定される **authenticate-method** プロパティは、**SeamLoginModule** で使用されるメソッドを指定してユーザー認証を行います。 このメソッドはパラメータを取らないため、 認証が成功したか失敗したかを示す Boolean を返します。 ユーザー名とパスワードは **Credentials.getUsername()** と **Credentials.getPassword()** からそれぞれ取得します (**credentials** コンポーネントへの参照は **Identity.instance().getCredentials()** から 取得可能)。 ユーザーがメンバーとなるすべてのロールは **Identity.addRole()** で割り当ててください。 以下に POJO コンポーネント内の認証メソッドの完全な例を示します。

```
@Name("authenticator")
public class Authenticator {
  @In EntityManager entityManager;
  @In Credentials credentials;
  @In Identity identity;

  public boolean authenticate() {
    try {
      User user = (User) entityManager.createQuery(
          "from User where username = :username and password = :password")
          .setParameter("username", credentials.getUsername())
          .setParameter("password", credentials.getPassword())
          .getSingleResult();

      if (user.getRoles() != null) {
```

```
          for (UserRole mr : user.getRoles())
            identity.addRole(mr.getName());
          }

          return true;
        } catch (NoResultException ex) {
          return false;
        }

    }

}
```

上記の例では、**User** と **UserRole** はアプリケーション固有のエンティティ Bean です。 パラメータ **roles** はユーザーがメンバーとなるロールで埋められます。 たとえば、「admin」や「user」など **Set** にリテラル文字列値として追加されます。 ユーザー記録が見つからず**NoResultException** が送出される場合は、 認証メソッドは **false** を返して認証が失敗したことを示します。

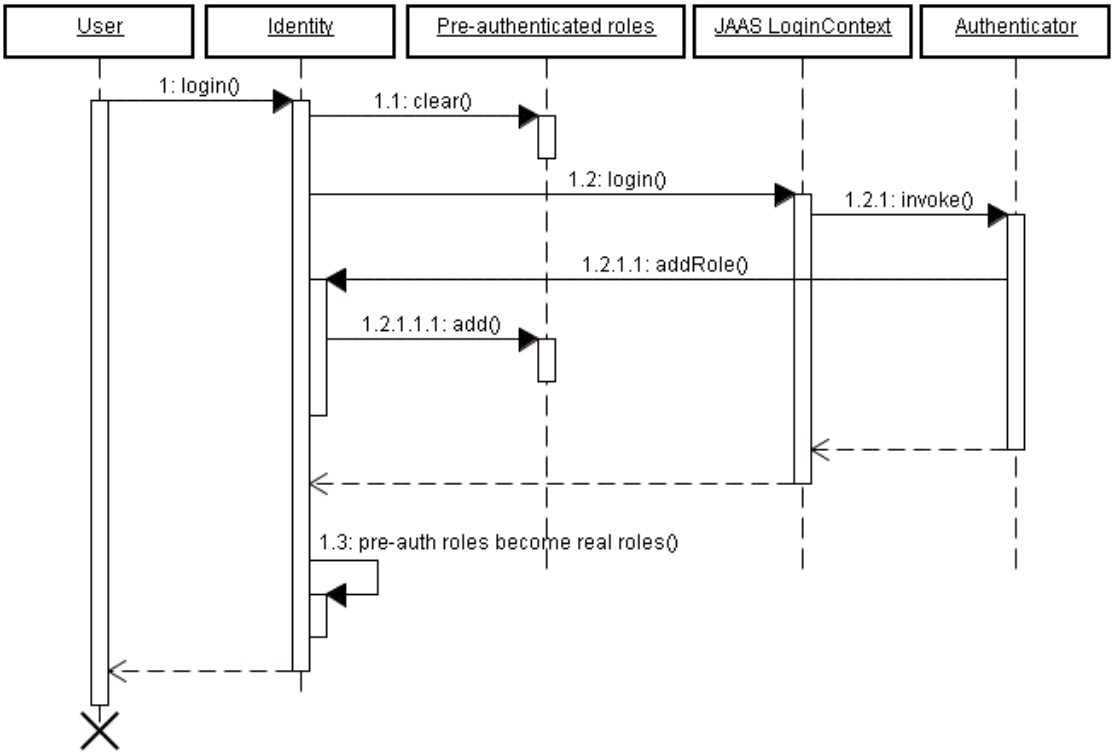


注記

認証メソッドを記述する場合、副次的な影響を受けない最小限の認証メソッドにすることが重要です。 認証メソッドは 1 回の要求で複数回呼び出される可能性があるため、 認証が成功あるいは失敗した時に実行される特殊なコードには、すべてイベントオブザーバーを実装させてください。Seam Security により発生するイベントについての詳細は本章の後半 「セキュリティイベント」 を参照してください。

15.3.2.1. Identity.addRole()

Identity.addRole() メソッドの動作は現在のセッション認証により異なります。 セッションが認証されていない場合、 **addRole()** は認証過程でのみ呼び出されるはずです。 ここで呼び出されると、ロール名は事前認証されたロールの一時リストに配置されます。 認証が成功したら事前認証されたロールが「実際の」ロールとなり、 これらのロールに対する **Identity.hasRole()** 呼び出しは **true** を返します。 以下のシーケンス図に、認証過程におけるその位置付けを明確にするため 1 番目のクラスオブジェクトとして事前認証されたロールリストを示します。



現在のセッションがすでに認証されている場合には、**`Identity.addRole()`** を呼び出すと指定されたロールが直ちに現在のユーザーに付与されます。

15.3.2.2. セキュリティ関連のイベントにイベントオブザーバーを記述する

ログインに成功してユーザーの統計データを更新する必要がある場合、**`org.jboss.seam.security.loginSuccessful`** イベントのイベントオブザーバーを以下のように記述することができます。

```
@In UserStats userStats;

@Observer("org.jboss.seam.security.loginSuccessful")
public void updateUserStats() {
  userStats.setLastLoginDate(new Date());
  userStats.incrementLoginCount();
}
```

このオブザーバーメソッドは認証コンポーネント自体を含めてどこにおいても構いません。 他のセキュリティ関連のイベントについては本章の後半でさらに見ていきます。

15.3.3. ログインフォームの記述

`credentials` コンポーネントは **`username`** および **`password`** のプロパティを提供し、最も一般的な認証シナリオに対応できるようになっています。 これらのプロパティはログインフォームのユーザー名とパスワードのフィールドに直接バインドされることができます。 これらのプロパティを設定した後 **`identity.login()`** を呼び出すと、 与えた資格情報でそのユーザーの認証が行われます。 簡単なログインフォームの例を以下に示します。

```
<div>
  <h:outputLabel for="name" value="Username"/>
  <h:inputText id="name" value="#{credentials.username}"/>
```

```
</div>

<div>
  <h:outputLabel for="password" value="Password"/>
  <h:inputSecret id="password" value="#{credentials.password}"/>
</div>

<div>
  <h:commandButton value="Login" action="#{identity.login}"/>
</div>
```

同様に、 **#{identity.logout}** を呼び出すとユーザーはログアウトされます。 この動作により、 現在認証されているユーザーのセキュリティの状態を削除し、そのユーザーのセッションを無効化します。

15.3.4. 設定のまとめ

以下の簡単な 3 つのステップを実行して認証を設定します。

- 認証メソッドを **components.xml** に設定します。
- 認証メソッドを記述します。
- ユーザーを認証できるようログインフォームを記述します。

15.3.5. Remember Me

Seam Security は多くの Web ベースアプリケーションに共通した **Remember Me** 機能の 2 種類のモードに対応しています。ユーザー名をユーザーのブラウザにクッキーとして保存しブラウザにパスワードを記憶させるモードと、固有のトークンをクッキーに保存してユーザーがそのサイトに戻ったときにパスワードを入力することなく自動的に認証できるモードです。

警告

これはユーザーにとっては便利ですが、 クライアントのマシンで永続クッキーを使った自動のクライアント認証はクロスサイトスクリプティング (XSS) のセキュリティホールによる影響が拡大されるため危険です。 認証クッキーがない場合、 攻撃側が XSS で盗むことができるクッキーはユーザーの **現在のセッションクッキー** のみです。 このため、 攻撃はユーザーがセッションを開いている間しか発生しません。 永続の **Remember Me** クッキーが盗まれると攻撃側はいつでも認証なしでログインができます。 自動のクライアント認証を使用したい場合は XSS 攻撃に対して Web サイトを保護することが不可欠となります。

ブラウザのベンダーはこの問題に対抗する **Remember Passwords** 機能を導入しました。 ブラウザは特定の Web サイトやドメインへのログインに使用するユーザー名とパスワードを記憶して、 アクティブなセッションがない場合はそのログインフォームに自動入力を行います。 Web サイトでのログインのキーボードショートカットはログインの過程をほぼ「Remember Me」クッキーと同じくらい便利に、かつさらなる安全性を実現しています。 一部のブラウザ (OS X の Safari など) では暗号化したグローバルなオペレーティングシステムのキーチェーンにログインのフォームデータを格納します。 ネットワーク環境ではラップトップとデスクトップ間でこのキーチェーンはユーザーと共に移動が可能です。クッキーは通常同期されません。

自動認証での永続的な **Remember Me** クッキーは広く使用されていますが、 セキュリティ上 不適切です。 ユーザーのログイン名だけを記憶し、 そのユーザー名をログインフォームに入力させる方がはるかに安全です。

デフォルト (安全、 ユーザー名のみ) モードに **Remember Me** 機能を有効にするために特別な設定は必要ありません。 ログインフォームで **Remember Me** チェックボックスを **`rememberMe.enabled`** にバインドするだけです。次の例をみてください。

```
<div>
  <h:outputLabel for="name" value="User name"/>
  <h:inputText id="name" value="#{credentials.username}"/>
</div>

<div>
  <h:outputLabel for="password" value="Password"/>
  <h:inputSecret id="password" value="#{credentials.password}"
redisplay="true"/>
</div>

<div class="loginRow">
  <h:outputLabel for="rememberMe" value="Remember me"/>
  <h:selectBooleanCheckbox id="rememberMe" value="#{rememberMe.enabled}"/>
</div>
```

15.3.5.1. トークンベースの Remember Me 認証

トークンベースの自動機能 **Remember Me** を使用するには、 まずトークンストアを設定する必要があります。 この認証トークンは一般的にはデータベース内に格納されます。 Seam はこの方法に対応しま

すが、**org.jboss.seam.security.TokenStore** インターフェースを使って独自のトークンストアを実装することも可能です。本項では **JpaTokenStore** 実装を使用して認証トークンをデータベーステーブル内に保存することを前提としています。

まず最初にトークンを保持させる新しいエンティティを作ります。以下に可能なエンティティの構造を示します。

```
@Entity
public class AuthenticationToken implements Serializable {
  private Integer tokenId;
  private String username;
  private String value;

  @Id @GeneratedValue
  public Integer getTokenId() {
    return tokenId;
  }

  public void setTokenId(Integer tokenId) {
     this.tokenId = tokenId;
  }

  @TokenUsername
  public String getUsername() {
     return username;
  }

  public void setUsername(String username) {
    this.username = username;
  }

  @TokenValue
  public String getValue() {
    return value;
  }

  public void setValue(String value) {
    this.value = value;
  }
}
```

このコードから分かるように、エンティティのユーザー名とトークンのプロパティの設定に **@TokenUsername** と **@TokenValue** という特殊なアノテーションが使われています。 認証トークンを保持するエンティティにはこれらのアノテーションが必要です。

次に、このエンティティ Bean で認証トークンを保管、取得するために **JpaTokenStore** を設定します。そのために **components.xml** の **token-class** 属性を指定します。

```
<security:jpa-token-store
     token-class="org.jboss.seam.example.seamspace.AuthenticationToken"/>
```

最後のステップとして **components.xml** に **RememberMe** コンポーネントを設定します。その**mode** は **autoLogin** に設定してください。

```
<security:remember-me mode="autoLogin"/>
```

これで **Remember Me** チェックボックスをチェックしたユーザーは自動的に認証されるようになります。

ユーザーがサイトを再訪した時に確実に自動認証が行われるよう **components.xml** に以下のセクションを含めてください。

```
<event type="org.jboss.seam.security.notLoggedIn">
  <action execute="#{redirect.captureCurrentView}"/>
  <action execute="#{identity.tryLogin()}"/>
</event>
<event type="org.jboss.seam.security.loginSuccessful">
  <action execute="#{redirect.returnToCapturedView}"/>
</event>
```

15.3.6. セキュリティ例外の処理

セキュリティエラー発生時にユーザーがデフォルトの基本エラーページを受け取らないようにするため、 **pages.xml** を編集してもう少し見栄えのするページにリダイレクトしてください。セキュリティ API により送出される例外には主として 2 つのタイプがあります。

- **NotLoggedInException** - ユーザーがログインすることなく制限された操作やページにアクセスしようとするとこの例外が送出されます。
- **AuthorizationException** – 既にログインしているユーザーがアクセス許可を持たない制限された操作やページにアクセスしようとしたときにのみこの例外が送出されます。

NotLoggedInException の場合、ユーザーをログインページかユーザー登録ページにリダイレクトしてログイン操作を行えるようにすることをお勧めします。 **AuthorizationException** の場合は、ユーザーをエラーページにリダイレクトした方が良いでしょう。 以下の例では、 この 2 つのセキュリティ例外をリダイレクトする **pages.xml** ファイルを示しています。

```
<pages>

  ...

  <exception class="org.jboss.seam.security.NotLoggedInException">
    <redirect view-id="/login.xhtml">
      <message>You must be logged in to perform this action</message>
    </redirect>
  </exception>

  <exception class="org.jboss.seam.security.AuthorizationException">
    <end-conversation/>
      <redirect view-id="/security_error.xhtml">
        <message>
          You do not have the necessary security privileges to perform
this action.
        </message>
      </redirect>
```

```
  </exception>

</pages>
```

ほとんどの Web アプリケーションでより洗練されたログインのリダイレクト処理を必要とします。Seam では特別な機能も備えており、次項に概要が記載されています。

15.3.7. ログインのリダイレクト

認証されていないユーザーが特定のビューまたはワイルドカードで指定されたビュー ID へのアクセスを試行する際に、 Seam ではユーザーを以下のようにログイン画面にリダイレクトすることができます。

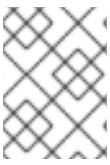
```
<pages login-view-id="/login.xhtml">

  <page view-id="/members/*" login-required="true"/>
...
</pages>
```

注記

これは上記の例外ハンドラと比べて改善されていますが、併用するとよいでしょう。

ユーザーがログイン後、 ログインを必要とした操作にユーザーを自動的にリダイレクトする場合を考えてみます。 次のイベントリスナーを **components.xml** に追加すると、 ログインせずに行われた制限ビューへのアクセス試行は記憶されます。 ログインに成功すると、ユーザーは当初の要求時に存在したページパラメータを持つ当該ビューにリダイレクトされます。

```
<event type="org.jboss.seam.security.notLoggedIn">
  <action execute="#{redirect.captureCurrentView}"/>
</event>

<event type="org.jboss.seam.security.postAuthenticate">
  <action execute="#{redirect.returnToCapturedView}"/>
</event>
```



注記

ログインのリダイレクトは対話スコープのメカニズムとして実装されるため、**authenticate()** メソッドの中で対話を終了させないでください。

15.3.8. HTTP 認証

推奨されませんが、どうしても必要であれば Seam は HTTP Basic あるいは HTTP Digest (RFC 2617) メソッドでの認証方法を提供しています。いずれの認証の形を使用する場合でも、まず **components.xml** で **authentication-filter** コンポーネントを有効にする必要があります。

```
<web:authentication-filter url-pattern="*.seam" auth-type="basic"/>
```

ベーシック認証を有効にするには、 **auth-type** を **basic** に設定します。 ダイジェスト認証を有効にするには、 **digest** に設定します。 ダイジェスト認証を使用する場合には **key** と **realm** も設定する必要があります。

```
<web:authentication-filter url-pattern="*.seam" auth-type="digest"
     key="AA3JK34aSDlkj" realm="My App"/>
```

key は任意の文字列です。 **realm** はユーザーが認証される時にユーザーに提供される認証レルム名です。

15.3.8.1. ダイジェスト認証の記述

ダイジェスト認証を使用する場合は、認証クラスは **org.jboss.seam.security.digest.DigestAuthenticator** 抽象クラスを拡張して、**validatePassword()** メソッドを使用しユーザーのプレーンテキストのパスワードとダイジェスト要求を照合する必要があります。 以下はコード例です。

```
public boolean authenticate() {
  try {
    User user = (User) entityManager.createQuery(
         "from User where username = "username")
         .setParameter("username", identity.getUsername())
         .getSingleResult();

    return validatePassword(user.getPassword());
  } catch (NoResultException ex) {
    return false;
  }
}
```

15.3.9. 高度な認証機能

本項では、より複雑なセキュリティ要件に応えられるセキュリティ API で提供されている高度な機能について紹介します。

15.3.9.1. 使用しているコンテナの JAAS の設定

Seam Security API で提供される簡素化された JAAS 設定を使用したくない場合、 **components.xml** に **jaas-config-name** プロパティを追加してシステムのデフォルトの JAAS 設定を使用することができます。 例として、 JBoss AS を使用していて **other**ポリシー (JBoss AS 提供の **UsersRolesLoginModule** ログインモジュールを使用する) を使用したい場合には、**components.xml** は以下のようになります。

```
<security:identity jaas-config-name="other"/>
```

これは単に Seam Security に対して設定された JAAS セキュリティポリシーに基づいて認証を行うように指示をしているだけで、Seam アプリケーションコンテナでユーザーが認証されるわけではないので留意してください。

15.4. アイデンティティ管理

アイデンティティ管理は、バックエンドの動作で使用されるアイデンティティストア (データベース、LDAP など) に関わらず Seam アプリケーションのユーザーとロールの管理に標準の API を提供します。アイデンティティ管理 API の中核となるのが **identityManager** コンポーネントです。それは、ユーザーの作成、 変更および削除、 ロールの許可とその取り消し、 パスワードの変更、 ユーザーアカウントの有効化と無効化、 ユーザーの認証、 ユーザーとロールの一覧表示などを行うための全メソッドを提供します。

使用する前に **identityManager** に **IdentityStore** を 1 つ以上設定する必要があります。 これらのコンポーネントがバックエンドのセキュリティプロバイダと連携して動作します。

15.4.1. IdentityManager の設定

identityManager コンポーネントにより、認証および承認に対して別々のアイデンティティストアを設定することができます。 つまり、 ユーザーを任意のアイデンティティストアに対して認証させることができますが (LDAP ディレクトリなど)、そのユーザーのロールは別のアイデンティティストア (リレーショナルデータベースなど) からロードさせます。

Seam は特に設定を必要としない 2 種類の **IdentityStore** 実装を提供しています。 デフォルトとなる **JpaIdentityStore** はリレーショナルデータベースを使用してユーザーとロールの情報を格納します。 もうひとつの実装は **LdapIdentityStore** で、LDAP ディレクトリを使用してユーザーとロールを格納します。

identityManager コンポーネントには**identityStore** と **roleIdentityStore** の 2 つの設定可能なプロパティがあります。 これらプロパティの値は **IdentityStore** インターフェースを使って Seam コンポーネントを参照する EL 式でなければなりません。 特に設定しないとデフォルトが使用されます (**JpaIdentityStore**)。 また、 **identityStore** プロパティのみを設定した場合には **roleIdentityStore** に同じ値が使用されます。 例えば、 **components.xml** にある次のエントリは **identityManager** がユーザー関連の操作およびロール関連の操作の両方に**LdapIdentityStore** を使用するよう設定します。

```
<security:identity-manager identity-store="#{ldapIdentityStore}"/>
```

下記の例ではユーザーに関する処理は **LdapIdentityStore** を、 またロールに関する処理には **JpaIdentityStore** を使用するよう **identityManager** を設定しています。

```
<security:identity-manager identity-store="#{ldapIdentityStore}"
          role-identity-store="#{jpaIdentityStore}"/>
```

次項からは各アイデンティティのストレージメソッドに関して詳細に説明していきます。

15.4.2. JpaIdentityStore

このメソッドはユーザーおよびロールをリレーショナルデータベースに保存します。 データベース設計およびテーブル構造に柔軟性を持たせるよう設計されています。 特殊なアノテーション一式により、エンティティ Bean によるユーザーとロールの記録保存を可能にします。

15.4.2.1. JpaIdentityStore の設定

JpaIdentityStore を使用する前に、 **user-class** と **role-class** の両方を設定しておく必要があります。 これらのプロパティはユーザーとロールそれぞれの記録保存に使用するエンティティクラス

郡を参照します。 以下の例では、 SeamSpace のサンプルにある **components.xml** ファイルを示しています。

```
<security:jpa-identity-store
          user-class="org.jboss.seam.example.seamspace.MemberAccount"
          role-class="org.jboss.seam.example.seamspace.MemberRole"/>
```

15.4.2.2. エンティティの設定

次のテーブルではユーザーとロールの保存用エンティティ Bean の設定に使用される特殊なアノテーションを説明しています。

表15.1 ユーザーエンティティのアノテーション

アノテーション	ステータス	詳細
@UserPrincipal	必須	このアノテーションはユーザーのユーザー名を含むフィールドまたはメソッドをマークします。

@UserPassword	必須	このアノテーションはユーザーのパスワードを含むフィールドまたはメソッドをマークします。パスワードハッシュに **hash** アルゴリズムを指定することができます。**hash** に指定できる値には **md5**、**sha**、**none** があります。たとえば以下のとおりです。
		`@UserPassword(hash = "md5")`
		`public String getPasswordHash() {`
		`  return passwordHash;`
		`}`
		他のハッシュアルゴリズムを実装する必要がある場合は、 **PasswordHash** を拡張することができます。

@UserFirstName	オプション	このアノテーションはユーザーの名前を含むフィールドまたはメソッドをマークします。
@UserLastName	オプション	このアノテーションはユーザーの姓を含むフィールドあるいはメソッドをマークします。
@UserEnabled	オプション	このアノテーションは有効なユーザーステータスを含むフィールドまたはメソッドをマークします。これは Boolean プロパティとなるはずです。このアノテーションが存在しないと、すべてのユーザーアカウントは有効であるとみなされます。
@UserRoles	必須	このアノテーションはユーザーのロールを含むフィールドまたはメソッドをマークします。このプロパティの詳細については後ほど記載します。

表15.2 ロールエンティティのアノテーション

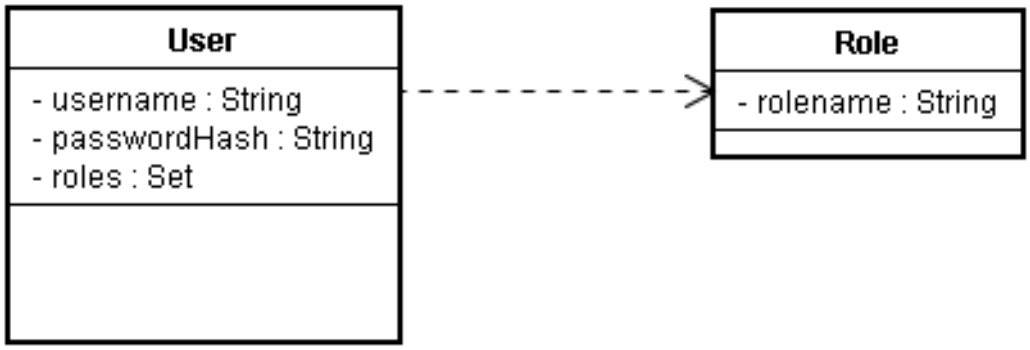
アノテーション	ステータス	詳細
@RoleName	必須	このアノテーションはロール名を含むフィールドまたはメソッドをマークします。
@RoleGroups	オプション	このアノテーションはロールのグループメンバーシップを含むフィールドまたはメソッドをマークします。
@RoleConditional	オプション	このアノテーションはロールが条件付きか否かを示すフィールドまたはメソッドをマークします。条件付きロールについては本章の後半で説明します。

15.4.2.3. エンティティ Bean の例

既に示したように **JpaIdentityStore** は、ユーザーとロールのテーブルのデーターベーススキーマ設計に関してはできるだけ柔軟につくられています。本項では、ユーザーとロール記録を保持することが可能なデータベースのスキーマについて見ていきます。

15.4.2.3.1. 最小限のスキーマの例

ここでは、クロスリファレンステーブル **UserRoles** を使って many-to-many の関係でシンプルなユーザーとロールのテーブルがリンクされています。



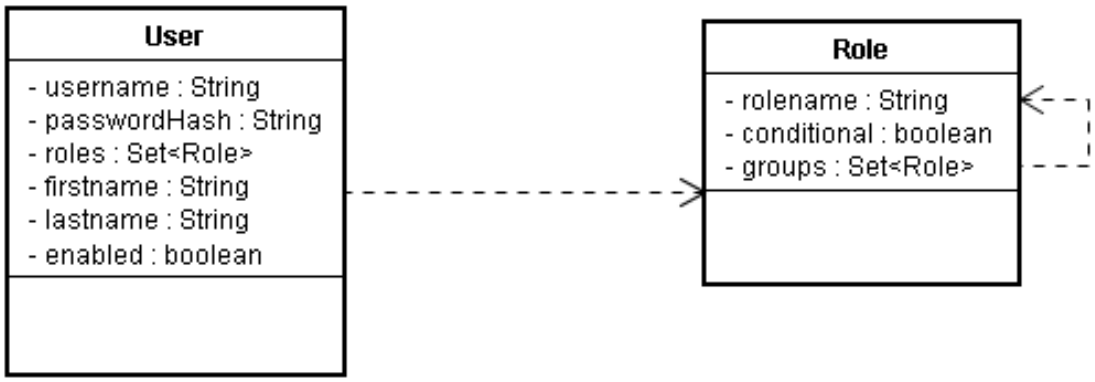

```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
  private Set<Role> roles;

  @Id @GeneratedValue
  public Integer getUserId() { return userId; }
  public void setUserId(Integer userId) { this.userId = userId; }

  @UserPrincipal
  public String getUsername() { return username; }
  public void setUsername(String username) { this.username = username; }

  @UserPassword(hash = "md5")
  public String getPasswordHash() { return passwordHash; }
  public void setPasswordHash(String passwordHash) {
```

```
    this.passwordHash = passwordHash;
  }

  @UserRoles
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "UserRoles",
       joinColumns = @JoinColumn(name = "UserId"),
       inverseJoinColumns = @JoinColumn(name = "RoleId"))
  public Set<Role> getRoles() { return roles; }
  public void setRoles(Set<Role> roles) { this.roles = roles; }
}
```

```
@Entity
public class Role {
  private Integer roleId;
  private String rolename;

  @Id @Generated
  public Integer getRoleId() { return roleId; }
  public void setRoleId(Integer roleId) { this.roleId = roleId; }

  @RoleName
  public String getRolename() { return rolename; }
  public void setRolename(String rolename) { this.rolename = rolename; }
}
```

15.4.2.3.2. 複雑なスキーマの例

この例は前述の最少限のスキーマの例にすべてのオプションフィールドを含ませることによりロールにグループメンバーを許可しています。




```
@Entity
public class User {
  private Integer userId;
  private String username;
  private String passwordHash;
  private Set<Role> roles;
  private String firstname;
  private String lastname;
  private boolean enabled;

  @Id @GeneratedValue
  public Integer getUserId() { return userId; }
```

```
  public void setUserId(Integer userId) { this.userId = userId; }

  @UserPrincipal
  public String getUsername() { return username; }
  public void setUsername(String username) { this.username = username; }

  @UserPassword(hash = "md5")
  public String getPasswordHash() { return passwordHash; }
  public void setPasswordHash(String passwordHash) {
    this.passwordHash = passwordHash;
  }

  @UserFirstName
  public String getFirstname() { return firstname; }
  public void setFirstname(String firstname) {
    this.firstname = firstname;
  }

  @UserLastName
  public String getLastname() { return lastname; }
  public void setLastname(String lastname) { this.lastname = lastname; }

  @UserEnabled
  public boolean isEnabled() { return enabled; }
  public void setEnabled(boolean enabled) { this.enabled = enabled; }

  @UserRoles
  @ManyToMany(targetEntity = Role.class)
  @JoinTable(name = "UserRoles",
    joinColumns = @JoinColumn(name = "UserId"),
    inverseJoinColumns = @JoinColumn(name = "RoleId"))
  public Set<Role> getRoles() { return roles; }
  public void setRoles(Set<Role> roles) { this.roles = roles; }
}
```

```
@Entity
public class Role {
  private Integer roleId;
  private String rolename;
  private boolean conditional;

  @Id @Generated
  public Integer getRoleId() { return roleId; }
  public void setRoleId(Integer roleId) { this.roleId = roleId; }

  @RoleName
  public String getRolename() { return rolename; }
  public void setRolename(String rolename) { this.rolename = rolename; }

  @RoleConditional
  public boolean isConditional() { return conditional; }
  public void setConditional(boolean conditional) {
    this.conditional = conditional;
  }

  @RoleGroups
```

```
    @ManyToMany(targetEntity = Role.class)
    @JoinTable(name = "RoleGroups",
               joinColumns = @JoinColumn(name = "RoleId"),
               inverseJoinColumns = @JoinColumn(name = "GroupId"))
    public Set<Role> getGroups() { return groups; }
    public void setGroups(Set<Role> groups) { this.groups = groups; }

}
```

15.4.2.4. JpaIdentityStore イベント

JpaIdentityStore を **IdentityManager** で使用する場合、 特定の **IdentityManager** メソッドが呼び出されるといくつかのイベントが発生します。

15.4.2.4.1. JpaIdentityStore.EVENT_PRE_PERSIST_USER

このイベントは **IdentityManager.createUser()** の呼び出しに応じて発生します。 ユーザーエンティティがデータベースに対して永続化される直前に、 このイベントが発生してンティティインスタンスをイベントパラメータとして渡します。 このエンティティは **JpaIdentityStore** に設定した **user-class** のインスタンスです。

標準の **createUser()** 機能の一部ではないエンティティフィールドの値を設定する場合にはオブザーバーが便利です。

15.4.2.4.2. JpaIdentityStore.EVENT_USER_CREATED

このイベントは **IdentityManager.createUser()** の呼び出しにも応じて発生します。 ただし、 このイベントはユーザーエンティティがデータベースに対してすでに永続化されてから発生します。 **EVENT_PRE_PERSIST_USER** イベントと同様、 イベントパラメータとしてエンティティのインスタンスを渡します。 連絡先の詳細情報の記録や他のユーザー固有データなど、ユーザーエンティティを参照する他のエンティティを永続化する必要がある場合にこのイベントを監視すると便利です。

15.4.2.4.3. JpaIdentityStore.EVENT_USER_AUTHENTICATED

このイベントは **IdentityManager.authenticate()** を呼び出すと発生します。 このイベントはエンティティインスタンスをイベントパラメータとして渡すため、 認証されるユーザーエンティティの追加のプロパティを読み取る場合に便利です。

15.4.3. LdapIdentityStore

このアイデンティティのストレージメソッドは LDAP ディレクトリにユーザーレコードを保存するよう設計されています。これは高度な設定が可能なため、ユーザーおよびロールの柔軟なディレクトリのストレージを実現しています。次項では、 このアイデンティティストアの設定オプションと設定例について説明していきます。

15.4.3.1. LdapIdentiyStore の設定

以下の表に **LdapIdentityStore** に対して **components.xml** で設定できるプロパティを示します。

表15.3 LdapIdentityStore の設定に関わるプロパティ

プロパティ	デフォルト値	詳細
server-address	**localhost**	LDAP サーバーのアドレスです。
server-port	**389**	LDAP サーバーがリッスンしているポート番号です。
user-context-DN	**ou=Person,dc=acme,dc=com**	ユーザーレコードを含むコンテキストの識別名 (DN) です。
user-DN-prefix	**uid=**	この値は username の前に置かれ、ユーザーレコードを検索します。
user-DN-suffix	**,ou=Person,dc=acme,dc=com**	この値は username の後ろに追加され、ユーザーレコードを検索します。
role-context-DN	**ou=Role,dc=acme,dc=com**	ロールレコードを含むコンテキストの識別名 (DN) です。
role-DN-prefix	**cn=**	この値はロール名の前に置かれ、ロール記録を検索する DN を形成します。
role-DN-suffix	**,ou=Roles,dc=acme,dc=com**	この値はロール名の後ろに追加され、 ロール記録を検索する DN を形成します。
bind-DN	**cn=Manager,dc=acme,dc=com**	LDAP サーバーとバインドするために使用するコンテキストです。
bind-credentials	**secret**	LDAP サーバーとバインドするために使用される資格情報 (パスワード) です。

プロパティ	デフォルト値	詳細
user-role-attribute	**roles**	ユーザーがメンバーであるロールの一覧を含むユーザーレコードの属性名です。
role-attribute-is-DN	**true**	この Boolean プロパティはユーザーレコードのロール属性自体が識別名か否かを示しています。
user-name-attribute	**uid**	ユーザー名を含むユーザーレコードの属性を示します。
user-password-attribute	**userPassword**	ユーザーのパスワードを含むユーザーレコードの属性を示します。
first-name-attribute	**null**	ユーザーの名前を含むユーザーレコードの属性を示します。
last-name-attribute	**sn**	ユーザーの姓を含むユーザーレコードの属性を示します。
full-name-attribute	**cn**	ユーザーのフルネームを含むユーザーレコードの属性を示します。
enabled-attribute	**null**	ユーザーが有効であるかを決定するユーザーレコードの属性を示します。
role-name-attribute	**cn**	ロール名を含むロール記録の属性を示します。
object-class-attribute	**objectClass**	ディレクトリ内のオブジェクトのクラスを決定する属性を示します。

プロパティ	デフォルト値	詳細
role-object-classes	**organizationalRole**	新しいロール記録の作成のためのオブジェクトクラスの配列です。
user-object-classes	**person,uidObject**	新しいユーザーレコード作成のためのオブジェクトクラスの配列です。

15.4.3.2. LdapIdentityStore の設定例

下の設定例では、 擬似ホスト **directory.mycompany.com** で動作している LDAP ディレクトリに **LdapIdentityStore** を設定する方法を示しています。 ユーザーはこのディレクトリ内に **ou=Person,dc=mycompany,dc=com** というコンテキストで保存され、**uid** 属性で識別されます (そのユーザー名に対応する)。 ロールはロール用のコンテキスト **ou=Roles,dc=mycompany,dc=com** に保存され、 ユーザーのエントリから **roles** 属性を通じて参照されます。 ロールのエントリはロール名に対応するロールの一般名 (**cn** 属性) で識別されます。 この例では、 ユーザーは **enabled** 属性の値を **false** に設定すると無効にすることができます。

```
<security:ldap-identity-store
  server-address="directory.mycompany.com"
  bind-DN="cn=Manager,dc=mycompany,dc=com"
  bind-credentials="secret"
  user-DN-prefix="uid="
  user-DN-suffix=",ou=Person,dc=mycompany,dc=com"
  role-DN-prefix="cn="
  role-DN-suffix=",ou=Roles,dc=mycompany,dc=com"
  user-context-DN="ou=Person,dc=mycompany,dc=com"
  role-context-DN="ou=Roles,dc=mycompany,dc=com"
  user-role-attribute="roles"
  role-name-attribute="cn"
  user-object-classes="person,uidObject"
  enabled-attribute="enabled"
/>
```

15.4.4. 独自の IdentityStore の記述

独自のアイデンティティストア実装を記述することで、そのままでは Seam でサポートされないセキュリティプロバイダに対し認証やアイデンティティ管理の操作を行うことができます。 必要となるのは **org.jboss.seam.security.management.IdentityStore** インターフェースを実装するクラスひとつのみです。

実装する必要があるメソッドの詳細に関する **IdentityStore** は JavaDoc を参照してください。

15.4.5. アイデンティティ管理による認証

Seam アプリケーションでアイデンティティ管理の機能を使用する場合には、認証コンポーネント (前述の認証の項を参照) を与えて認証を有効にする必要はありません。 単純に **components.xml** の

identity 設定から **authenticator-method** を省略するだけで、 特別な設定をすることなく **SeamLoginModule** は **IdentityManager** を使用してアプリケーションのユーザー認証を行います。

15.4.6. IdentityManager の使用

IdentityManager へのアクセス方法は 2 通りあります。 Seam コンポーネントにインジェクトする方法は次のとおりです。

```
@In IdentityManager identityManager;
```

その静的な **instance()** メソッド経由によるアクセス方法は以下のとおりです。

```
IdentityManager identityManager = IdentityManager.instance();
```

以下のテーブルに **IdentityManager** のAPIのメソッドを示します。

表15.4 アイデンティティ管理 API

メソッド	戻り値	詳細
createUser(String name, String password)	**boolean**	指定された名前とパスワードで新規ユーザーのアカウントを作成します。成功すれば **true** を、しなければ**false** を返します。
deleteUser(String name)	**boolean**	指定された名前のユーザーアカウントを削除します。 成功すれば **true** を、しなければ **false** を返します。
createRole(String role)	**boolean**	指定された名前の新しいロールを作成します。 成功すれば **true** を、しなければ **false** を返します。
deleteRole(String name)	**boolean**	指定された名前のロールを削除します。成功すれば **true** を、しなければ**false** を返します。
enableUser(String name)	**boolean**	指定された名前のユーザーアカウントを有効にします。有効でないアカウントは認証できません。 成功すれば **true** を、しなければ **false** を返します。
disableUser(String name)	**boolean**	指定された名前のユーザーアカウントを無効にします。成功すれば **true** を、しなければ **false** を返します。
changePassword(String name, String password)	**boolean**	指定された名前のユーザーアカウントのパスワードを変更します。 成功すれば **true** を、しなければ**false** を返します。

メソッド	戻り値	詳細
isUserEnabled(String name)	**boolean**	指定されたユーザーアカウントが有効であれば **true** を、これ以外は **false** を返します。
grantRole(String name, String role)	**boolean**	指定ロールを指定したユーザーまたはロールに付与します。 ロールを付与するには、 そのロールが既に存在している必要があります。 ロールの付与が成功した場合には **true** を返し、 そのロールがユーザーに既に付与されていた場合には **false** を返します。
revokeRole(String name, String role)	**boolean**	特定のユーザーまたはロールから指定したロールを取り消します。ユーザーが当該のロールのメンバーであり、かつ取り消しが成功した場合には **true** を返し、ユーザーが当該ロールのメンバーでなければ **false** を返します。
userExists(String name)	**boolean**	指定ユーザーが存在すれば **true** を、 存在していなければ **false** を返します。
listUsers()	**List**	英数字順にソートされたすべてのユーザー名の一覧を返します。
listUsers(String filter)	**List**	指定されたフィルタパラメータでフィルタしたユーザー名のリストを英数字順にソートして返します
listRoles()	**List**	すべてのロール名の一覧を返します。
getGrantedRoles(String name)	**List**	指定されたユーザー名に明示的に付与された全ロールの一覧を返します。
getImpliedRoles(String name)	**List**	指定されたユーザー名に対して暗示的に付与されている全ロールの一覧を返します。 暗示的に付与されているロールとは、ユーザーに直接付与されたロールではなく、 ユーザーがメンバーとなるロールに対して付与されたロールなどです。 例えば、 **admin** ロールが **user** ロールのメンバーであり、 ユーザーが **admin** ロールのメンバーであった場合、 このユーザーの暗示的なロールは **admin** ロールと **user** ロールの両方になります。

メソッド	戻り値	詳細
authenticate(String name, String password)	**boolean**	設定された Identity Store を使ってユーザー名とパスワードを認証します。 認証が成功すれば **true** を、失敗すれば **false** を返します。認証が成功しても、このメソッドの戻り値以外は何も変化しませんし、**Identity** コンポーネントの状態も変化しません。適切に Seam にログインするには **Identity.login()** を使用する必要があります。
addRoleToGroup(String role, String group)	**boolean**	特定のロールを指定したグループのメンバーとして追加します。成功すれば true を返します。
removeRoleFromGroup(String role, String group)	**boolean**	指定されたロールを指定されたグループから削除します。成功すれば true を返します。
listRoles()	**List**	すべてのロール名を一覧表示します。

ユーザー呼び出しにはアイデンティティ管理 API でメソッドを呼び出すための適切な承認が必要となります。 次の表では **IdentityManager** 内の各メソッドに対するパーミッション要件を示します。以下に記載されるパーミッションターゲットはリテラル文字列値です。

表15.5 アイデンティティ管理のセキュリティパーミッション

メソッド	パーミッションターゲット	パーミッションのアクション
createUser()	**seam.user**	**create**
deleteUser()	**seam.user**	**delete**
createRole()	**seam.role**	**create**
deleteRole()	**seam.role**	**delete**
enableUser()	**seam.user**	**update**
disableUser()	**seam.user**	**update**
changePassword()	**seam.user**	**update**
isUserEnabled()	**seam.user**	**read**
grantRole()	**seam.user**	**update**

メソッド	パーミッションターゲット	パーミッションのアクション
revokeRole()	**seam.user**	**update**
userExists()	**seam.user**	**read**
listUsers()	**seam.user**	**read**
listRoles()	**seam.role**	**read**
addRoleToGroup()	**seam.role**	**update**
removeRoleFromGroup()	**seam.role**	**update**

以下のコードのリストでは **admin** ロールの全メンバーにアイデンティティ管理関連の全メソッドへのアクセス権を許可する一連のセキュリティルールの例を示しています。

```
rule ManageUsers
  no-loop
  activation-group "permissions"
when
  check: PermissionCheck(name == "seam.user", granted == false)
  Role(name == "admin")
then
  check.grant();
end

rule ManageRoles
  no-loop
  activation-group "permissions"
when
  check: PermissionCheck(name == "seam.role", granted == false)
  Role(name == "admin")
then
  check.grant();
end
```

15.5. エラーメッセージ

セキュリティ API はセキュリティ関連の各種イベントに対してデフォルト Faces メッセージをいくつか生成します。 以下にメッセージを上書きする場合に **message.properties** リソースファイルで指定するメッセージキーを一覧表示します。 特定のメッセージを隠す場合は、キー (値が空白となる) をリソースファイルに追加します。

表15.6 セキュリティメッセージキー

メッセージキー	詳細
org.jboss.seam.loginSuccessful	このメッセージは、セキュリティ API を通してユーザーのログインが成功した場合に生成されます。
org.jboss.seam.loginFailed	このメッセージは、正しくないユーザー名またはパスワードを入力したか、何らかの認証のエラーによりユーザーがログインに失敗したときに生成されます。
org.jboss.seam.NotLoggedIn	このメッセージは、セキュリティチェックが必要な操作の実行またはページへのアクセスをユーザーが試行し、現在認証されていない場合に生成されます。
org.jboss.seam.AlreadyLoggedIn	このメッセージは、既に認証されたユーザーが再度ログインを試みた時に生成されます。

15.6. 承認

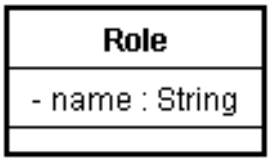
本項ではコンポーネント、 コンポーネントのメソッド、 ページへのアクセスの安全化を図るため Seam Security API で提供される各種の承認メカニズムについて説明しています。 いずれかの高度な機能 (ルールベースのパーミッションなど) を使用したい場合は **components.xml** ファイルを設定する必要があるかもしれません。 前述の「設定」の項を参照してください。

15.6.1. 核となる概念

Seam Security はユーザーはロールまたはパーミッションあるいはその両方を与えられるという原理で動作します。 これによりユーザーは必要なセキュリティ権限を持たないユーザーには許容されない操作を行うことができます。Seam Security API 提供のそれぞれの承認メカニズムは、ロールとパーミッションに関するこの中核となる概念に基づき構築され、 拡張可能なフレームワークでアプリケーションリソースの安全化を図る複数の方法を提供しています。

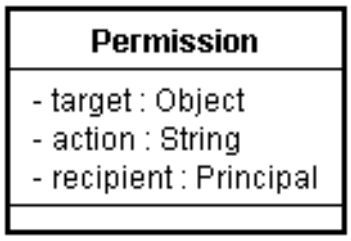
15.6.1.1. ロールとは

ロールとは、 アプリケーション内で 1 つ以上の特定の操作を行う特権を付与されてる可能性があるユーザーのタイプです。 構成はシンプルで、 ユーザーまたは他のロールに適用される名前 (「admin」、「user」、「customer」など) により構成されています。 論理的なユーザーグループの作成に使用され、 任意のアプリケーションの権限を容易に割り当てることが可能です。



15.6.1.2. パーミッションとは？

パーミッションとは1つの特定の操作を実行するための特権(時として1回限り)を言います。パーミッションのみで動作するアプリケーションを構築することも可能ですが、グループに対して特権を付与する場合はロールの方が便利です。パーミッションはロールに比べると構造が若干複雑になり、対象、操作、受信者の3種類の「側面」で構成されます。パーミッションの対象は特定の受信者(ユーザー)によって実行が許可される特定の操作に対するオブジェクト(あるいは、任意の名前またはクラス)です。例として、ユーザー「Bob」に顧客オブジェクトを削除するパーミッションがあるとします。この場合、パーミッションの対象は「顧客」、パーミッションの操作は「削除」、そして受益者は「Bob」ということになります。



本ガイド内では、通常パーミッションを **target:action** という形式で受信者を省略して表示されています。実際には受信者は常に必要です。

15.6.2. コンポーネントをセキュアにする

最も簡単な形式の承認、コンポーネントのセキュリティから見ていくことにします。**@Restrict** アノテーションから始めましょう。

注記

@Restrict アノテーションはセキュリティコンポーネントに対してパワフルで柔軟なメソッドですが、EL式に対応できません。したがって、コンパイル時の安全性のためタイプセーフと同等の方法を使用することをお勧めします(本章後半に記載)。

15.6.2.1. @Restrict アノテーション

@Restrict アノテーションにより Seam のコンポーネントはクラスあるいはメソッドいずれかのレベルでの安全化を図ることができます。メソッドとその宣言クラスの両方に **@Restrict** アノテーションを付与すると、メソッドの制限が優先されるためクラスの制限は適用されません。メソッド呼び出しがセキュリティチェックに失敗した場合には、**Identity.checkRestriction()** のコントラクトに応じて例外が送出されます(Inline Restriction については本項の後半を参照)。コンポーネントクラス自体での **@Restrict** は、そのメソッドそれぞれに **@Restrict** を追加したのと同じことになります。

空の **@Restrict** は **component:methodName** のパーミッションチェックを暗示します。次のコンポーネントメソッドの例を見てみましょう。

```
@Name("account")
public class AccountAction {
  @Restrict
  public void delete() {
    ...
  }
}
```

この例では **account:delete** は、**delete()** メソッドを呼び出すために必要な暗黙権限です。これは **@Restrict("#{s:hasPermission('account','delete')}")** と記述するのと同等です。 別の例についても見てみます。

```
@Restrict @Name("account")
public class AccountAction {
  public void insert() {
    ...
  }
  @Restrict("#{s:hasRole('admin')}")
  public void delete() {
    ...
  }
}
```

ここでは、 コンポーネントクラス自体に **@Restrict** アノテーションが付与されています。 つまり **@Restrict** アノテーションを上書きしないメソッドはすべて暗黙のパーミッションチェックが必要になります。 この例の場合、 **insert()** メソッドには **account:insert** のパーミッションが必要となり、 **delete()** メソッドにはユーザーが **admin** ロールのメンバーであることが必要となります。

先に進む前に、 上の例で見た **#{s:hasRole()}** 式について見てみましょう。 **s:hasRole** も **s:hasPermission** も EL 式であり、 **Identity** クラスの同様の名前のメソッドに委任します。 こうした関数は EL 式内でセキュリティ API 全体に渡り使用することができます。

EL 式とすることで、 **@Restrict** アノテーションの値は Seam コンテキスト中のいずれのオブジェクトでも参照することができるようになります。 特定のオブジェクトのインスタンスのパーミッションをチェックする場合に非常に有効な方法です。 下の例を見てみましょう。

```
@Name("account")
public class AccountAction {
  @In Account selectedAccount;
  @Restrict("#{s:hasPermission(selectedAccount,'modify')}")
  public void modify() {
    selectedAccount.modify();
  }
}
```

この例では、 **hasPermission()** 関数呼び出しは **selectedAccout** を参照しています。 この変数の値は Seam コンテキスト中で検索され、 **Identity** 内の **hasPermission()** メソッドに渡されます。 これにより特定の **Account** オブジェクトの変更に要されるパーミッションをユーザーが持っているかどうかを判断します。

15.6.2.2. インラインによる制約

時として、 **@Restrict** アノテーションを使わずにコードでセキュリティチェックを行う必要がある場合があります。 この様な場合には、 以下のように **Identity.checkRestriction()** を使ってセキュリティ式を評価することで行います。

```
public void deleteCustomer() {
  Identity.instance().checkRestriction("#
{s:hasPermission(selectedCustomer,
                                                    'delete')}");
}
```

指定した式が **true** に評価しない場合は 2 種類の例外のうちいずれかが発生します。 ユーザーがログインしていなかった場合は **NotLoggedInException** が送出されます。 ユーザーがログインしている場合は **AuthorizationException** が送出されます。

また、 下のように Java コードから直接 **hasRole()** や **hasPermission()** のメソッドを呼び出すこともできます。

```
if (!Identity.instance().hasRole("admin"))
  throw new AuthorizationException("Must be admin to perform this
action");

if (!Identity.instance().hasPermission("customer", "create"))
     throw new AuthorizationException("You may not create new customers");
```

15.6.3. ユーザーインターフェースのセキュリティ

適切に設計されたインターフェースはユーザーに使用許可がないオプションはそのユーザーには表示しません。 Seam Security はユーザーの権限に応じて個別のコントロールやページセクションの条件的な表示が可能で、 コンポーネントのセキュリティに使用する式と同じ EL 式を使用します。

このセクションではインターフェースのセキュリティ例についていくつか見ていきます。 まず、 ユーザーがまだログインしていない場合にのみ表示させたいログインフォームがあるとします。 **identity.isLoggedIn()** プロパティで次のように記述することができます。

```
<h:form class="loginForm" rendered="#{not identity.loggedIn}">
```

ユーザーがログインしていなければこのログインフォームが表示されます (実にシンプルですね)。 次に、 このページにメニューがあり、 **manager** ロールを持っているユーザーだけがアクセス可能な操作をいくつか持たせたいとします。このような場合の 1 つの方法として次のように記述することができます。

```
<h:outputLink action="#{reports.listManagerReports}"
   rendered="#{s:hasRole('manager')}"> Manager Reports
</h:outputLink>
```

これもシンプルで、 ユーザーが **manager** ロールのメンバーでなければ outputLink は表示されません。 **rendered** 属性は一般に制御そのものに使われたり、 前後にある **<s:div>** や **<s:span>** 制御で使われます。

次にもう少し複雑な条件的な表示の例を見てみましょう。 ページに **h:dataTable** コントロールがあり、 特定の権限を持つユーザーだけにその記録に操作リンクを表示させたいとします。 **s:hasPermission** EL 関数によりオブジェクトパラメータを使ってユーザーがそのオブジェクトに対して必要なパーミッションを持っているかどうかを判断することができます。 安全なリンクを持つ dataTable は次のようになります。

```
<h:dataTable value="#{clients}" var="cl">
  <h:column>
    <f:facet name="header">Name</f:facet>
    #{cl.name}
  </h:column>
  <h:column>
    <f:facet name="header">City</f:facet>
    #{cl.city}
  </h:column>
```

```
  <h:column>
    <f:facet name="header">Action</f:facet>
    <s:link value="Modify Client" action="#{clientAction.modify}"
            rendered="#{s:hasPermission(cl,'modify')}"/>
    <s:link value="Delete Client" action="#{clientAction.delete}"
            rendered="#{s:hasPermission(cl,'delete')}"/>
  </h:column>
</h:dataTable>
```

15.6.4. ページ単位のセキュリティ

ページセキュリティを使用する場合には **pages.xml** ファイルが必要になります。 ページセキュリティの設定は簡単です。 保護したい **page** のエレメントに **<restrict/>** エレメントを含ませるだけです。 **restrict** エレメントで明示的な制限が指定されていない場合、 Faces 以外 (GET) の要求によるアクセスには暗黙的な **/viewId.xhtml:render** パーミッションが必要となり、 また JSF ポストバック (フォームのサブミッション) がそのページから発生する場合には **/viewId.xhtml:restore** パーミッションが必要となります。 これ以外は、 指定した制限は標準のセキュリティ式で評価が行われます。 以下にいくつかの例を示します。

```
<page view-id="/settings.xhtml">
  <restrict/>
</page>
```

このページでは Faces 以外の要求には **/settings.xhtml:render** の暗黙のパーミッションを必要とし、 Faces 要求には **/settings.xhtml:restore** の暗黙のパーミッションが必要となります。

```
<page view-id="/reports.xhtml">
  <restrict>#{s:hasRole('admin')}</restrict>
</page>
```

このページに対する Faces と Faces 以外 の要求はいずれもユーザーが **admin** ロールのメンバーであることを必要とします。

15.6.5. エンティティをセキュアにする

Seam Security ではエンティティに対して特定の操作 (読み込む、 挿入、 更新、 削除) にセキュリティ制約を適用することもできます。

エンティティクラスのすべてのアクションをセキュアにするためには、下のようにクラスに **@Restrict** アノテーションを付与します。

```
@Entity
@Name("customer")
@Restrict
public class Customer {
  ...
}
```

@Restrict アノテーションに式が指定されていない場合、 デフォルトの操作は **entity:action** のパーミッションチェックとなります。 パーミッションの対象はエンティティのインスタンスで、 **action** は **read**、 **insert**、 **update**、 **delete** のいずれかになります。

また、以下のとおり関連するエンティティのライフサイクルメソッドに **@Restrict** アノテーションを付与して特定の操作を制約することもできます。

- **@PostLoad** – エンティティのインスタンスがデータベースからロードされた後に呼び出されます。 このメソッドは **read** パーミッションの設定に使用します。
- **@PrePersist** – エンティティの新しいインスタンスが挿入される前に呼び出されます。 このメソッドは **insert** パーミッションの設定に使用します。
- **@PreUpdate** – エンティティが更新される前に呼び出されます。 このメソッドは **update** パーミッションの設定に使用します。
- **@PreRemove** – エンティティが削除される前に呼び出されます。 このメソッドは **delete** パーミッションの設定に使用します。

insert の動作に対してセキュリティチェックを行うようエンティティを設定する方法を以下の例に示します。 メソッドは何も操作を行う必要はなく、重要なのはアノテーションが正しく付与されていることです。

```
@PrePersist
@Restrict
public void prePersist() {}
```

注記

/META-INF/orm.xml にコールバックメソッドを指定することもできます。

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
          http://java.sun.com/xml/ns/persistence/orm
          http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
          version="1.0">

  <entity class="Customer">
    <pre-persist method-name="prePersist" />
  </entity>

</entity-mappings>
```

この場合も **Customer** の **prePersist()** メソッドに **@Restrict** アノテーションを付与する必要があります。

次の設定は **Seamspace** サンプルをベースとしています。 認証済みユーザーが新しい **MemberBlog** 記録を挿入できるパーミッションを持っているかどうかを確認します。 チェックが行われるエンティティはワーキングメモリに自動的に挿入されます **(**この例の場合は **MemberBlog)**。

```
rule InsertMemberBlog
  no-loop
  activation-group "permissions"
when
```

```
  principal: Principal()
  memberBlog: MemberBlog(member : member ->

(member.getUsername().equals(principal.getName())))
  check: PermissionCheck(target == memberBlog,
                         action == "insert", granted == false)
then
  check.grant();
end;
```

このルールは、 現在認証済みのユーザー名 (**Principal** ファクトで示される) がブログのエントリを作成したメンバーの名前と一致すると **memberBlog:insert** パーミッションを付与します。 **principal: Principal()** の構造は変数のバインディングです。 認証中にワーキングメモリに配置された **Principal** オブジェクトのインスタンスをバインドし、 それを**principal** という変数に割り当てます。 変数のバインディングにより、 次の行のような他の場所で変数の参照が可能になり、 メンバーの名前を **Principal** の名前と比較します。 詳細は JBoss Rules のドキュメントを参照してください。

最後に、使用する JPA プロバイダを Seam Security と統合するためにリスナークラスをインストールします。

15.6.5.1. JPA でのエンティティセキュリティ

EJB3 エンティティ Bean のセキュリティチェックは **EntityListener** を使って行われます。 次の **META-INF/orm.xml** ファイルでこのリスナーをインストールします。

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="
                   http://java.sun.com/xml/ns/persistence/orm
                   http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
                 version="1.0">

  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener
                class="org.jboss.seam.security.EntitySecurityListener"/>
      </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>

</entity-mappings>
```

15.6.5.2. 管理 Hibernate セッションによるエンティティのセキュリティ

Seam で設定された Hibernate **SessionFactory** を使用し、アノテーションまたは **orm.xml** を使用している場合には、エンティティセキュリティ機能を使用するための特別な変更は必要はありません。

15.6.6. タイプセーフなパーミッションのアノテーション

Seam では **@Restrict** に対する代替のアノテーションをいくつか提供しています。 異なる方法で任意の EL 式に対応し、 コンパイルタイムの安全性が向上します。

Seam には標準の CRUD ベースのパーミッション用にアノテーション一式が同梱されています。 次のアノテーションは **org.jboss.seam.annotations.security** パッケージで提供されています。

- @Insert
- @Read
- @Update
- @Delete

これらのアノテーションを使用するためには、セキュリティチェックを行いたいメソッドやパラメータに配置します。メソッドに置く場合は、パーミッションがチェックされる対象クラスを指定します。 以下の例を見てください。

```
@Insert(Customer.class)
public void createCustomer() { ... }
```

この例ではユーザーに対してパーミッションチェックが行われ、 新しい **Customer** オブジェクトを作成するパーミッションを有していることを確認します。 パーミッションチェックの対象は **Customer.class** (実際の **java.lang.Class** インスタンス自体) となり、 操作はアノテーション名の小文字表記です。この例では **insert** です。

同様に以下のようにしてコンポーネントメソッドのパラメータにアノテーションを付与することができます。 これを行う場合には、 パラメータの値自体がパーミッションチェックの対象となるためパーミッションの対象を指定する必要はありません。

```
public void updateCustomer(@Update Customer customer) {
  ...
}
```

独自のセキュリティアノテーションを作成するためには、 **@PermissionCheck** アノテーションを付与するだけです。 例を示します。

```
@Target({METHOD, PARAMETER})
@Documented
@Retention(RUNTIME)
@Inherited
@PermissionCheck
public @interface Promote {
  Class value() default void.class;
}
```

デフォルトのパーミッション操作の名前 (アノテーション名の小文字版) を別の値で上書きしたい場合は、 **@PermissionCheck** アノテーション内にその値を指定することができます。

```
@PermissionCheck("upgrade")
```

15.6.7. タイプセーフなロールのアノテーション

タイプセーフなパーミッションのアノテーションをサポートするのに加えて、 Seam セキュリティはタイプセーフなロールのアノテーションを提供しています。 現在認証済みのユーザーのロールのメンバーシップに基づいてコンポーネントメソッドへのアクセスを制限することができます。 Seam はこのようなアノテーションで特に設定を必要としないものをひとつ提供しています

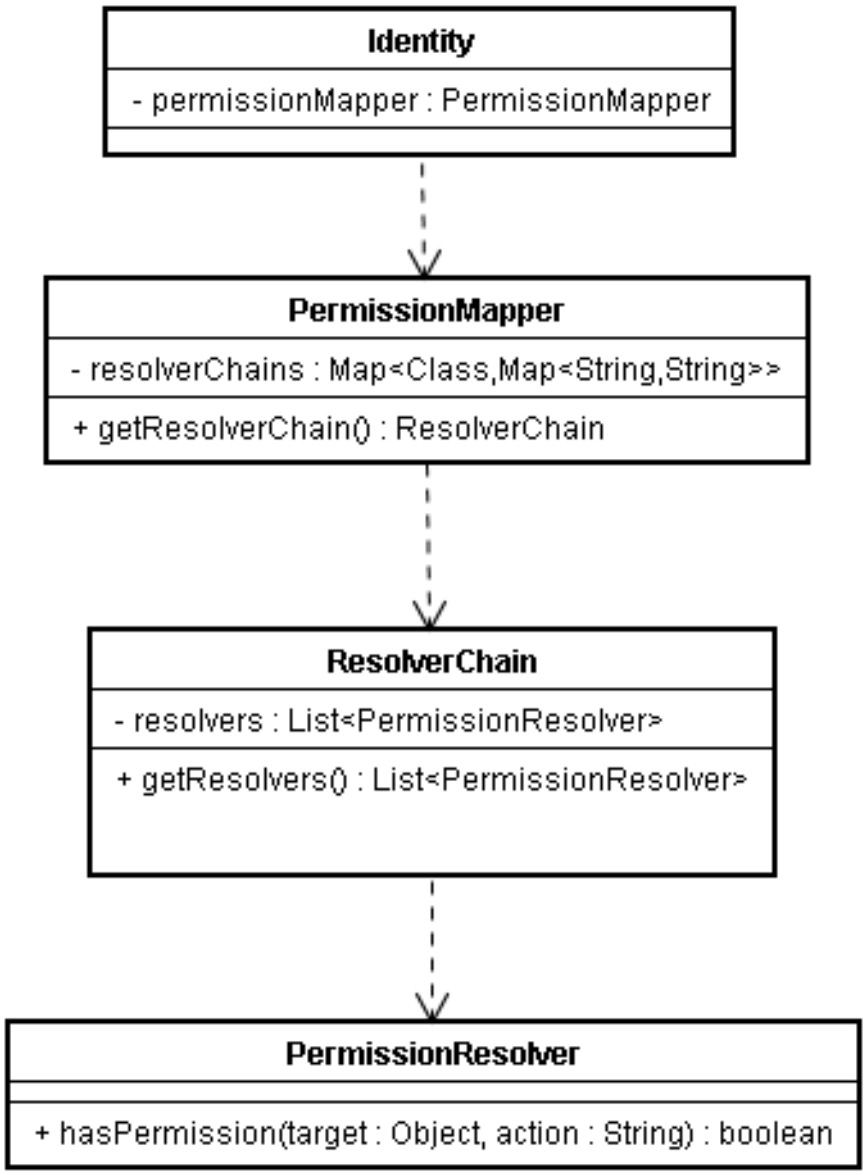
(**org.jboss.seam.annotations.security.Admin**)。 **admin** ロール (アプリケーションによってこのロールがサポートされている限り) に属しているユーザーに対して特定のメソッドのアクセスを制限します。 独自のロールアノテーションを作成するためは、 以下の例のように **org.jboss.seam.annotations.security.RoleCheck** メタアノテーションを付与します。

```
@Target({METHOD})
@Documented
@Retention(RUNTIME)
@Inherited
@RoleCheck
public @interface User { }
```

続けて **@User** アノテーションを付与されるメソッドはすべて自動的にインターセプトされます。 ユーザーは該当するロール名のメンバーシップについてチェックされます (アノテーション名の小文字部分、 この場合は **user**)。

15.6.8. パーミッションの承認モデル

Seam Security はアプリケーションに対するパーミッションの決定に対して拡張可能なフレームワークを提供します。下記のクラスダイアグラム図には、パーミッションフレームワークの主要コンポーネントの概要について示しています。



関連するクラスについての詳細を以下のセクションに示します。

15.6.8.1. PermissionResolver

個々のオブジェクトのパーミッションを解決するためのメソッドを提供するインターフェースです。 Seam は以下の組み込み **PermissionResolver** の実装を提供しています。 詳細については本章の後半に記載します。

- **RuleBasedPermissionResolver** – Drools を使ってルールベースのパーミッションチェックを解決します。
- **PersistentPermissionResolver** – リレーショナルデータベースなど永続的なストアにオブジェクトのパーミッションを保存します。

15.6.8.1.1. 独自の PermissionResolver の記述

独自のパーミッションリゾルバを実装するのは簡単です。 以下の表のように、 **PermissionResolver** インターフェースは実装しなければならない 2 種類のメソッドを定義します。 **PermissionResolver** を Seam プロジェクトにデプロイする場合、 デプロイ時に自動的にスキャンされてからデフォルトの **ResolverChain** で登録されます。

表15.7 PermissionResolver インターフェース

戻り値のタイプ	メソッド	詳細
boolean	**hasPermission(Object target, String action)**	このメソッドは現在認証済みのユーザー (**Identity.getPrincipal()** への呼び出しで取得) が **target** と **action** のパラメータで指定されるパーミッションを持っているかどうかを解決します。 ユーザーが指定パーミッションを持っている場合は **true** を、 持っていない場合は **false** を返します。
void	**filterSetByAction(Set<Object> targets, String action)**	このメソッドは、 同じ **action** パラメータ値を持つ **hasPermission()** メソッドに渡されると **true** を返す指定セットからオブジェクトを削除します。

注記

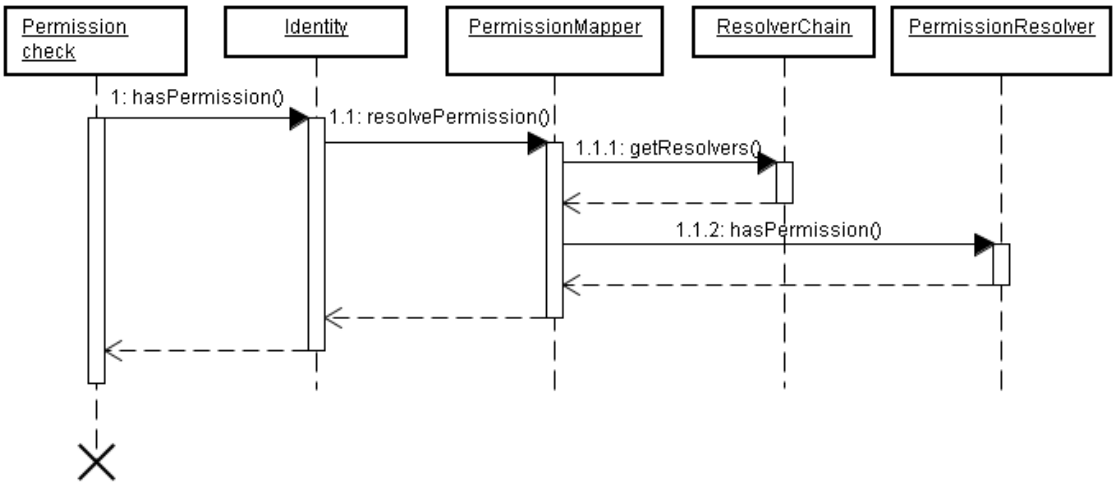
ユーザーのセッションにキャッシュされるため、**PermissionResolver** の実装はいくつかの制約に順守していなければなりません。 まず、 セッションスコープより粒度の細かい状態は含むことはできず、 コンポーネント自体がアプリケーションスコープまたはセッションスコープのいずれかになるはずです。 次に複数のスレッドから同時にアクセスされる可能性があるため、依存インジェクションを使用してはいけません。 最適なパフォーマンスを得るには **@BypassInterceptors** アノテーションを付与して Seam のインターセプタスタックをすべて一緒に迂回することをお勧めします。

15.6.8.2. ResolverChain

ResolverChain には順序つき一覧の **PermissionResolver** が含まれ、 特定のオブジェクトクラスやパーミッション対象に対してオブジェクトのパーミッションを解決します。

デフォルトの **ResolverChain** にはアプリケーションのデプロイメント時に見つかったすべてのパーミッションリゾルバが含まれます。 デフォルトの **ResolverChain** が作成されると **org.jboss.seam.security.defaultResolverChainCreated** イベントが発生します (また、 **ResolverChain** インスタンスがイベントパラメータとして渡されます)。 これによりデプロイメント時には見つからなかったリゾルバの追加、 またはチェーン内にあるリゾルバの並び替えや削除ができるようになります。

以下の順序図はパーミッションチェック時のパーミッションフレームワーク内のコンポーネント間の相互作用を示しています。 パーミッションチェックはセキュリティインターセプタ、 **s:hasPermission** EL 式などのソースから、または **Identity.checkPermission** への API 呼び出しにより発生することができます。



- 1. パーミッションチェックが開始され (コードまたは EL 式のいずれかにより)、 これにより **Identity.hasPermission()** への呼び出しが行われます。
- 1.1. **Identity** は **PermissionMapper.resolvePermission()** を呼び出し、解決されるパーミッションを渡します。
- 1.1.1. **PermissionMapper** はクラスによりキー付けされた**ResolverChain** インスタンスの **Map** を維持します。 このマップを使ってパーミッションの対象オブジェクトに正しい **ResolverChain** を検索します。 適切な **ResolverChain** が見つかると、**ResolverChain.getResolvers()** を呼び出してそれが含む **PermissionResolver** の一覧を読み出します。
- 1.1.2. **ResolverChain** 内の各 **PermissionResolver** に対して、 **PermissionMapper** はその **hasPermission()** メソッドを呼び出し、チェックすべきパーミッションインスタンスを渡します。 **PermissionResolver** が **true** を返す場合はパーミッションチェックが成功しているため、 **PermissionMapper** も **Identity** に **true** を返します。 いずれの **PermissionResolver** も **true** を返さなければパーミッションチェックは失敗したことになります。

15.6.9. RuleBasedPermissionResolver

これは Seam 提供の組み込みパーミッションリゾルバーの 1 つです。Drools (JBoss Rules) セキュリティルール一式に基づいてパーミッションの評価を行います。ルールエンジンの利点は、ユーザーパーミッションの評価に使用されるビジネスロジックを 1 か所にまとめることができること、また Drools

のアルゴリズムは複数の条件を伴う複雑なルールを多数評価する場合に非常にスピードの面で効率的であることです。

15.6.9.1. 要件

Seam Security 提供のルールベースのパーミッション機能を使用したい場合は、Drools では下記の **JAR** ファイルの配信がプロジェクトで必要となります。

- drools-api.jar
- drools-compiler.jar
- drools-core.jar
- janino.jar
- antlr-runtime.jar
- mvel2.jar

15.6.9.2. 設定

RuleBasedPermissionResolver の設定には、**components.xml** で Drools のルールベースがまず設定されていることが必要です。 以下の例のように、デフォルトではルールベースに **securityRules** という名前が付けられていることが期待されます。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:security="http://jboss.com/products/seam/security"
            xmlns:drools="http://jboss.com/products/seam/drools"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/core
               http://jboss.com/products/seam/core-2.2.xsd
               http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd
               http://jboss.com/products/seam/drools
               http://jboss.com/products/seam/drools-2.2.xsd
               http://jboss.com/products/seam/security
               http://jboss.com/products/seam/security-2.2.xsd">

  <drools:rule-base name="securityRules">
    <drools:rule-files>
      <value>/META-INF/security.drl</value>
    </drools:rule-files>
  </drools:rule-base>

</components>
```

デフォルトのルールベースの名前は **RuleBasedPermissionResolver** の **security-rules** プロパティを指定することで上書きすることができます。

```
<security:rule-based-permission-resolver
          security-rules="#{prodSecurityRules}"/>
```

RuleBase コンポーネントを設定したら、 次にセキュリティルールを記述します。

15.6.9.3. セキュリティルールの記述

セキュリティルールを記述する最初のステップは、アプリケーションの jar ファイルの **/META-INF** ディレクトリ内に新しいルールファイルを作成することです。 このファイルは **security.drl** のような名前が付けられるはずですが、 **components.xml** で対応するよう設定されていればどのような名前でも構いません。

ルールのファイルを記述する場合は Drools のドキュメントをお勧めします。 シンプルなルールファイルの例を示します。

```
package MyApplicationPermissions;

import org.jboss.seam.security.permission.PermissionCheck;
import org.jboss.seam.security.Role;

rule CanUserDeleteCustomers
when
  c: PermissionCheck(target == "customer", action == "delete")
  Role(name == "admin")
then
  c.grant();
end
```

最初にあるのはパッケージ宣言です。Drools 内の 1 パッケージはルールの集合です。 パッケージ名はルールベースの範囲外のものには関係しないため、 どのような名前を付けても構いません。

次に、 **PermissionCheck** クラスと **Role** クラスに関するいくつかのインポート文があります。 これらのインポート文は使用するルールがこれらのクラスを参照することをルールエンジンに伝えます。

最後にルールコードがあります。 1 パッケージ内の各ルールは一意の名前を持っているはずで、 通常はそのルールの目的を表しています。 この例の場合、 **CanUserDeleteCustomers** がルールの名前で、顧客記録の削除をユーザーが許可されているかどうかのチェックに使用されます。

ルール定義のボディにははっきり異なる 2 つのセクションがあります。 ルールには左部分 (LHS) と右部分 (RHS) があります。 LHS はルールの条件部分、 つまり、 ルールが実行されるために満たさなければならない条件の一覧です。 LHS は **when** セクションで表されます。 RHS はルールの動作セクションまたは結果になり、 LHS にある条件がすべて満たされた場合にのみ実行されます。 RHS は **then** セクションで表されます。 ルールの最後は **end** 行で表されます。

サンプルの LHS には 2 つの条件が記載されています。 最初の条件は次のとおりです。

```
c: PermissionCheck(target == "customer", action == "delete")
```

簡単に言うと、満たされるべきこの条件は、 ワーキングメモリ内に **customer** と同じ **target** プロパティと **delete** と同じ **action** プロパティを持つ **PermissionCheck** オブジェクトがなければならないことを示しています。

ワーキングメモリは、Drools の技術用語で **ステートフルセッション** としても知られています。これは、ルールエンジンがパーミッションチェックに関して決定をするために必要となるコンテキスト情報を含むセッションスコープのオブジェクトです。 **hasPermission()** メソッドが呼び出されるたび、仮の **PermissionCheck** オブジェクトまたは **ファクト** がワーキングメモリに挿入されます。 この **PermissionCheck** はチェックされるパーミッションに正確に対応するため、 **hasPermission("account", "create")** を呼び出すと 「account」と同じ **target** と「create」

と同じ **action** を持つ **PermissionCheck** オブジェクトがパーミッションチェックの間にワーキングメモリに挿入されます。

PermissionCheck ファクトの他に、 認証済みユーザーがメンバーとなるそれぞれのロールに **org.jboss.seam.security.Role** ファクトがあります。 これらの **Role** ファクトは、各パーミッションチェックの開始時にユーザーの認証済みロールと同期されます。結果として、 パーミッションチェックの間にワーキングメモリに挿入された **Role** オブジェクトはすべて、 認証済みユーザーが実際にそのロールのメンバーでない限り、 次回のパーミッションチェックが起こる前に削除されます。 ワーキングメモリには認証プロセスの結果として作成される **java.security.Principal** オブジェクトも含まれます。

RuleBasedPermissionResolver.instance().getSecurityContext().insert() を呼び出すと追加でワーキングメモリに長期に生存するファクトを挿入することができ、 これがオブジェクトをパラメータとして渡します。 **Role** オブジェクトは例外です。 各パーミッションチェックの開始時に同期されるためです。

先の簡単な例に戻るためには、 LHS の 1 番目の行の先頭に **c:** が付けられています。 これは変数バインディングで、 条件に一致するオブジェクトを参照するために使用します (この例の場合は **PermissionCheck**)。 LHS の 2 行目には下の記述があります。

```
Role(name == "admin")
```

この条件はワーキングメモリ内に「admin」という **name** を持つ **Role** オブジェクトがなければならないことを記述しています。 このため、 **customer:delete** パーミッションをチェックしていてユーザーが **admin** ロールのメンバーである場合にこのルールが実行されます。

RHS ではルール実行の結果を示しています。

```
c.grant()
```

RHS は Java コードから構成されています。 この例の場合は **c** というオブジェクトの **grant()** メソッドを呼び出します。 これが **PermissionCheck** オブジェクトの変数バインディングです。 **PermissionCheck** オブジェクトの **name** プロパティと **action** プロパティの他に **granted** プロパティもあります。 これは最初は **false** に設定されています。 **PermissionCheck** の **grant()** を呼び出すと、**granted** プロパティは **true** に設定されます。 これはパーミッションのチェックが成功し、ユーザーはパーミッションチェックを求めた操作を実行する権限を持っているということになります。

15.6.9.4. 非文字列のパーミッションターゲット

ここまでは文字列リテラルのパーミッション対象のチェックについてのみ見てきました。 しかし、もっと複雑なパーミッション対象に対するセキュリティルールを記述することも可能です。 例えば、ユーザーによるブログへのコメント作成を許可するセキュリティルールを記述したいとします。 以下にひとつの方法を示します。 この例では、 パーミッションチェックの対象が **MemberBlog** インスタンスであり、 現在の認証済みユーザーが **user** ロールのメンバーであることが必要であると表現されています。

```
rule CanCreateBlogComment
  no-loop
  activation-group "permissions"
when
  blog: MemberBlog()
  check: PermissionCheck(target == blog, action == "create",
                         granted == false)
  Role(name == "user")
```

```
then
  check.grant();
end
```

15.6.9.5. ワイルドカードによるパーミッションチェック

以下のようにして、ルールの **PermissionCheck** の **action** 制約を省略することでワイルドカードのパーミッションチェックの実装が可能になります (特定のパーミッション対象にすべての操作を許可します。

```
rule CanDoAnythingToCustomersIfYouAreAnAdmin
when
  c: PermissionCheck(target == "customer")
  Role(name == "admin")
then
  c.grant();
end;
```

上記のルールにより、**admin** ロールを持つユーザーは、どの**customer** のパーミッションチェックに対しても**すべての**操作が可能となります。

15.6.10. PersistentPermissionResolver

もうひとつの Seam 提供の組み込みパーミッションリゾルバー、 **PersistentPermissionResolver** ではリレーショナルデータベースなどの永続的な保存場所からのパーミッションの読み込みを可能にします。 このパーミッションリゾルバでは ACL (Access Control List) スタイルのインスタンスベースのセキュリティを提供しており、 個別のユーザーやロールに対して特定のオブジェクトのパーミッションを割り当てることができます。 また、 同じようにして任意に名前が付けられた永続的なパーミッションの対象を割り当てることもできます (必ずしもオブジェクトまたはクラスベースである必要はありません)。

15.6.10.1. 設定

PersistentPermissionResolver を使用するには、 **components.xml** で有効な **PermissionStore** を設定する必要があります。 これが設定されていないと、 **PersistentPermissionResolver** はデフォルトのパーミッションストア 「JpaIdentityStore イベント」 を使用しようとします。 デフォルト以外のパーミッションストアを使用するには **permission-store** プロパティを以下のように設定します。

```
<security:persistent-permission-resolver
          permission-store="#{myCustomPermissionStore}"/>
```

15.6.10.2. パーミッションストア

PersistentPermissionResolver はパーミッションの永続化が行われるバックエンドの保存場所に接続するパーミッションストアを必要とします。 Seam は特に設定を必要としない **PermissionStore** 実装をひとつ **JpaPermissionStore** 提供しています。 これはリレーショナルデータベース内にパーミッションを保存します。 **PermissionStore** インターフェースを実装することで独自のパーミッションストアを作成することができます。 これは次のメソッドを定義します。

表15.8 PermissionStore のインターフェース

戻り値のタイプ	メソッド	詳細
List<Permission>	**listPermissions(Object target)**	このメソッドは指定した対象のオブジェクトに付与されているすべての権限を表す**Permission**オブジェクトの**List**を返します。
List<Permission>	**listPermissions(Object target, String action)**	このメソッドは指定した対象オブジェクトに付与された特定の操作を持つパーミッションをすべて表している**Permission**オブジェクトの**List**を返します。
List<Permission>	**listPermissions(Set<Object> targets, String action)**	このメソッドは指定した複数の対象オブジェクト一式に付与された特定の操作を持つパーミッションをすべて表している**Permission**オブジェクトの**List**を返します。
boolean	**grantPermission(Permission)**	このメソッドは指定された**Permission**オブジェクトをバックエンドの保存場所に永続化し、成功すると**true**を返します。
boolean	**grantPermissions(List<Permission> permissions)**	このメソッドは指定した**List**内に含まれる複数の**Permission**オブジェクトをすべて永続化し、成功すると**true**を返します。
boolean	**revokePermission(Permission permission)**	このメソッドは指定された**Permission**オブジェクトを永続ストレージから削除します。
boolean	**revokePermissions(List<Permission> permissions)**	このメソッドは指定されたリストにあるすべての**Permission**オブジェクトを永続ストレージから削除します。

戻り値のタイプ	メソッド	詳細
`List<String>`	`listAvailableActions(Object target)`	このメソッドは指定した対象オブジェクトのクラスに対して使用可能な全操作 (文字列型) の一覧を返すはずです。 パーミッション管理と併用して特定のクラスのパーミッションを付与するためのユーザーインターフェース構築に使用されます。

15.6.10.3. JpaPermissionStore

Seam 提供のデフォルトの **PermissionStore** 実装です。 パーミッションをリレーショナルデータベースに保存します。 使用する前に、 ユーザーとロールのパーミッションの保存用に 1 つまたは 2 つのエンティティクラスでの設定が必要です。 これらのエンティティクラスには特殊なセキュリティアノテーション一式を付与して、 格納されるパーミッションの各種の側面に対応するエンティティプロパティを設定しなければなりません。

ユーザーパーミッションとロールパーミッションの両方の保存に同一のエンティティ (データベーステーブルは 1 つ) を使用したい場合、 設定が必要となるのは **user-permission-class** プロパティのみです。 ユーザーパーミッションの保存場所とロールパーミッションの保存場所に別々のテーブルを使用する場合は **user-permission-class** プロパティも設定する必要があります。

例えば、ユーザーとロールのパーミッションを 1 つのエンティティクラスに保存するよう設定する場合は次のようになります。

```
<security:jpa-permission-store
          user-permission-class="com.acme.model.AccountPermission"/>
```

ユーザーパーミッションとロールパーミッションを別のエンティティクラスに保存する場合の設定は次のようになります。

```
<security:jpa-permission-store
          user-permission-class="com.acme.model.UserPermission"
          role-permission-class="com.acme.model.RolePermission"/>
```

15.6.10.3.1. パーミッションアノテーション

ユーザーパーミッションとロールパーミッションを含むエンティティクラスは **org.jboss.seam.annotations.security.permission** パッケージにある特別なアノテーション一式で設定を行う必要があります。 次の表でこれらのアノテーションを説明します。

表15.9 エンティティパーミッションアノテーション

アノテーション	ターゲット	詳細
@PermissionTarget	**FIELD,METHOD**	このアノテーションはパーミッションの対象を含んでいるエンティティのプロパティを識別します。 プロパティは **java.lang.String** タイプでなければなりません。
@PermissionAction	**FIELD,METHOD**	このアノテーションはパーミッションの操作を含んでいるエンティティプロパティを識別します。 このプロパティは **java.lang.String** タイプでなければなりません。
@PermissionUser	**FIELD,METHOD**	このアノテーションはパーミッションの受信者のユーザーを含んでいるエンティティプロパティを識別します。 このプロパティは **java.lang.String** タイプで、ユーザーのユーザー名を含んでいなければなりません。
@PermissionRole	**FIELD,METHOD**	このアノテーションはパーミッションの受信者のロールを含んでいるエンティティプロパティを識別します。 このプロパティは **java.lang.String** タイプで、ロール名を含んでいなければなりません。
@PermissionDiscriminator	**FIELD,METHOD**	このアノテーションはユーザーパーミッションとロールパーミッションの両方を同じエンティティまたはテーブルに保存する場合に使用します。 ユーザーパーミッションとロールパーミッションの区別に使用するエンティティのプロパティを識別します。 デフォルトでは、 列値が文字列リテラルの **user** を含む場合はその記録はユーザーパーミッションとして処理されます。 文字列リテラルの **role** を含む場合はロールパーミッションとして処理されます。 また、 アノテーション中で **userValue** と **roleValue** のプロパティを指定するとこれらのデフォルトを無効にすることもできます。 例えば、 **user** の代わりに **u** を、 **role** の代わりに **r** を使うには、 以下のようにアノテーションを記述します。 `@PermissionDiscriminator( userValue = "u", roleValue = "r")`

15.6.10.3.2. エンティティの例

この例ではユーザーパーミッションとロールパーミッションの両方を保存するひとつのエンティティクラスを示しています。 Seamspace からのサンプルです。

```
@Entity
public class AccountPermission implements Serializable {
  private Integer permissionId;
  private String recipient;
  private String target;
  private String action;
  private String discriminator;

  @Id @GeneratedValue
  public Integer getPermissionId() {
    return permissionId;
  }

  public void setPermissionId(Integer permissionId) {
    this.permissionId = permissionId;
  }

  @PermissionUser @PermissionRole
  public String getRecipient() {
    return recipient;
  }

  public void setRecipient(String recipient) {
    this.recipient = recipient;
  }

  @PermissionTarget
  public String getTarget() {
    return target;
  }

  public void setTarget(String target) {
    this.target = target;
  }

  @PermissionAction
  public String getAction() {
    return action;
  }

  public void setAction(String action) {
    this.action = action;
  }

  @PermissionDiscriminator
  public String getDiscriminator() {
    return discriminator;
  }

  public void setDiscriminator(String discriminator) {
    this.discriminator = discriminator;
  }
}
```

上の例に見るように、 **getDiscriminator()** メソッドに **@PermissionDiscriminator** アノテーションを付与して、 **JpaPermissionStore** にユーザーのパーミッションを表す記録とロールのパーミッションを表す記録を判定させています。 **getRecipient()** メソッドには **@PermissionUser** と **@PermissionRole** の両アノテーションが付与されています。 **discriminator** プロパティの値に応じて、 エンティティの **recipient** プロパティはユーザーの名前かロールの名前のいずれかを含むということになります。

15.6.10.3.3. クラス固有のパーミッションの設定

org.jboss.seam.annotation.security.permission パッケージに含まれるパーミッションは、対象クラスに対して許可可能な固有のパーミッション一式を設定するのに使用することができます。

表15.10 クラスパーミッションアノテーション

アノテーション	ターゲット	詳細
@Permissions	**TYPE**	コンテナのアノテーションです。 **@Permission** アノテーションの配列を含むことができます。
@Permission	**TYPE**	このアノテーションでは対象クラスに対して許可可能なパーミッション操作を1つ定義します。 その **action** プロパティを指定する必要があり、ビットマスク値でパーミッションの操作を永続化する場合にはオプションの **mask** プロパティも指定する必要があります (次項を参照)。

この例では上記のアノテーションを使っています。 SeamSpace サンプルでもご覧頂けます。

```
@Permissions({
  @Permission(action = "view"),
  @Permission(action = "comment")
})

@Entity
public class MemberImage implements Serializable {...}
```

この例では **view** と **comment** の 2 つのパーミッションアクションを **MemberImage** エンティティクラスに対して宣言する方法を示しています。

15.6.10.3.4. パーミッションマスク

デフォルトでは、 同じ対象オブジェクトと受信者に対する複数のパーミッションは単一のデータベース記録として、コンマで区切った許可される操作の一覧を含む **action** プロパティまたは列と共に永続化されます。 ビットマスク化した整数値を使ってパーミッションの操作一覧を保存することができます。 これにより大量のパーミッションの永続化に必要となる物理的な保存場所を低減します。

例えば、受信者「Bob」に特定の **MemberImage** (エンティティ Bean) インスタンスの **view** と **comment** の両方のパーミッションが付与された場合、 デフォルトではパーミッションエンティティの **action** プロパティは付与された 2 つのパーミッションアクションを表す「**view,comment**」を含みます。 または、 以下のように定義したビットマスク値を使用している場合は

```
@Permissions({
  @Permission(action = "view", mask = 1),
  @Permission(action = "comment", mask = 2)
})

@Entity
public class MemberImage implements Serializable {...}
```

action プロパティには「3」が含まれます (bit 1 と 2 が on の状態)。 特定の対象クラスに対して許可可能な操作が大量にある場合には、 アクションにビットマスクを使用することにより、 パーミッションの記録に必要な保存領域が大幅に低減します。

重要

mask の指定値は 2 の n 乗になっていなければなりません。

15.6.10.3.5. 識別子ポリシー

パーミッションを保存、参照する場合には、**JpaPermissionStore** は特定のオブジェクトのインスタンスを固有に識別できなければなりません。 これを行うには各対象クラスに **識別子ストラテジー** を割り当てることにより一意の識別子の値を生成することができます。 各識別子のストラテジー実装は特定のクラスタイプの一意識別子の生成方法を認識するので、 新しい識別子ストラテジーはシンプルです。

IdentifierStrategy インターフェースは非常に単純で、2 つのメソッドを宣言しているだけです。

```
public interface IdentifierStrategy {
  boolean canIdentify(Class targetClass);
  String getIdentifier(Object target);
}
```

識別子ストラテジーが指定された対象クラスに対し固有の識別子を生成することが可能な場合は、最初のメソッド **canIdentify()** は **true** を返します。第 2 のメソッド **getIdentifier()** は指定された対象オブジェクトに対して一意の識別子の値を返します。

Seam は 2 つの **IdentifierStrategy** 実装、 **ClassIdentifierStrategy**、**EntityIdentifierStrategy** も提供しています。 これについては次項で説明します。

特定のクラスに対して任意の識別子ストラテジーを明示的に設定する場合は、 そのストラテジーに **org.jboss.seam.annotations.security.permission.Identifier** アノテーションを付与し、 **IdentifierStrategy** インターフェースの具体的な実装に値を設定します。 **name** プロパティを指定することもできます (効果は使用される **IdentifierStrategy** 実装により異なります)。

15.6.10.3.6. ClassIdentifierStrategy

この識別子ストラテジーはクラスに一意の識別子を生成し、(指定されていれば) **@Identifier** アノテーション中の **name** の値を使用します。 **name** プロパティを与えないと識別子ストラテジーはクラスのコンポーネント名を使用しようとします (クラスが Seam コンポーネントの場合)。 最後の手段としてクラス名に基づいた識別子を作成します (パッケージ名を除く)。 たとえば、 以下の例にあるクラスの識別子は **customer** となります。

```
@Identifier(name = "customer")
public class Customer {...}
```

以下のクラスの識別子は **customerAction** となります。

```
@Name("customerAction")
public class CustomerAction {...}
```

最後に、 以下のクラスの識別子は **Customer** となります。

```
public class Customer {...}
```

15.6.10.3.7. EntityIdentifierStrategy

この識別子ストラテジーはエンティティ Bean に一意の識別子を生成します。エンティティの主キーを示す文字列とエンティティの名前 (または設定された名前) をつなぎ合わせます。 識別子の名前セクションの生成ルールは **ClassIdentifierStrategy** のルールと同様です。主キーの値 (エンティティの ID) は **PersistenceProvider** コンポーネントを使って取得でき、 Seam アプリケーションでどの永続性実装を使用しているかに関わらず値を決定できます。 **@Entity** でアノテートされていないエンティティについては、 エンティティクラス自身に下のように明示的にアイデンティティストラテジーを設定する必要があります。

```
@Identifier(value = EntityIdentifierStrategy.class)
public class Customer {...}
```

次のエンティティクラスがあるとします。

```
@Entity
public class Customer {
  private Integer id;
  private String firstName;
  private String lastName;

  @Id
  public Integer getId() { return id; }
  public void setId(Integer id) { this.id = id; }

  public String getFirstName() { return firstName; }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() { return lastName; }
  public void setLastName(String lastName) { this.lastName = lastName; }
}
```

id が **1** の **Customer** のインスタンスに対する識別子は**Customer:1** となります。 エンティティクラスに次のような明示的な識別子名のアノテーションがあれば

```
@Entity @Identifier(name = "cust")
public class Customer {...}
```

結果として、**id** が **123** の **Customer** は 「**cust:123**」という識別子を持つことになります。

15.7. パーミッション管理

Seam Security ではアイデンティティ管理 API によりユーザーとロールの管理を行うのと同様、ここではパーミッション管理 API により永続的なユーザーのパーミッションを管理することが可能です (**PermissionManager** コンポーネント)。

15.7.1. PermissionManager

PermissionManager コンポーネントはアプリケーションスコープの Seam コンポーネントであり、数種類のパーミッション管理の方法を提供しています。 使用する前にパーミッションストアで設定する必要があります。 デフォルトでは **JpaPermissionStore** の使用を試行します。 カスタムのパーミッションストアを設定するためには、**components.xml** に **permission-store** プロパティを指定します。

```
<security:permission-manager permission-store="#{ldapPermissionStore}"/>
```

以下の表に **PermissionManager** 提供の各メソッドの詳細を示します。

表15.11 PermissionManager API のメソッド

戻り値のタイプ	メソッド	詳細
List<Permission>	**listPermissions(Object target, String action)**	指定されたターゲットとアクションに対して付与されたすべてのパーミッションを示す **Permission** オブジェクトの一覧を返します。
List<Permission>	**listPermissions(Object target)**	指定されたターゲットとアクションに対して付与されたすべてのパーミッションを示す **Permission** オブジェクトの一覧を返します。
boolean	**grantPermission(Permission permission)**	バックエンドのパーミッションストアに指定した **Permission** を永続化 (許可) します。 操作が成功すると **true** を返します。
boolean	**grantPermissions(List<Permission> permissions)**	バックエンドのパーミッションストアに指定した複数の **Permission** の 一覧を永続化 (許可) します。 操作が成功すると **true** を返します。
boolean	**revokePermission(Permission permission)**	バックエンドのパーミッションストアから指定した **Permission** を削除 (無効) にします。 操作が成功すると **true** を返します。

戻り値のタイプ	メソッド	詳細
boolean	**revokePermissions(List<Permission> permissions)**	バックエンドのパーミッションストアから指定した複数の **Permission** の一覧を削除 (無効) にします。 操作が成功すると **true** を返します。
List<String>	**listAvailableActions(Object target)**	指定された対象ターゲットに対する適用可能な操作の一覧を返します。 このメソッドが返す操作は、対象オブジェクトのクラスに設定されている **@Permission** アノテーションにより異なります。

15.7.2. PermissionManager 操作のためのパーミッションチェック

PermissionManager メソッドを起動するためには、現在の認証ユーザーがその管理操作を実行するために承認されなければなりません。 下の表に特定のメソッド呼び出しに必要なパーミッションを記載します。

表15.12 パーミッション管理用のセキュリティパーミッション

メソッド	パーミッションターゲット	パーミッションのアクション
listPermissions()	指定された **target**	**seam.read-permissions**
grantPermission()	指定された **Permission** の対象、 または指定した複数の **Permission** の一覧に対する各対象 (呼び出すメソッドにより異なる)	**seam.grant-permission**
grantPermission()	指定された **Permission** の対象	**seam.grant-permission**
grantPermissions()	指定された **Permission** の一覧の各対象	**seam.grant-permission**
revokePermission()	指定された **Permission** の対象	**seam.revoke-permission**
revokePermissions()	指定された **Permission** の一覧の各対象	**seam.revoke-permission**

15.8. SSL によるセキュリティ

Seam は HTTPS プロトコルによる機密ページの提供に基本的な対応を行います。これを設定する場合は、**pages.xml** でページの **scheme** を指定します。以下の例では HTTPS を使った **/login.xhtml** ビューの設定方法について示しています。

```
<page view-id="/login.xhtml" scheme="https"/>
```

この設定は自動的に **s:link** や **s:button** の JSF コントロールを拡張しリンクを正しいプロトコルで表示します (**view** を指定した場合)。 前述の例を基にして、 **/login.xhtml** は HTTPS を使用するよう設定されているため以下のリンクは HTTPS を使用します。

```
<s:link view="/login.xhtml" value="Login"/>
```

ユーザーが間違ったプロトコルで直接ページを閲覧するとリダイレクトが引き起こされ、同じビューが正しいプロトコルで再度読み込まれます。 たとえば、 **scheme="https"** ページを HTTP で閲覧しようとすると HTTPS を使用する同じページへのリダイレクトが引き起こされます。

また、 すべてのページに **デフォルトのスキーマ** を設定することも可能です。これは数ページにのみ HTTPS を使用したい場合などに役立ちます。デフォルトのスキーマが指定されていない場合は、 現在のスキーマが使用されます。 従って、 ユーザーが HTTPS を必要とするページにアクセスすると、 HTTPS を必要としないページにユーザーが移行した後も HTTPS が継続して使用されます。これはセキュリティ上は好ましいですが、パフォーマンスは低減します。HTTP をデフォルトの **scheme** として定義するには、次の行を **pages.xml** に追加してください。

```
<page view-id="*" scheme="http" />
```

アプリケーションのいずれのページも HTTPS を **使用しない** 場合はデフォルトのスキーマを定義する必要はありません。

スキーマの変更のたび Seam が自動的に現在の HTTP セッションを無効にするよう設定することができます。そのためには **components.xml** に次の行を追加します。

```
<web:session invalidate-on-scheme-change="true"/>
```

このオプションにより、HTTPS を使うページから HTTP を使うページへの機密データの漏出やセッション ID のスニッフィングからさらに強力に保護します。

15.8.1. デフォルトのポートの上書き

HTTP と HTTPS ポートを手作業で設定したい場合は、**pages** エレメントの **http-port** 属性と **https-port**属性を指定することで **pages.xml** 内で設定することができます。

```
<pages xmlns="http://jboss.com/products/seam/pages"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://jboss.com/products/seam/pages
http://jboss.com/products/seam/pages-2.2.xsd"
       no-conversation-view-id="/home.xhtml"
       login-view-id="/login.xhtml" http-port="8080" https-port="8443">
```

15.9. CAPTCHA

厳密にはセキュリティ API の一部ではありませんが、 自動化されたプロセスがアプリケーションと動作しないようにするために Seam は組み込みの CAPTCHA (**Completely Automated Public Turing test to tell Computers and Humans Apart**) アルゴリズムを提供しています。

15.9.1. CAPTCHA サーブレットの設定

CAPTCHA を使用するには、Seam Resource Servlet を設定する必要があります。これによりページに CAPTCHA チャレンジのイメージを提供します。次を **web.xml** に追加します。

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-
class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

15.9.2. フォームへの CAPTCHA の追加

CAPTCHA チャレンジをフォームに追加するのは簡単です。

```
<h:graphicImage value="/seam/resource/captcha"/>
  <h:inputText id="verifyCaptcha" value="#{captcha.response}"
     required="true">
    <s:validate />
  </h:inputText>
<h:message for="verifyCaptcha"/>
```

必要なのはこれだけです。 **graphicImage** コントロールが CAPTCHA チャレンジを表示し、 **inputText** がユーザーからのレスポンスを受けます。 このレスポンスはフォームが送信された時に自動的に CAPTCHA と検証されます。

15.9.3. CAPTCHA アルゴリズムのカスタマイズ

組み込みコンポーネントを無効にすることにより CAPTCHA のアルゴリズムをカスタマイズできます。

```
@Name("org.jboss.seam.captcha.captcha")
@Scope(SESSION)
public class HitchhikersCaptcha extends Captcha
{
  @Override @Create
  public void init() {
    setChallenge("What is the answer to life, the universe and
everything?");
    setCorrectResponse("42");
  }

  @Override
  public BufferedImage renderChallenge() {
    BufferedImage img = super.renderChallenge();
    img.getGraphics().drawOval(5, 3, 60, 14); //add an obscuring
```

```
decoration
    return img;
  }
}
```

15.10. セキュリティイベント

以下の表に特定のセキュリティ関連のイベントに応答して Seam Security が引き起こすイベント（7章*イベント、インターセプタ、例外処理*を参照）をいくつか示します。

表15.13 セキュリティイベント

イベントキー	詳細
org.jboss.seam.security.loginSuccessful	ログイン試行に成功すると引き起こされます。
org.jboss.seam.security.loginFailed	ログイン試行に失敗すると引き起こされます。
org.jboss.seam.security.alreadyLoggedIn	すでに認証されているユーザーが再度ログインを試行した場合に引き起こされます。
org.jboss.seam.security.notLoggedIn	ユーザーがログインしていない場合にセキュリティチェックが失敗すると引き起こされます。
org.jboss.seam.security.notAuthorized	ログインしているがユーザーに十分な権限がないためセキュリティチェックに失敗した場合に引き起こされます。
org.jboss.seam.security.preAuthenticate	ユーザー認証の直前に引き起こされます。
org.jboss.seam.security.postAuthenticate	ユーザー認証の直後に引き起こされます。
org.jboss.seam.security.loggedOut	ユーザーがログアウトした後に引き起こされます。
org.jboss.seam.security.credentialsUpdated	ユーザーの資格情報が変更されている場合に引き起こされます。
org.jboss.seam.security.rememberMe	Identity の rememberMe プロパティが変更されると引き起こされます。

15.11. 別の権限での実行

ユーザーは上位権限で特定の操作を行う必要がある場合があります。 たとえば、 未認証のユーザーが新しいユーザーアカウントを作成する必要があるとしましょう。 Seam Security はこのような状況に **RunAsOperation** クラスで対応します。 このクラスは、単一の一組の操作に対して**Principal** か **Subject** のいずれか、またはユーザーのロールを無効にすることができます。

以下のコード例で **RunAsOperation** の使い方を示します。**addRole()** メソッドを呼び出し、操作の間に「借りる」ロールセットを提供します。**execute()** メソッドは上位特権で実行されるコードを持っています。

```
new RunAsOperation() {
  public void execute() {
    executePrivilegedOperation();
  }
}.addRole("admin")
 .run();
```

同様に、 **getPrincipal()** や **getSubject()** メソッドを無効にしてその操作の間だけ **Principal** インスタンスや **Subject** インスタンスを使用するよう指定することができます。 最後に、 **RunAsOperation** を実行するために **run()** メソッドを使用します。

15.12. IDENTITY コンポーネントの拡張

アプリケーションに特殊なセキュリティ要件がある場合には Identity コンポーネントを拡張する必要があるかもしれません。 次の例では追加の **companyCode** フィールドで拡張した Identity コンポーネントによって資格証明が処理される例を示します (通常は **Credentials** コンポーネントで処理されます)。 **APPLICATION** の install precedence により、組み込みの Identity に代わりこの拡張された Identity が必ずインストールされるようにします。

```
@Name("org.jboss.seam.security.identity")
@Scope(SESSION)
@Install(precedence = APPLICATION)
@BypassInterceptors
@Startup
public class CustomIdentity extends Identity {
  private static final LogProvider log =
                       Logging.getLogProvider(CustomIdentity.class);

  private String companyCode;

  public String getCompanyCode() {
    return companyCode;
  }

  public void setCompanyCode(String companyCode) {
    this.companyCode = companyCode;
  }

  @Override
  public String login() {
    log.info("###### CUSTOM LOGIN CALLED ######");
    return super.login();
  }
}
```

警告

Identity コンポーネントは **@Startup** の印を付けてください。 これにより **SESSION** コンテキストが開始された直後に使用できるようになります。これをしないと、特定の Seam 機能が使用するアプリケーションで動作しないことがあります。

15.13. OPENID

警告

Technology Preview の機能は Red Hat サブスクリプションレベルアグリーメント (SLA) では完全に対応していません。また、機能的に完全ではない場合があるため実稼働での使用を目的としていません。ただし、こうした機能により今後の新製品開発に早くアクセスすることができるため、開発段階でお客様が機能性をテストしたり、フィードバックをお寄せいただくことができます。Red Hat は 今後強化された Technology Preview の機能を一般的に利用できるよう検討しており、商業的に合理的な範囲でお客様がこうした機能を使用しているときに直面するすべての問題の解決に向けて努力します。

OpenID は外部の Web ベース認証用のコミュニティ標準です。 いずれの Web アプリケーションでもユーザー選択の外部 OpenID サーバーに役割を委任することでそのローカルの認証処理を補完 (または置換) することができます。 ユーザー (複数の Web アプリケーションのそれぞれのログイン詳細を覚えておく必要がなくなる) にとっても開発者 (複雑な認証システム全体を管理する必要がなくなる) にとっても利点となります。

OpenID を使用する場合、 ユーザーが OpenID プロバイダを選択しそのプロバイダがユーザーに OpenID を割り当てます。 ID は **http://maximoburrito.myopenid.com** などの URL 形式をとります (識別子の **http://** の部分はサイトにログインするときは省略して構いません)。 Web アプリケーション (**relying party (証明書利用者)** として知られる) は接続する OpenID サーバーを決定し、認証用のリモートサイトにリダイレクトします。認証に成功するとそのユーザーには本人のアイデンティティを証明する (暗号化されて安全な) トークンが与えられ、元の Web アプリケーションに戻されます。 これでローカルの Web アプリケーションはアプリケーションにアクセスしているユーザーが提示された OpenID を所有していると仮定できます。

ただし、 認証は承認を意味するわけではありません。 Web アプリケーションは OpenID 認証の取扱い方法を確定する必要があります。 Web アプリケーションは即座にログインしたとしてユーザーを取り扱うよう選択し、システムへの完全なアクセスを許可することができます。 または、 OpenID をローカルユーザーアカウントにマッピングして未登録のユーザーに登録するよう求めることができます。これはローカルアプリケーションの設計上の決定事項です。

15.13.1. OpenID の設定

Seam は **opemid4java** パッケージを使用し、 Seam 統合を利用するため 4 種類の **JAR** が追加で必要です。 **htmlparser.jar**、 **openid4java.jar**、 **openxri-client.jar**、 **openxri-syntax.jar** です。

OpenID 処理には **OpenIdPhaseListener** を必要とし、 **faces-config.xml** ファイルに追加する必要があります。 フェーズリスナーは OpenID プロバイダからのコールバックを処理してローカルアプリケーションへの再エントリを許可します。

```
<lifecycle>
  <phase-listener>
    org.jboss.seam.security.openid.OpenIdPhaseListener
  </phase-listener>
</lifecycle>
```

この構成により使用するアプリケーションへの OpenID のサポートを利用可能にします。 OpenID サポートコンポーネントとなる **org.jboss.seam.security.openid.openid** は、**openid4java** クラス群がクラスパスにある場合には自動的にインストールされます。

15.13.2. OpenIDLgin フォームの提示

OpenID ログインを開始するには、ユーザーの OpenID を要求するそのユーザーにフォームを提示します。**#{openid.id}** の値がユーザーの OpenID を受け取り、 **#{openid.login}** アクションが認証要求を開始します。

```
<h:form>
  <h:inputText value="#{openid.id}" />
  <h:commandButton action="#{openid.login}" value="OpenID Login"/>
</h:form>
```

ユーザーがログインフォームをサブミットするとユーザーの OpenID プロバイダにリダイレクトされます。 最終的にユーザーは **OpenIdPhaseListener** によって提供される Seam の疑似ビュー **/openid.xhtml** を介してアプリケーションに戻されます。 アプリケーションは、 ユーザーがそのアプリケーションから全く離れなかったかのようにそのビューから **pages.xml** の操作で OpenID のレスポンスを処理できます。

15.13.3. 即時ログイン

もっともシンプルなストラテジーは、ユーザーを単純に即時ログインさせることです。 次のナビゲーションルールでは **#{openid.loginImmediately()}** アクションを使ってこれを処理する方法を示します。

```
<page view-id="/openid.xhtml">
  <navigation evaluate="#{openid.loginImmediately()}">
    <rule if-outcome="true">
      <redirect view-id="/main.xhtml">
        <message>OpenID login successful...</message>
      </redirect>
    </rule>
    <rule if-outcome="false">
      <redirect view-id="/main.xhtml">
        <message>OpenID login rejected...</message>
      </redirect>
```

```
    </rule>
  </navigation>
</page>
```

loginImmediately() アクションは OpenID が有効であるかどうかを確認します。 有効であればアイデンティティコンポーネントに **OpenIdPrincipal** が追加され、ユーザーがログインしたと印を付けます (**#{identity.loggedIn}** に **true** の印を付ける)。そして **loginImmediately()** アクションが **true** を返します。OpenID が有効ではない場合、 メソッドは **false** を返してそのユーザーはアプリケーションに未認証で入ります。 ユーザーの OpenID が有効の場合、 **#{openid.validatedId}** の式を使ってアクセス可能となるため **#{openid.valid}** は true になります。

15.13.4. ログインの保留

アプリケーションにユーザーを即時ログインさせたくない場合、 ナビゲーションは **#{openid.valid}** プロパティを確認して、ユーザーをローカルの登録または処理ページにリダイレクトしなければなりません。 そこでさらに情報を求めてからローカルのユーザーアカウントを作成する、または CAPTCHA を提示してプログラム的な登録を回避することができます。 プロセスが完了したら、**org.jboss.seam.security.openid.OpenId** コンポーネントとの直接連携または EL (前述を参照) のいずれかを介して **loginImmediately** メソッドを呼び出すことでユーザーをログインさせることができます。 また、 カスタムのコードを記述して Seam のアイデンティティコンポーネントと連携させることによりカスタマイズな操作を作成することもできます。

15.13.5. ログアウト

ログアウト (OpenID 関連を忘れる) は **#{openid.logout}** を呼び出すことで実行します。 このメソッドは Seam Security を使用していない場合は直接呼び出すことができます。Seam Security を使用している場合は継続して **#{identity.logout}** を使用し、イベントハンドラをインストールしてログアウトイベントをキャプチャし、 OpenID ログアウトメソッドを呼び出します。

```
<event type="org.jboss.seam.security.loggedOut">
  <action execute="#{openid.logout}" />
</event>
```

これを必ず含めてください。含めないとユーザーは同じセッションに再度ログインすることができません。

第16章 国際化とローカリゼーションおよびテーマ

アプリケーションの国際化およびローカライズにはいくつかのステージが必要となります。

注記

国際化の機能は JSF コンテキストでのみ使用可能です。

16.1. アプリケーションの国際化

Java EE 5 アプリケーションは数多くのコンポーネントから構成されています。アプリケーションをローカライズするにはこれらのコンポーネントすべてが正しく設定されていなければなりません。

開始する前に、 データベースのサーバーとクライアントがロケールに対応する正しい文字エンコーディングを使用していることを確認します。 通常、 UTF-8 エンコーディングを使用するのが一般的です (正しいエンコーディングの設定方法については本チュートリアルの範囲外です)。

16.1.1. アプリケーションサーバーの設定

アプリケーションサーバーが要求パラメータを正しいエンコーディングでクライアントの要求から受け取るようにするには、Tomcat コネクタを設定する必要があります。**URIEncoding="UTF-8"** 属性を **$JBOSS_HOME/server/$PROFILE/deploy/jboss-web.deployer/server.xml** のコネクタ設定に追加します。

```
<Connector port="8080" URIEncoding="UTF-8"/>
```

代わりに、JBoss AS に要求パラメータの適切なエンコーディングはその要求から取得するよう指示することもできます。

```
<Connector port="8080" useBodyEncodingForURI="true"/>
```

16.1.2. 翻訳されたアプリケーション文字列

アプリケーション内のすべての **メッセージ** にも翻訳された文字列が必要となります (ビューのフィールドラベルなど)。 まず、 リソースバンドルが目的の文字エンコーディングを使ってコード化されることを確認します。 デフォルトでは ASCII が使用されます。 ASCII は十分多くの言語に対応していますが、 すべての言語の文字を取り扱っているわけではありません。

リソースバンドルは ASCII で作成されるか、Unicode 文字の表示に Unicode エスケープのコードを使用する必要があります。 バイトコードに対してプロパティファイルをコンパイルしないため、使用する文字セットを JVM に伝える方法がありません。 このため、 ASCII 文字または ASCII 文字セットにはないエスケープ文字を使用しなければなりません。 **\uXXXX** を使用するといずれの Java ファイルでも Unicode 文字を表すことができます。 ***XXXX*** はその文字を表す 16 進数です。

ラベルの翻訳はメッセージリソースバンドルにネイティブエンコーディングで書き込むことができます (「ラベル」)。ネイティブエンコーディングで書き込んだファイルの内容は、JDK で提供される **native2ascii** ツールを使って、Unicode エスケープシーケンスで ASCII 以外の文字を表すものに変換することができます。

このツールの使い方は Java 6 の場合はこちら に記載されています。 たとえば、 あるファイルを UTF-8 から変換するには次のようにします。

```
 $ native2ascii -encoding UTF-8 messages_cs.properties >
messages_cs_escaped.properties
```

16.1.3. その他のエンコーディング設定

確認する必要があるのは、正しい文字セットを使ってローカライズされたデータとメッセージが表示されること、そしてサブミットされたデータがすべて正しいエンコーディングを使用することです。

表示文字のエンコーディングを設定するには **<f:view locale="cs_CZ"/>** タグを使用します (この **locale** 値は JSF にチェコ語を使用するよう指示します)。XML にローカライズされた文字列を埋め込む場合に、XML ドキュメント自体のエンコーディングを変更したいと思われるかもしれません。これを行うには XML 宣言の **<?xml version="1.0" encoding="UTF-8"?>** にある **encoding** 属性値を変更します。

JSF や Facelet は指定文字エンコーディングを使った要求をサブミットするはずです。ただし、エンコーディングを指定していない要求がサブミットされるようにするには、要求エンコーディングにサーブレットフィルタを強制的に使用させることができます。**components.xml** で設定します。

```
<web:character-encoding-filter encoding="UTF-8" override-client="true"
     url-pattern="*.seam" />
```

16.2. ロケール

各ユーザーログインのセッションは、**java.util.Locale** の関連付けられたインスタンスを持ち、アプリケーションに対しては **locale** という名前のコンポーネントとして使用可能です。通常の環境では、ロケールに特別な設定は不要です。Seam ではアクティブなロケールの決定は次のように JSF に委譲します。

- ロケールが HTTP 要求と関連付けられ (ブラウザのロケール)、そのロケールが **faces-config.xml** にあるサポートロケールの一覧にある場合は、そのロケールをその後のセッションに使用します。
- これ以外で、デフォルトロケールが **faces-config.xml** 中に指定されていた場合、そのロケールをその後のセッションに使用します。
- いずれにも該当しない場合、サーバのデフォルトロケールを使用します。

Seam 設定プロパティの **org.jboss.seam.international.localeSelector.language**、**org.jboss.seam.international.localeSelector.country**、**org.jboss.seam.international.localeSelector.variant** により手作業でのロケール設定が**可能**ですが、以前に示した方法ではなくてこれを実行する妥当な理由は考え付きません。

アプリケーションのユーザーインターフェースを使いユーザーに手作業でロケール設定をさせると便利です。Seam はデフォルトのアルゴリズムで決定されるロケールを無効化する組み込み機能を提供しています。JSP または Facelet ページのフォームに以下の断片を追加して行います。

```
<h:selectOneMenu value="#{localeSelector.language}">
  <f:selectItem itemLabel="English" itemValue="en"/>
  <f:selectItem itemLabel="Deutsch" itemValue="de"/>
  <f:selectItem itemLabel="Francais" itemValue="fr"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}"
                 value="#{messages['ChangeLanguage']}"/>
```

あるいは、 **faces-config.xml** のサポートロケールの全一覧が必要な場合は次を使います。

```
<h:selectOneMenu value="#{localeSelector.localeString}">
  <f:selectItems value="#{localeSelector.supportedLocales}"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}"
                 value="#{messages['ChangeLanguage']}"/>
```

ユーザーがドロップダウンからアイテムを選択してコマンドボタンをクリックすると、 その後のセッションに対して Seam と JSF のロケールはオーバライドされます。

組み込みの **org.jboss.seam.international.localeConfig** コンポーネントを使ってサーバーのデフォルトロケールとサポートされているロケールを設定することができます。 まず、 Seam コンポーネント記述子で Seam 国際パッケージの XML 名前空間を宣言します。 次にデフォルトのロケールとサポートされるロケールを次のように定義します。

```
<international:locale-config default-locale="fr_CA"
               supported-locales="en fr_CA fr_FR"/>
```

サポートされるロケールは一致するリソースバンドルが必要であることを忘れないようにしてください。 次に言語固有のラベルを定義します。

16.3. ラベル

<f:loadBundle /> により JSF はユーザーインターフェースのラベルや説明用テキストの国際化に対応しています。 Seam アプリケーションではこの方法をとるか、 組み込みの EL 式を利用したテンプレート化ラベルの表示に Seam の **messages** コンポーネントを利用することができます。

16.3.1. ラベルの定義

Seam の **java.util.ResourceBundle** で利用できる国際化ラベルを **org.jboss.seam.core.resourceBundle** としてアプリケーションに対して使用できるようにします。 デフォルトでは、 Seam で使用されるリソースバンドルは **messages** の名称なので **messages.properties**、 **messages_en.properties**、 **messages_en_AU.properties** などの名称のファイルにラベルを定義する必要があります。 これらのファイルは通常 **WEB-INF/classes** ディレクトリに属します。

従って、 **messages_en.properties** では次のようになります。

```
Hello=Hello
```

そして、 **messages_en_AU.properties** では次のようになります。

```
Hello=G'day
```

org.jboss.seam.core.resourceLoader.bundleNames と呼ばれる Seam 設定プロパティを設定することで、リソースバンドルに別の名前を選択することができます。 リソースバンドル名の一覧を指定してメッセージの検索をさせる (深さ優先) こともできます。

```
<core:resource-loader>
  <core:bundle-names>
    <value>mycompany_messages</value>
```

```
    <value>standard_messages</value>
  </core:bundle-names>
</core:resource-loader>
```

特定のページだけにメッセージを定義したい場合は、 その JSF ビュー ID と同じ名前でリソースバンドルに指定します。 このとき ID の最初の / と最後の拡張子を除去します。 つまり **/welcome/hello.jsp** にのみメッセージを表示したいのであれば、 表示させるメッセージを **welcome/hello_en.properties** に配置します。

pages.xml に明示的なバンドル名を指定することもできます。

```
<page view-id="/welcome/hello.jsp" bundle="HelloMessages"/>
```

これで **HelloMessages.properties** に定義されたメッセージを **/welcome/hello.jsp** で使うことができます。

16.3.2. ラベルの表示

Seam のリソースバンドルを使ってラベルを定義する場合、 各ページそれぞれに **<f:loadBundle ... />** を入力しなくてもラベルを使用することができます。 代わりに以下のように入力します。

```
<h:outputText value="#{messages['Hello']}"/>
```

または

```
<h:outputText value="#{messages.Hello}"/>
```

さらに、 メッセージ自体に EL 式を含ませることができます。

```
Hello=Hello, #{user.firstName} #{user.lastName}
```

```
Hello=G'day, #{user.firstName}
```

コード内にもメッセージを使用することができます。

```
@In private Map<String, String> messages;
```

```
@In("#{messages['Hello']}") private String helloMessage;
```

16.3.3. Faces メッセージ

facesMessages コンポーネントはユーザーに成功か失敗かを表示するのに便利な方法です。 上述した機能は Faces のメッセージにも有効です。

```
@Name("hello")
@Stateless
public class HelloBean implements Hello {
  @In FacesMessages facesMessages;
  public String sayIt() {
```

```
    facesMessages.addFromResourceBundle("Hello");
  }
}
```

ユーザーのロケールに応じて、 **Hello, Gavin King** あるいは **G'day, Gavin** と表示されます。

16.4. タイムゾーン

org.jboss.seam.international.timezone という名称の **java.util.Timezone** のセッションスコープのインスタンスと、**org.jboss.seam.international.timezoneSelector** という名称のタイムゾーンを変更する Seam コンポーネントもあります。 デフォルトでは、タイムゾーンはサーバーのデフォルトタイムゾーンです。 タイムゾーンが **<f:convertDateTime>** を使用して明示的に指定されない限り、 残念ながら JSF 仕様ではすべての日付と時刻は UTC を前提としており、 UTC として表示されます。

Seam はこの動作を無効にして、 すべての日付と時刻を Seam タイムゾーンにデフォルト設定します。 さらに Seam には、 Seam タイムゾーンでの変換を常に行う **<s:convertDateTime>** タグを備えています。

また、 Seam では文字列値を日付に変換するデフォルトの日付変換機能があります。 これにより日付を取得する入力フィールドで変換機能を指定する必要がなくなります。 パターンはユーザーのロケールにより選択され、 タイムゾーンは上述したように選択されます。

16.5. テーマ

Seam アプリケーションはとても簡単にスキン変更をすることも可能です。 テーマ API はローカライゼーション API にとても似ています。ただし、もちろんこれら 2 つの関心事は関連がなく、アプリケーションの中にはローカライゼーションとテーマの両方をサポートするものもあります。

まず、サポートされるテーマのセットを設定します。

```
<theme:theme-selector cookie-enabled="true">
  <theme:available-themes>
    <value>default</value>
    <value>accessible</value>
    <value>printable</value>
  </theme:available-themes>
</theme:theme-selector>
```

最初に記述されたテーマがデフォルトテーマです。

テーマはそのテーマと同じ名前のプロパティファイルにテーマとして定義されます。 例えば、**default** テーマは **default.properties** に一連のエントリとして定義されます。 例えば、**default.properties** は以下のように定義します。

```
css ../screen.css template /template.xhtml
```

通常、 テーマリソースバンドルのエントリは Facelets テンプレートの名前とイメージ、 または CSS スタイルへのパスです (通常はテキストであるローカリゼーションのリソースバンドルとは異なります)。

これで JSP や Facelet ページでこれらのエントリを使えるようになりました。例えば、Facelets ページでスタイルシートを適用するには

```
<link href="#{theme.css}" rel="stylesheet" type="text/css" />
```

あるいは、 サブディレクトリにページ定義が存在している場合は次のようになります。

```
<link href="#{facesContext.externalContext.requestContextPath}#
{theme.css}"
      rel="stylesheet" type="text/css" />
```

最も強力な使い方として Faceletsでは **<ui:composition>** で使用されるテンプレートを適用できます。

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    template="#{theme.template}">
```

ロケール選択と同様、 ユーザーが自由にテーマを変更できる組み込みのテーマ選択があります。

```
<h:selectOneMenu value="#{themeSelector.theme}">
  <f:selectItems value="#{themeSelector.themes}"/>
</h:selectOneMenu>
<h:commandButton action="#{themeSelector.select}" value="Select Theme"/>
```

16.6. クッキーによるロケールとテーマ設定の永続化

ロケール選択、 テーマ選択、 タイムゾーン選択はすべてクッキーに対するロケールとテーマ設定の永続化に対応しています。 単純に **components.xml** で **cookie-enabled** プロパティを設定します。

```
<theme:theme-selector cookie-enabled="true">
  <theme:available-themes>
    <value>default</value>
    <value>accessible</value>
    <value>printable</value>
  </theme:available-themes>
</theme:theme-selector>

<international:locale-selector cookie-enabled="true"/>
```

第17章 SEAM TEXT

多くの人が一緒に作業するウェブサイトでは、 フォーラムへの投稿、 wiki ページ、 ブログ、 コメントなどでフォーマット済みテキストの入力を容易にするために人間が扱いやすいマークアップ言語が必要です。 Seam には **Seam Text** と呼ばれる言語に従ったフォーマット済みテキストを表示するために **<s:formattedText/>** コントロールが備わっています。 Seam Text は ANTLR ベースのパーサを利用して実装します (ANTLR に関する知識は必要ありません)。

17.1. フォーマットの基本

次に簡単な例を示します。**It's easy to make *emphasized*, |monospaced|, ~deleted~, super^scripted^ or _underlined_ text.**

これを **<s:formattedText/>** を使って表示すると、 以下の HTML が生成されます。

```
<p>
  It's easy to make <i>emphasized</i>, <tt>monospaced</tt>,
  <del>deleted</del>, super<sup>scripted</sup> or
  <u>underlined</u> text.
</p>
```

空行は新しいパラグラフを作成するときに使用します。 また、**+** は見出しに使用します。

```
        +This is a big heading
        You /must/ have some text following a heading!

        ++This is a smaller heading
        This is the first paragraph. We can split it across multiple
        lines, but we must end it with a blank line.

        This is the second paragraph.
```

単なる新しい行だけなら無視されます。新しい段落にするためにテキストを配置するには空行が必要です。 これが結果となる HTML です。

```
<h1>This is a big heading</h1>
<p>
  You <i>must</i> have some text following a heading!
</p>

<h2>This is a smaller heading</h2>
<p>
  This is the first paragraph. We can split it across multiple
  lines, but we must end it with a blank line.
</p>

<p>
  This is the second paragraph.
</p>
```

文字は順序指定された一覧のアイテムを作成します。順序指定されていない一覧は**=** 文字を使います。

```
        An ordered list:

        #first item
        #second item
        #and even the /third/ item

        An unordered list:

        =an item
        =another item
```

```
<p>
  An ordered list:
</p>

<ol>
  <li>first item</li>
  <li>second item</li>
  <li>and even the <i>third</i> item</li>
</ol>

<p>
  An unordered list:
</p>

<ul>
  <li>an item</li>
  <li>another item</li>
</ul>
```

引用符付きのセクションは二重引用符で囲みます。

```
        He said:

        "Hello, how are /you/?"

        She answered, "Fine, and you?"
```

```
<p>
  He said:
</p>

<q>Hi, how are
<i>you</i>?</q>

<p>
  She answered, <q>Fine, and you?</q>
</p>
```

17.2. 特殊な文字でのコードとテキストの記述

*****、**|** **#** などの特殊文字や **<, > and &** などの HTML 文字は **** でエスケープすることができます。

```
You can write down equations like 2\*3\=6 and HTML tags like \<body\>
using the escape character: \\.
```

```
<p>
  You can write down equations like 2*3=6 and HTML tags
  like &lt;body&gt; using the escape character: \.
</p>
```

また、バッククオート (`) を使ってコードのブロックを囲むことができます。

```
My code doesn't work:
  `for (int i=0; i<100; i--)
       {
       doSomething();
       }`
Any ideas?
```

```
<p>
  My code doesn't work:
</p>

<pre>for (int i=0; i&lt;100; i--)
{
    doSomething();
}</pre>

<p>
  Any ideas?
</p>
```

固定スペースのフォーマット済みテキストのほとんどはコードか特殊文字を伴うため、インラインの固定スペースフォーマットは常にエスケープします。

```
This is a |<tag attribute="value"/>| example.
```

上記のように固定スペースバー内の文字をエスケープしないで記述することができます。また、その他の方法ではインライン固定スペーステキストをフォーマットすることができないことを意味します。

17.3. リンク

次のようにリンクを作成できます。

```
Go to the Seam website at [=>http://jboss.com/products/seam].
```

リンクテキストを指定したい場合

```
Go to [the Seam website=>http://jboss.com/products/seam].
```

上級者向けには、この構文で記述された **wikiword** のリンクを解釈できるよう **Seam Text** パーサをカスタマイズすることも可能です。

17.4. HTML の記述

テキストには限定された HTML のサブセットを含めることができます (クロスサイトスクリプティング攻撃には利用できないサブセットが選ばれました)。リンク作成時に便利です。

```
You might want to link to
      <a href="http://jboss.com/products/seam">something cool</a>,
      or even include an image: <img src="/logo.jpg"/>
```

テーブルも作成できます。

```
<table>
   <tr><td>First name:</td><td>Gavin</td></tr>
   <tr><td>Last name:</td><td>King</td></tr>
</table>
```

17.5. SEAMTEXTPARSER の使用

<s:formattedText/> JSF コンポーネントは内部的に **org.jboss.seam.text.SeamTextParser** を使用します。 このクラスを直接使って独自のテキスト解析機能、 レンダリング機能、 HTML サニテーションの手順を実装することができます。 JavaScript ベースの HTML エディタなどリッチテキストの入力用にカスタムのフロントエンドインターフェースがある場合、 クロスサイトスクリプティング (XSS) 攻撃から防御する目的でユーザーの入力を確認する場合に便利です。 またカスタムの Wiki テキスト解析やレンダリングエンジンとしても使用できます。

次の例ではカスタムのテキストパーサを定義しています。 これはデフォルトの HTML サニタイザーを無効にします。

```
public class MyTextParser extends SeamTextParser {

    public MyTextParser(String myText) {
        super(new SeamTextLexer(new StringReader(myText)));

        setSanitizer(
                    new DefaultSanitizer() {
                        @Override
                        public void validateHtmlElement(Token element)
throws SemanticException {
                            // TODO: I want to validate HTML elements
myself!
                        }
                    }
                    );
    }

    // Customizes rendering of Seam text links such as [Some
Text=&gt;http://example.com]
    @Override
    protected String linkTag(String descriptionText, String linkText) {
        return "&lt;a href=\"" + linkText + "\"&gt;My Custom Link: " +
            descriptionText + "&lt;/a&gt;";
    }

    // Renders a &lt;p&gt; or equivalent tag
    @Override
    protected String paragraphOpenTag() {
```

```
            return "&lt;p class=\"myCustomStyle\"&gt;";
        }

        public void parse() throws ANTLRException {
            startRule();
        }

    }
```

linkTag() と **paragraphOpenTag()** メソッドは、レンダリングされた出力をカスタマイズするために無効にすることができる 2 種類のメソッドです。 これらのメソッドは通常、 **String** 出力を返します。 詳細については Java ドキュメントを参照してください。
org.jboss.seam.text.SeamTextParser.DefaultSanitizer Java ドキュメントには HTML のエレメント、 属性、 デフォルトでフィルタされる属性値などの詳細も記載されています。

第18章 ITEXT PDF 生成

Seam には iText を使用してドキュメント生成を行うためのコンポーネントセットが含まれています。Seam iText ドキュメントサポートの主な狙いは PDF ドキュメントの生成ですが、 RTF ドキュメント生成に対しても基本的なサポートを提供します。

18.1. PDF サポートの使用

iText サポートは **jboss-seam-pdf.jar** により提供されます。 この **JAR** には iText JSF コントロール (PDF にレンダリング可能なビューを構成) と DocumentStore コンポーネント (ユーザーにレンダリングしたドキュメントを提供) が含まれます。 アプリケーションに PDF サポートを含ませるには **jboss-seam-pdf.jar** を **WEB-INF/lib** ディレクトリに iText の **JAR** ファイルと共に含ませます。Seam の iText サポートを使用する上で必要な設定はこれだけです。

Seam iText モジュールにはビューテクノロジーとして Facelets を使用する必要があります。 ライブラリの今後のバージョンも JSP の使用に対応する可能性があります。 また、 **seam-ui** パッケージの使用も必要となります。

examples/itext プロジェクトには実行可能なデモ用 PDF サポートのサンプルが含まれています。正確なパッケージ化の導入を行い、 現在サポートされている PDF 生成の主要な機能を実際に示すサンプルがいくつか含まれています。

18.1.1. ドキュメントの作成

<p:document>

説明

ドキュメントは **http://jboss.com/products/seam/pdf** 名前空間にあるタグを使い Facelet XHTML ファイルで生成されます。ドキュメントにはそのルートに必ず **document** タグがあるはずです。**document** タグは Seam がドキュメントを DocumentStore に生成し、その格納コンテンツに HTML リダイレクトをレンダリングするよう準備を行います。

Attributes

type

生成されるドキュメントのタイプです。有効な値は **PDF**、**RTF**、**HTML** です。Seam のデフォルト設定は PDF 生成となるため、多くの機能は PDF ドキュメント生成時にのみ正常に動作します。

pageSize

生成されるページのサイズです。最も一般的に使用される値は、**LETTER** と **A4** です。対応するページサイズの全一覧は、**com.lowagie.text.PageSize** クラスにあります。代わりに、pageSize は直接、ページの幅と高さを示すことも可能です。例えば、値「612 792」は LETTER ページサイズと同じです。

orientation

ページの向きです。有効な値は **portrait** と **landscape** です。横向きモードではページの高さと幅のサイズ値が逆になります。

margins

左右と上下の余白の値です。

marginMirroring

余白の設定が交互のページで逆になることを示します。

disposition

Web ブラウザで PDF を生成する場合に、ドキュメントの HTTP **Content-Disposition** を決定します。 有効な値は、 可能であればブラウザウィンドウ内にドキュメントを表示させることを表す **inline** と、 ドキュメントをダウンロードとして処理することを表す **attachment** です。 デフォルト値は **inline** です。

fileName

添付用です。 この値はダウンロードしたファイル名を上書きします。

メタデータ属性

- **title**
- **subject**
- **keywords**
- **author**
- **creator**

使用方法

```
<p:document
xmlns:p="http://jboss.com/products/seam/pdf">
  The document goes here.
</p:document>
```

18.1.2. 基本的なテキストのエレメント

Seam では PDF に適したコンテンツを生成できるよう特殊な UI コンポーネントを提供します。 **<p:image>** タグと **<p:paragraph>** タグはシンプルなドキュメントの基盤を形成します。**<p:font>** のようなタグはスタイル情報を提供します。

<p:paragraph>	**説明**
	
	ほとんどの場合、 テキストを段落ごとに区切ることでテキストの断片に論理的な流れやフォーマットおよびスタイルを与えることができるようになります。
	
	Attributes
	
	• **firstLineIndent**
	• **extraParagraphSpace**
	• **leading**
	• **multipliedLeading**
	• **spacingBefore** – エレメントの前に空白スペースが挿入されます。
	• **spacingAfter** – エレメントの後に空白スペースが挿入されます。
	• **indentationLeft**
	• **indentationRight**
	• **keepTogether**
	
	使用方法
	
	`<p:paragraph alignment="justify">`
	`  This is a simple document. It isn't very fancy.`
	`</p:paragraph>`

<p:text>	**説明**
	
	text タグにより通常の JSF 変換メカニズムを使用して、アプリケーションデータからテキストの断片を生成することができます。HTML ドキュメントをレンダリングする場合に使用する **outputText** タグと非常に似ています。
	
	Attributes
	
	• **value** – 表示される値です。 一般的には値バインディング式になります。
	
	使用方法
	
	`<p:paragraph>`
	`  The item costs <p:text value="#{product.price}">`
	`  <f:convertNumber type="currency"`
	`  currencySymbol="$"/>`
	`  </p:text>`
	`</p:paragraph>`

<p:html>

説明

html タグは HTML コンテンツを PDF にレンダリングします。

Attributes

- **value** – 表示されるテキストです。

使用方法

```
<p:html value="This is HTML with <b>some markup</b>"
/>
<p:html>
  <h1>This is more complex HTML</h1>
  <ul>
    <li>one</li>
    <li>two</li>
    <li>three</li>
  </ul>
</p:html>

<p:html>
    <s:formattedText value="*This* is |Seam Text| as
                            HTML.
        It's very^cool^." />
</p:html>
```

<p:font>

説明

font タグはその中のすべてのテキストに使用されるデフォルトフォントを定義します。

Attributes

- **name** – フォントの名前です。例えば、**COURIER**、**HELVETICA**、**TIMES-ROMAN**、**SYMBOL**、**ZAPFDINGBATS** です。
- **size** – フォントのポイントサイズです。
- **style** – フォントのスタイルです。以下の組み合わせとなります。**NORMAL**、**BOLD**、**ITALIC**、**OBLIQUE**、**UNDERLINE**、**LINE-THROUGH**
- **encoding** – 文字セットエンコーディングです。

使用方法

```
<p:font name="courier" style="bold" size="24">
  <p:paragraph>My Title</p:paragraph>
</p:font>
```

<p:textcolumn>	**説明** **p:textcolumn** はテキストの流れを制御するために使用可能なテキスト列を挿入します。最も一般的な使用方法は、右から左の方向のフォントに対応することです。 **Attributes** • **left** – テキスト列の左の範囲です。 • **right** – テキスト列の右の範囲です。 • **direction** – 列のテキスト表記の向きです。**RTL**、**LTR**、**NO-BIDI**、**DEFAULT** **使用方法** `<p:textcolumn left="400" right="600" direction="rtl"> <p:font name="/Library/Fonts/Arial Unicode.ttf" encoding="Identity-H" embedded="true">#{phrases.arabic}</p:font> </p:textcolumn>`
<p:newPage>	**説明** **p:newPage** は改ページを挿入します。 **使用方法** `<p:newPage />`

<p:image>	**説明**
	
	p:image は画像をドキュメントに挿入します。画像はクラスパスまたは Web アプリケーションコンテキストから **value** 属性を使ってロードすることができます。
	
	リソースはアプリケーションコードで動的に生成することもできます。**imageData** 属性は値バインディング式を指定することができ、この値は **java.awt.Image** オブジェクトです。
	
	Attributes
	
	• **value** – アプリケーションが生成した画像にバインドするリソース名またはメソッド式です。
	• **rotation** – 度数で表される画像の回転です。
	• **height** – 画像の高さです。
	• **width** – 画像の幅です。
	• **alignment** – 画像の位置です (可能な値については「位置調整の値」を参照)。
	• **alt** – 画像の別のテキスト表現です。
	• **indentationLeft**
	• **indentationRight**
	• **spacingBefore** – エレメントの前に空白スペースが挿入されます。
	• **spacingAfter** – エレメントの後に空白スペースが挿入されます。
	• **widthPercentage**
	• **initialRotation**
	• **dpi**
	• **scalePercent** – 画像に使用する倍率です (パーセンテージ)。単一のパーセンテージ値として表すか、**x** と **y** の別々の倍率値を表す 2 つのパーセンテージ値として表すこともできます。
	• **wrap**
	• **underlying**
	
	使用方法
	
	`<p:image value="/jboss.jpg" />`
	
	`<p:image value="#{images.chart}" />`

<p:anchor>	**説明**
	
	p:anchor はドキュメントからクリックできるリンクを定義します。次の属性に対応します。
	
	Attributes
	
	• **name** – ドキュメント内のアンカーの目的点の名前です。
	
	• **reference** – リンクの参照先です。ドキュメント内の別のポイントへのリンクは「**#**」で開始します。例えば「**#link1**」は **link1** の **name** が付いたアンカーの位置を参照します。ドキュメントの外にあるリソースをポイントするには、リンクは完全な URL である必要があります。
	
	使用方法
	
	`<p:listItem>`
	`  <p:anchor reference="#reason1">Reason 1</p:anchor>`
	`</p:listItem>`
	`...`
	`<p:paragraph>`
	`  <p:anchor name="reason1">`
	`  It's the quickest way to get "rich"`
	`  </p:anchor>`
	`  ...`
	`</p:paragraph>`

18.1.3. ヘッダーとフッター

<p:header>
 	
<p:footer>
	説明
	
	p:header と **p:footer** のコンポーネントにより、生成されたドキュメントの各ページにヘッダーとフッターを配置します。ヘッダーとフッターの宣言はドキュメントの冒頭に出現するはずです。
	
	Attributes
	
	• **alignment** – ヘッダーとフッターのボックスセクションの位置です (位置の値については 「位置調整の値」 を参照)。
	
	• **backgroundColor** – ヘッダーとフッターボックスの背景色です (色の値については 「色の値」 を参照)。
	
	• **borderColor** – ヘッダーとフッターボックスの境界線の色です。**borderColorLeft**、**borderColorRight**、**borderColorTop**、**borderColorBottom** を使用して境界線ごとに設定が可能です (色の値については 「色の値」 を参照)。
	
	• **borderWidth** – 境界線の太さです。**borderWidthLeft**、**borderWidthRight**、**borderWidthTop**、**borderWidthBottom** を使用して境界線ごとに指定が可能です。
	
	使用方法
	
	`<p:facet name="header">`
	`  <p:font size="12">`
	`  <p:footer borderWidthTop="1"`
	`borderColorTop="blue"`
	`  borderWidthBottom="0" alignment="center">`
	`  Why Seam? [<p:pageNumber />]`
	`  </p:footer>`
	`  </p:font>`
	`</f:facet>`

<p:pageNumber>	**説明**
	
	現在のページ番号は **p:pageNumber** タグを使うとヘッダーまたはフッターの内側に配置できます。このページ番号タグはヘッダーまたはフッターのコンテキスト内でのみ、1 度だけ使用可能です。
	
	使用方法
	
	`<p:footer borderWidthTop="1" borderColorTop="blue"`
	`  borderWidthBottom="0" alignment="center">`
	`  Why Seam? [<p:pageNumber />]`
	`</p:footer>`

18.1.4. 章とセクション

<p:chapter>
 	
<p:section>
	説明
	
	生成されるドキュメントが book または article の構造をとる場合、**p:chapter** と **p:section** のタグを使用して構成することができます。セクションは章の内側でのみ使用できますが、必要に応じていずれの深さにもネストさせることができます。ほとんどの PDF ビューアはドキュメント内の章とセクション間を簡単に移動できる機能を備えています。
	
	Attributes
	
	• **alignment** – ヘッダーとフッターのボックスセクションの位置です (位置の値については 「位置調整の値」 を参照)。
	
	• **number** – 章番号です。 各章すべてに章番号を割り当ててください。
	
	• **numberDepth** – セクションの番号付けの深さです。すべてのセクションはその前後の章やセクションに応じて番号が付けられます。デフォルトの番号の深さ **3** で表示すると、第 3 章の第 1 セクションにある 4 番目のセクションは 3.1.4 となります。章番号を省略するには、番号の深さに **2** を使用します。この場合、セクション番号は 1.4 と表示されます。
	
	使用方法
	
	`<p:document`
	`xmlns:p="http://jboss.com/products/seam/pdf"`
	`title="Hello">`
	`<p:chapter number="1">`
	
	`<p:title><p:paragraph>Hello</p:paragraph></p:title>`
	`<p:paragraph>Hello #{user.name}!</p:paragraph>`
	`</p:chapter>`
	
	`<p:chapter number="2">`
	`<p:title>`
	`<p:paragraph>`
	`Goodbye`
	`</p:paragraph>`
	`</p:title>`
	`<p:paragraph>Goodbye #{user.name}.</p:paragraph>`
	`</p:chapter>`
	
	`</p:document>`

<p:header>	**説明** いずれの章やセクションにも **p:title** を含むことができます。タイトルは章やセクション番号の隣に表示されます。タイトルの本文には raw テキストを含めるか、**p:paragraph** とすることも可能です。

18.1.5. 一覧

一覧の構成は **p:list** と **p:listItem** のタグを使って表示させることができます。 一覧には適宜ネストされたサブリストを含ませることもできます。 一覧のアイテムは一覧の外側では使用できません。次のドキュメントは **ui:repeat** タグを使って Seam コンポーネントから取得した値の一覧を表示しています。

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
   xmlns:ui="http://java.sun.com/jsf/facelets" title="Hello">

  <p:list style="numbered">
    <ui:repeat value="#{documents}" var="doc">
      <p:listItem>#{doc.name}</p:listItem>
    </ui:repeat>
  </p:list>

</p:document>
```

<p:list>	**Attributes** • **style** – 一覧の並び順や箇条書きのスタイルです。**NUMBERED**、**LETTERED**、**GREEK**、**ROMAN**、**ZAPFDINGBATS**、**ZAPFDINGBATS_NUMBER** のいずれかです。スタイルの指定がない場合は、一覧のアイテムはデフォルトで箇条書きになります。 • **listSymbol** – 箇条書きリストの場合、 箇条書きの文の前に付ける記号を指定します。 • **indent** – 一覧のインデントのレベルです。 • **lowerCase** – 文字を使った一覧スタイルの場合、 その文字を小文字にするかどうかを示します。 • **charNumber** – **ZAPFDINGBATS** の場合、 箇条書きに使用する文字コードを示します。 • **numberType** – **ZAPFDINGBATS_NUMBER** の場合、 番号付けのスタイルを示します。 **使用方法** `<p:list style="numbered"> <ui:repeat value="#{documents}" var="doc"> <p:listItem>#{doc.name}</p:listItem> </ui:repeat> </p:list>`

<p:listItem>	**説明** **p:listItem** は次の属性に対応しています。 **Attributes** • **alignment** – ヘッダーとフッターのボックスセクションの位置です (位置の値については 「位置調整の値」 を参照)。 • **alignment** – 一覧アイテムの位置です (可能な値については「位置調整の値」 を参照)。 • **indentationLeft** – 左のインデントの数です。 • **indentationRight** – 右のインデントの数です。 • **listSymbol** – この一覧のアイテムに対しデフォルトの一覧の記号を上書きします。 **使用方法** `...`

18.1.6. 表

表の構成は **p:table** と **p:cell** のタグを使って作成することができます。 多くの表構成とは異なり、明示的な行の宣言はありません。 表に列が 3 つある場合は、3 セルで自動的に行を 1 つ形成します。 ヘッダーとフッターの行を宣言することができ、 表の構成が複数ページに渡る場合にはヘッダーとフッターは繰り返し使用されます。

<p:table>	**説明**
	
	p:table は次の属性に対応しています。
	
	Attributes
	
	• **columns** – 表の 1 行を構成する列数 (セル数) です。
	• **widths** – 各列の相対幅です。各列に対して値は 1 つです。例えば、**widths="2 1 1"** の場合、列は 3 つあり 1 番目の列幅は 2 番目と 3 番目の列の 2 倍になることを示しています。
	• **headerRows** – ヘッダーとフッターの行とみなされる最初の行数です。 表が複数ページに渡る場合は繰り返し使用されます。
	• **footerRows** – フッターの行とみなされる行数です。**headerRows** 値からこの値が差し引かれます。ヘッダーを構成する行が 2 つ、フッターを構成する行が 1 つあるドキュメントの場合、**headerRows** は **3** に **footerRows** は **1** にそれぞれ設定されます。
	• **widthPercentage** – 表でまたがるページ幅の割合です。
	• **horizontalAlignment** – 表の水平位置です (可能な値については「位置調整の値」 を参照)。
	• **skipFirstHeader**
	• **runDirection**
	• **lockedWidth**
	• **splitRows**
	• **spacingBefore** – エレメントの前に空白スペースが挿入されます。
	• **spacingAfter** – エレメントの後に空白スペースが挿入されます。
	• **extendLastRow**
	• **headersInEvent**
	• **splitLate**
	• **keepTogether**
	
	使用方法
	
	`<p:table columns="3" headerRows="1">`
	`  <p:cell>name</p:cell>`
	`  <p:cell>owner</p:cell>`
	`  <p:cell>size</p:cell>`
	`  <ui:repeat value="#{documents}" var="doc">`
	`  <p:cell>#{doc.name}</p:cell>`
	`  <p:cell>#{doc.user.name}</p:cell>`
	`  <p:cell>#{doc.size}</p:cell>`
	`  </ui:repeat>`
	`</p:table>`

<p:cell>	**説明** **p:cell** は次の属性に対応しています。 **Attributes** • **colspan** – **colspan** を 1 より大きい値に宣言することでセルが複数の列にまたがることができます。セルは複数行にまたがることはできません。 • **horizontalAlignment** – セルの水平位置です (可能な値については「位置調整の値」 を参照)。 • **verticalAlignment** – セルの垂直位置です (可能な値については「位置調整の値」 を参照)。 • **padding** – **paddingLeft**、**paddingRight**、**paddingTop**、**paddingBottom** を使って特定のサイドのパディングを指定します。 • **useBorderPadding** • **leading** • **multipliedLeading** • **indent** • **verticalAlignment** • **extraParagraphSpace** • **fixedHeight** • **noWrap** • **minimumHeight** • **followingIndent** • **rightIndent** • **spaceCharRatio** • **runDirection** • **arabicOptions** • **useAscender** • **grayFill** • **rotation** **使用方法** `<p:cell>...</p:cell>`

18.1.7. ドキュメントの定数

本項では複数のタグで属性により共有される定数をいくつか説明します。

18.1.7.1. 色の値

Seam のドキュメントはまだフルカラー仕様に対応していません。 現在、 次の指定色にのみ対応しています。**white**、 **gray**、 **lightgray**、 **darkgray**、 **black**、 **red**、 **pink**、 **yellow**、 **green**、 **magenta**、 **cyan**、 **blue** です。

18.1.7.2. 位置調整の値

Seam PDF は次の横方向の位置調整値、 **left**、 **right**、 **center**、 **justify**、 **justifyall** に対応します。 縦方向の値は **top**、 **middle**、 **bottom**、 **baseline** です。

18.2. グラフ

グラフ作成のサポートも **jboss-seam-pdf.jar** で提供されます。 グラフは PDF ドキュメント内で使用でき、またはグラフは HTML ページ内でイメージになります。 グラフ作成を行うには JFreeChart ライブラリ (**jfreechart.jar** と **jcommon.jar**) を **WEB-INF/lib** ディレクトリに追加する必要があります。 現在、 円グラフ、 棒グラフ、 折れ線グラフの 3 種類のグラフに対応しています。

<p:chart>	**説明** Seam コンポーネントにより Java ですでに作成されているグラフを表示します。 **Attributes** • **chart** -- 表示されるグラフオブジェクトです。 • **height** -- グラフの高さです。 • **width** -- グラフの幅です。 **使用方法** `<p:chart chart="#{mycomponent.chart}" width="500" height="500" />`
<p:barchart>	**説明** 棒グラフを表示します。 **Attributes** • **borderVisible** – グラフ全体を囲む境界線を表示するかどうかを制御します。 • **borderPaint** – 境界線の色を表示させる場合の色です。 • **borderBackgroundPaint** – グラフのデフォルト背景色です。 • **borderStroke** • **domainAxisLabel** – 領域軸のテキストのラベルです。 • **domainLabelPosition** – 領域軸カテゴリのラベルの角度です。有効な値は **STANDARD**、**UP_45**、**UP_90**、**DOWN_45**、**DOWN_90** です。値は弧度で整数にも負数にもなります。

- **domainAxisPaint** – 領域軸のラベルの色です。
- **domainGridlinesVisible**– 領域軸のグリッド線をグラフに表示させるかどうかを制御します。
- **domainGridlinePaint**– 領域グリッド線を表示させる場合の色です。
- **domainGridlineStroke** – 領域のグリッド線を表示する場合の線のスタイルです。
- **height** – グラフの高さです。
- **width** – グラフの幅です。
- **is3D** – グラフを 2D ではなく 3D で表示させることを示す Boolean 値です。
- **legend** – グラフに説明文を含ませるかどうかを示す Boolean 値です。
- **legendItemPaint**– 説明文内のテキストラベルのデフォルト色です。
- **legendItemBackgoundPaint**– グラフの背景色とは異なる色にする場合の説明文の背景色です。
- **legendOutlinePaint**– 説明文を囲む境界線の色です。
- **orientation** – 図表の向きで、**vertical** (デフォルト) または **horizontal** です。
- **plotBackgroundPaint** – 図表の背景色です。
- **plotBackgroundAlpha** – 図表の背景色のアルファ (透明度) レベルです。 0 (完全に透明) から 1 (完全に不透明) の間の数字にします。
- **plotForegroundAlpha** – 図表のアルファ (透明度) レベルです。 0 (完全に透明) から 1 (完全に不透明) の間の数字にします。
- **plotOutlinePaint** – その範囲のグリッド線を表示させる場合の色です。
- **plotOutlineStroke** – その範囲のグリッド線を表示させる場合の線のスタイルです。
- **rangeAxisLabel** – その範囲の軸のテキストラベルです。
- **rangeAxisPaint** – その範囲の軸レベルの色です。
- **rangeGridlinesVisible** – 範囲軸のグリッド線をグラフに表示させるかどうかを制御します。
- **rangeGridlinePaint** – その範囲のグリッド線を表示させる場合の色です。
- **rangeGridlineStroke** – その範囲のグリッド線を表示させる場合の線スタイルです。
- **title** – グラフのタイトルテキストです。
- **titlePaint** – グラフのタイトルテキストの色です。
- **titleBackgroundPaint** – グラフタイトルを囲む背景色です。
- **width** – グラフの幅です。

使用方法

```
<p:barchart title="Bar Chart" legend="true"
width="500" height="500">
  <p:series key="Last Year">
    <p:data columnKey="Joe" value="100" />
    <p:data columnKey="Bob" value="120" />
  </p:series>
  <p:series key="This Year">
    <p:data columnKey="Joe" value="125" />
    <p:data columnKey="Bob" value="115" />
  </p:series>
</p:barchart>
```

<p:linechart>

説明

折れ線グラフを表示します。

Attributes

- **borderVisible** – グラフ全体を囲む境界線を表示するかどうかを制御します。
- **borderPaint** – 境界線の色を表示させる場合の色です。
- **borderBackgroundPaint** – グラフのデフォルト背景色です。
- **borderStroke** –
- **domainAxisLabel** – 領域軸のテキストのラベルです。
- **domainLabelPosition** – 領域軸カテゴリのラベルの角度です。有効な値は **STANDARD**、**UP_45**、**UP_90**、**DOWN_45**、**DOWN_90** です。値は弧度で整数にも負数にもなります。
- **domainAxisPaint** – 領域軸のラベルの色です。
- **domainGridlinesVisible**– 領域軸のグリッド線をグラフに表示させるかどうかを制御します。
- **domainGridlinePaint**– 領域グリッド線を表示させる場合の色です。
- **domainGridlineStroke** – 領域のグリッド線を表示する場合の線のスタイルです。
- **height** – グラフの高さです。
- **width** – グラフの幅です。
- **is3D** – グラフを 2D ではなく 3D で表示させることを示す Boolean 値です。
- **legend** – グラフに説明文を含ませるかどうかを示す Boolean 値です。
- **legendItemPaint** – 説明文内のテキストラベルのデフォルト色です。
- **legendItemBackgoundPaint** – グラフの背景色とは異なる色にする場合の説明文の背景色です。
- **legendOutlinePaint** – 説明文を囲む境界線の色です。

- **orientation** – 図表の向きで、**vertical** (デフォルト) または **horizontal** です。
- **plotBackgroundPaint** – 図表の背景色です。
- **plotBackgroundAlpha** – 図表の背景のアルファ (透明度) レベルです。0 (完全に透明) から 1 (完全に不透明) の間の数字にします。
- **plotForegroundAlpha** – 図表のアルファ (透明度) レベルです。 0 (完全に透明) から 1 (完全に不透明) の間の数字にします。
- **plotOutlinePaint** – その範囲のグリッド線を表示させる場合の色です。
- **plotOutlineStroke** – その範囲のグリッド線を表示させる場合の線のスタイルです。
- **rangeAxisLabel** – その範囲の軸のテキストラベルです。
- **rangeAxisPaint** – その範囲の軸レベルの色です。
- **rangeGridlinesVisible** – 範囲軸のグリッド線をグラフに表示させるかどうかを制御します。
- **rangeGridlinePaint** – その範囲のグリッド線を表示させる場合の色です。
- **rangeGridlineStroke** – その範囲のグリッド線を表示させる場合の線スタイルです。
- **title** – グラフのタイトルテキストです。
- **titlePaint** – グラフのタイトルテキストの色です。
- **titleBackgroundPaint** – グラフタイトルを囲む背景色です。
- **width** – グラフの幅です。

使用方法

```
<p:linechart title="Line Chart" width="500"
height="500">
  <p:series key="Prices">
    <p:data columnKey="2003" value="7.36" />
    <p:data columnKey="2004" value="11.50" />
    <p:data columnKey="2005" value="34.625" />
    <p:data columnKey="2006" value="76.30" />
    <p:data columnKey="2007" value="85.05" />
  </p:series>
</p:linechart>
```

<p:piechart>

説明

円グラフを表示します。

Attributes

- **title** – グラフのタイトルテキストです。
- **label** – 円グラフの各セクションのデフォルトラベルテキストです。

- **legend** – グラフに説明文を含ませるかどうかを示す Boolean 値です。 デフォルト値は **true** です。
- **is3D** – グラフを 2D ではなく 3D で表示させることを示す Boolean 値です。
- **labelLinkMargin** – ラベルのリンクの余白です。
- **labelLinkPaint** – ラベルのリンク線に使用するペイントです。
- **labelLinkStroke** – ラベルのリンク線に使用する線です。
- **labelLinksVisible** – ラベルのリンクを使用するかどうかを制御するフラグです。
- **labelOutlinePaint** – セクションラベルの輪郭描写に使用するペイントです。
- **labelOutlineStroke** – セクションラベルの輪郭描写に使用する線です。
- **labelShadowPaint** – セクションラベルの影を描写するのに使用するペイントです。
- **labelPaint** – セクションラベルの描写に使用する色です。
- **labelGap** – 図表幅のパーセンテージで表すラベルと図表の間隔です。
- **labelBackgroundPaint** – セクションラベルの背景描写に使用する色です。null にすると背景に色は挿入されません。
- **startAngle** – 1 番目のセクションの開始角度です。
- **circular** – グラフが円形に描写されることを示す Boolean 値です。**false** にするとグラフは楕円形で描写されます。 デフォルトは**true** です。
- **direction** – 円グラフのセクションが描写される方向です。**clockwise** (右回り) か **anticlockwise** (左回り) のいずれかです。 デフォルトは **clockwise** (右回り) です。
- **sectionOutlinePaint** – すべてのセクションの輪郭のペイントです。
- **sectionOutlineStroke** – すべてのセクションの輪郭の線です。
- **sectionOutlinesVisible** – 図表内の各セクションに輪郭を描写するかどうかを示します。
- **baseSectionOutlinePaint** – ベースセクションの輪郭のペイントです。
- **baseSectionPaint** – ベースセクションのペイントです。
- **baseSectionOutlineStroke** – ベースセクションの輪郭の線です。

使用方法

```
<p:piechart title="Pie Chart" circular="false"
            direction="anticlockwise" startAngle="30"
            labelGap="0.1" labelLinkPaint="red">
  <p:series key="Prices">
    <p:data key="2003" columnKey="2003" value="7.36"
```

```
/>
    <p:data key="2004" columnKey="2004" value="11.50"
/>
    <p:data key="2005" columnKey="2005" value="34.625"
/>
    <p:data key="2006" columnKey="2006" value="76.30"
/>
    <p:data key="2007" columnKey="2007" value="85.05"
/>
  </p:series>
</p:piechart>
```

<p:series>

説明

カテゴリデータはシリーズに分割できます。 シリーズタグを使ってデータセットをシリーズで分類し、そのシリーズ全体にスタイリングを適用します。

Attributes

- **key** – シリーズ名です。
- **seriesPaint** – シリーズの各アイテムの色です。
- **seriesOutlinePaint** – シリーズ内の各アイテムの輪郭色です。
- **seriesOutlineStroke** – シリーズ内の各アイテムを描くのに使用する線です。
- **seriesVisible** – シリーズを表示させるかどうかを示す Boolean です。
- **seriesVisibleInLegend** – シリーズを説明文内に表示させるかどうかを表す Boolean です。

使用方法

```
<p:series key="data1">
  <ui:repeat value="#{data.pieData1}" var="item">
    <p:data columnKey="#{item.name}"
            value="#{item.value}" />
  </ui:repeat>
</p:series>
```

<p:data>	**説明**
	
	データタグはグラフ内で表示される各データポイントを表現します。
	
	Attributes
	
	• **key** – データアイテムの名前です。
	• **series** – **<p:series>** の内側に埋め込まれない場合のシリーズ名です。
	• **value** – 数値データ値です。
	• **explodedPercent** – 円グラフの場合、グラフの一片がどのように展開されるかを示します。
	• **sectionOutlinePaint** – 棒グラフの場合のセクション輪郭の色です。
	• **sectionOutlineStroke** – 棒グラフの場合のセクション輪郭の線タイプです。
	• **sectionPaint** – 棒グラフのセクションの色です。
	
	使用方法
	
	`<p:data key="foo" value="20" sectionPaint="#111111" explodedPercent=".2" />`
	`<p:data key="bar" value="30" sectionPaint="#333333" />`
	`<p:data key="baz" value="40" sectionPaint="#555555" sectionOutlineStroke="my-dot-style" />`

<p:color>	**説明**
	
	色コンポーネントは、色埋めされた形を描く場合に参照できる色または階調を宣言します。
	
	Attributes
	
	• **color** – 色の値です。階調色の場合はこれが開始色となります。 色の値については「色の値」を参照してください。
	• **color2** – 階調色の場合はこれが階調の最終となります。
	• **point** – 階調色開始点の座標です。
	• **point2** – 階調色終了点の座標です。
	
	使用方法
	
	`<p:color id="foo" color="#0ff00f"/>`
	`<p:color id="bar" color="#ff00ff" color2="#00ff00" point="50 50" point2="300 300"/>`

<p:stroke>	**説明** グラフ内に線を描くのに使用する線を表します。 **Attributes** • **width** – 線の幅です。 • **cap** – ラインキャップのタイプです。有効な値は **butt**、**round**、**square** です。 • **join** – ラインジョインのタイプです。有効な値は **miter**、**round**、**bevel** です。 • **miterLimit** – マイター接合の場合の接合サイズの制限です。 • **dash** – 線を描くのに使用する点線パターンを設定します。 空白で区切った整数を使って交互に描かれる箇所と描かれない箇所の長さを示します。 • **dashPhase** – 線の開始となる点線パターン内の点を示します。 **使用方法** `<p:stroke id="dot2" width="2" cap="round" join="bevel" dash="2 3" />`

18.3. バーコード

Seam は iText を使って幅広い種類の形式でバーコードを生成することができます。 こうしたバーコードは PDF ドキュメントに埋め込んだり、 Web ページにイメージとして表示させたりできます。 ただし HTML イメージを使用している場合は、バーコードはバーコードテキストを現在表示することができません。

<p:barCode>	**説明** バーコードイメージを表示します。 **Attributes** • **type** – iText によりサポートされているバーコードのタイプです。有効な値は **EAN13**、**EAN8**、**UPCA**、**UPCE**、**SUPP2**、**SUPP5**、**POSTNET**、**PLANET**、**CODE128**、**CODE128_UCC**、**CODE128_RAW**、**CODABAR** です。 • **code** – バーコードでコード化される値です。 • **xpos** – PDF 用です。 ページ上のバーコードの絶対**x** 位置です。 • **ypos** – PDF 用です。 ページ上のバーコードの絶対**y** 位置です。 • **rotDegrees** – PDF 用です。度数で表されるバーコードの回転係数です。 • **barHeight** – バーコードのバーの高さです。 • **minBarWidth** – バーの幅の最小値です。 • **barMultiplier** – 幅広のバーに対するバー乗数、 または**POSTNET** と **PLANET** コードのバー間の距離です。 • **barColor** – バーを描く色です。 • **textColor** – バーコード上のテキストの色です。 • **textSize** –バーコード上のテキストサイズです。 • **altText** – HTML イメージリンクの**alt** テキストです。 **使用方法** `<p:barCode type="code128" barHeight="80" textSize="20" code="(10)45566(17)040301" codeType="code128_ucc" altText="My BarCode" />`

18.4. 入力フォーム

名前が付いたフィールドを持つ複雑な生成済みの PDF がある場合、 この PDF にアプリケーションからの値で埋めてユーザーに表示することが可能です。

<p:form>	**説明** 埋めるフォームのテンプレートを定義します。 **Attributes** • **URL** – テンプレートとして使用する PDF ファイルをポイントする URL です。値にプロトコル (**://**) がない場合は、そのファイルはローカルに読み込まれます。 • **filename** – 生成された PDF ファイルに使用するファイル名です。 • **exportKey** – これを設定すると、生成された PDF ファイルをイベントコンテキスト内の指定されたキーの DocumentData オブジェクトに配置した場合にリダイレクトは発生しなくなります。

<p:field>	**説明** フィールド名をその値につなげます。 **Attributes** • **name** – フィールド名です。 • **value** – フィールド値です。 • **readOnly** – フィールドが読み取り専用かどうかを指定します。デフォルトは **true** です。

```
<p:form
   xmlns:p="http://jboss.com/products/seam/pdf"
   URL="http://localhost/Concept/form.pdf">
  <p:field name="person.name" value="Me, myself and I"/>
</p:form>
```

18.5. SWING/AWT コンポーネントをレンダリングする

Seam は Swing コンポーネントを PDF イメージにレンダリングするサポートを実験的に提供しています。Swing の外観サポート、具体的にはネイティブウィジェットを使用するものは正しくレンダリングを行いません。

<p:swing>	**説明** Swing コンポーネントを PDF ドキュメントにレンダリングします。 **Attributes** • **width** – レンダリングされるコンポーネントの幅です。 • **height** – レンダリングされるコンポーネントの高さです。 • **component** – Swing または AWT コンポーネントが値となる式です。 **使用方法** `<p:swing width="310" height="120" component="#{aButton}" />`

18.6. ITEXT の設定

ドキュメントを生成すること自体には追加で設定を行う必要はありませんが、若干の設定を加えることでアプリケーションはより使いやすくなります。

デフォルト実装では汎用 URL **/seam-doc.seam** から PDF ドキュメントを提供します。 たとえば、 多くのユーザーは **/myDocument.pdf** などのように実際の PDF 名と拡張子を含んでいる URL が表示されるのを好みます。完全な名前のファイルを提供するには、**DocumentStoreServlet** に各ドキュメントタイプのマッピングが含まれる必要があります。

```
<servlet>
  <servlet-name>Document Store Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.document.DocumentStoreServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Document Store Servlet</servlet-name>
  <url-pattern>DOCUMENT_TYPE</url-pattern>
</servlet-mapping>
```

DOCUMENT_TYPE には以下の値を指定できます。

- ***.pdf**
- ***.xls**
- ***.csv**

複数のドキュメントタイプを含めるには、必要な各ドキュメントタイプに対して **<servlet-name>** および **<url-pattern>** サブ要素とともに **<servlet-mapping>** 要素を追加します。

use-extensions オプションは、生成されるドキュメントタイプに正しいファイル名拡張子を付けて URL を生成するよう DocumentStore コンポーネントに指示します。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:document="http://jboss.com/products/seam/document"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
            xsi:schemaLocation="
                http://jboss.com/products/seam/document
                http://jboss.com/products/seam/document-2.2.xsd
                http://jboss.com/products/seam/components
                http://jboss.com/products/seam/components-2.2.xsd">
  <document:document-store use-extensions="true"/>
</components>
```

DocumentStore は対話スコープにドキュメントを保持するため、対話が終了するとドキュメントの有効期限が切れます。この時点でドキュメントへの参照は無効になります。**documentStore** の **error-page** プロパティを編集することで、ドキュメントが存在しない場合にデフォルトのビューが表示されるよう指定します。

```
<document:document-store use-extensions="true"
          error-page="/documentMissing.seam" />
```

18.7. その他のドキュメント

iText に関する詳細は次を参照してください。

- iText ホームページ
- iText の実行可能なデモ

第19章 MICROSOFT® EXCEL® 表計算アプリケーション

Seam では JExcelAPI ライブラリを通じて、Microsoft® Excel® のスプレッドシートを作成することができます。 作成されたドキュメントは **Microsoft Excel** の 95、 97、 2000、 XP、 2003 の各バージョンと互換性があります。 現時点では、 使用できるライブラリの機能は限られています。 機能および制約についての詳細は JExcelAPI のドキュメントを参照してください。

19.1. MICROSOFT EXCEL のサポート

使用しているアプリケーションに **Microsoft Excel** を含ませるには **WEB-INF/lib** ディレクトリに **jboss-seam-excel.jar** と **jxl.jar** を含ませる必要があります。 **jboss-seam-excel.jar** にはドキュメントのレンダリング用ビューの構成に使用する **Microsoft Excel** JSF のコントロールとレンダリングされたドキュメントをユーザーに提供する DocumentStore コンポーネントが含まれます。 また、 **web.xml** ファイルの DocumentStore サーブレットを設定する必要があります。 **Microsoft Excel** Seam モジュールは **seam-ui** パッケージを必要とし、 また Facelets がビューテクノロジーとして使用されていなければなりません。

examples/excel プロジェクトでは実際の **Microsoft Excel** サポートのサンプルをご覧頂けます。 これはそのサポートによる表示された機能や正しいデプロイメントパッケージングなどを示しています。

他の種類の **Microsoft Excel** スプレッドシートをサポートするためのモジュールのカスタマイズも容易に行うことができます。 **ExcelWorkbook** インターフェースを実装してから次を **components.xml** に登録します。

```
<excel:excelFactory>
  <property name="implementations">
    <key>myExcelExporter</key>
    <value>my.excel.exporter.ExcelExport</value>
  </property>
</excel:excelFactory>
```

以下のようにコンポーネントタグに **Microsoft Excel** 名前空間を登録します。

```
xmlns:excel="http://jboss.com/products/seam/excel"
```

次に、 好みのエクスポーターを使用する場合は **myExcelExporter** に UIWorkbook タイプを設定します。 デフォルトは **jxl** ですが、 **csv** タイプを使って CSV を使用することもできます。

ドキュメントを **.xls** 拡張子で使用するようドキュメントサーブレットを設定する方法については「iText の設定」 を参照してください。

Microsoft® **Internet Explorer®** で生成されたファイル、 特に HTTPS で生成されたファイルへのアクセスに問題が発生する場合は **web.xml** またはブラウザのセキュリティ制限が厳しすぎていないか確認してください (http://www.nwnetworks.com/iezones.htm/ を参照)。

19.2. 簡単なワークブックの作成

ワークシートのサポートは **<h:dataTable>** と同様に使用され、 **List**、 **Set**、 **Map**、 **Array**、 **DataModel** にバインドさせることができます。

```
<e:workbook xmlns:e="http://jboss.com/products/seam/excel">
  <e:worksheet>
```

```
    <e:cell column="0" row="0" value="Hello world!"/>
  </e:worksheet>
</e:workbook>
```

以下に一般的な使用例を示します。

```
<e:workbook xmlns:e="http://jboss.com/products/seam/excel">
  <e:worksheet value="#{data}" var="item">
    <e:column>
      <e:cell value="#{item.value}"/>
    </e:column>
  </e:worksheet>
</e:workbook>
```

トップレベルの **workbook** エレメントはコンテナとして動作するため属性はありません。 子エレメントの **worksheet** には、 データへの EL バインディングとなる **value="#{data}"** と、 現在のアイテム名となる **var="item"** の 2 種類の属性があります。 worksheet には **column** が 1 つだけあり、 この中に **cell** があります。 現在反復しているアイテム内のデータへの最終バインディングとなります。

これでデータをスプレッドシートにバインドすることができます。

19.3. WORKBOOK

workbook は worksheet や stylesheet link のトップレベルの親となります。

<e:workbook>	**Attributes** • **type** – エクスポートモデルを定義します。 値は文字列で、**jxl** または **csv** のいずれかになります。 デフォルトは**jxl** です。 • **templateURI** – ワークブックの基本を形成するテンプレートになります。 値は文字列です (URI)。 • **arrayGrowSize** – ワークブックのデータストレージスペースを増加させるバイト単位のメモリ量です。 プロセスが Web アプリケーションサーバー内の小さなワークブックを数多く読み込む場合はデフォルトサイズを小さくする必要があるかもしれません。 デフォルト値は 1 MB です。 • **autoFilterDisabled** – 自動フィルタ機能を無効にするかどうかを決定する Boolean 値です。 • **cellValidationDisabled** – セル検証を無視するかどうかを決める Boolean 値です。 • **characterSet** – スプレッドシートの読み込みに使用する文字セットです。 書き込まれているスプレッドシートには影響ありません。 値は文字列になります (文字セットエンコーディング)。 • **drawingsDisabled** – 描画を無効にするかどうかを決める Boolean 値です。 • **excelDisplayLanguage** – 生成されたファイルが表示される言語です。 値は文字列になります (2 文字の ISO 3166 国名コード)。 • **excelRegionalSettings** – 生成されたファイル用の地域設定です。 値は文字列です (2 文字の ISO 3166 国名コード)。

- **formulaAdjust** – フォーミュラを調整するかどうかを決める Boolean 値です。
- **gcDisabled** – ガーベッジコレクションを無効にするかどうかを決める Boolean 値です。
- **ignoreBlanks** – 空白を無視するかどうかを決める Boolean 値です。
- **initialFileSize** – ワークシート読み込み時にワークブックのデータストレージに割り振るバイト単位の初期メモリ量です。 プロセスが Web アプリケーションサーバー内の小さなワークブックを数多く読み込む場合はデフォルトのサイズを小さくする必要があるかもしれません。 デフォルト値は 5 MB です。
- **locale** – スプレッドシートの生成に JExcelAPI が使用するロケールです。 この値は生成されたファイルの言語や地域には影響ありません。 値は文字列です。
- **mergedCellCheckingDisabled** – マージしたセルのチェック機能を無効にするかどうかを決める Boolean 値です。
- **namesDisabled** – 名前処理を無効にするかどうかを決める Boolean 値です。
- **propertySets** – プロパティセット (マクロなど) がワークブックでコピーされるかどうかを決める Boolean 値です。 この機能を有効にすると JXL プロセスのメモリ使用量が増加します。
- **rationalization** – シート書き込みの前にセルのフォーマットを合理化するかどうかを決める Boolean 値です。 デフォルトは **true** です。
- **supressWarnings** – 警告を抑制するかどうかを決める Boolean 値です。 使用するロッガーのタイプにより警告の動作を JVM 全体にセットすることになります。
- **temporaryFileDuringWriteDirectory** – 一時ファイルの目的ディレクトリを含む文字列値です。 **useTemporaryFileDuringWrite** と併用します。 **NULL** に設定するとデフォルトのテンポラリディレクトリが使用されます。
- **useTemporaryFileDuringWrite** – ワークブック生成中に一時ファイルを使用するかどうかを決める Boolean 値です。設定しないとワークブックは完全にメモリ内で生成されます。 このフラグを設定するとメモリ使用とパフォーマンス間のトレードオフの評価が行われることになります。
- **workbookProtected** – ワークブックを保護するかどうかを決める Boolean 値です。
- **filename** – ダウンロードのファイル名として使用される文字列の値です。 DocumentServelet を何らかのパターンにマッピングする場合はそのファイルの拡張子が一致しなければなりません。
- **exportKey** – イベントスコープのデータを DocumentData オブジェクトに格納するキーです。 使用するとリダイレクトが起こらなくなります。

子となるエレメント

- **<e:link/>** – ゼロまたはそれ以上のスタイルシートのリンクです (「スタイルシートへのリンク」を参照)。

	• **<e:worksheet/>** – ゼロまたはそれ以上のワークシートです (「worksheet」 を参照)。 ファセット • **none**

```
<e:workbook>
  <e:worksheet>
    <e:cell value="Hello World" row="0" column="0"/>
  </e:worksheet>
<e:workbook>
```

上記は 1 つの worksheet とセル A1 に「Hello World」を持つ workbook を定義しています。

19.4. WORKSHEET

worksheet は workbook の子であり、 column とワークシートコマンドの親です。明示的に配置した cell、formula、image や hyperlink なども含むことができ、また workbook を構成するページになります。

<e:worksheet>	• **value** – バッキングデータに対する EL 式の文字列です。 この式のターゲットに Iterable がないかどうかを検査します。 ターゲットが MAP の場合、 その反復は Map.Entry entrySet() 全体に行われるため、 .key または .value を参照内のターゲットに対して使用します。 • **var** – セル値の属性で参照される現在の行の iterator 変数名です。 値は文字列になります。 • **name** – ワークシートの名前です。 値は文字列になります。 デフォルトでは **Sheet<replaceable>#</replaceable>** に設定されます。 # はワークシートのインデックスです。 特定のワークシートの名前が存在する場合はそのシートが選択されます。 これを使い同じ名前で各ワークシートを定義することで複数のデータセットを単一のワークシートにマージすることができます。 **startRow** と **startCol** を使ってこれらが同じ空間を埋めないようにします。 • **startRow** – データの開始行を定義する数値になります。 左上隅以外の場所からデータの位置付けを行うのに使用されます。 特に単一のワークシートに複数のデータセットを使用する場合に便利です。 デフォルト値は **0** です。 • **startColumn** – データの開始列を定義する数値です。 左上隅以外の場所からデータの位置付けを行うのに使用されます。 単一のワークシートに複数のデータセットを使用する場合に特に便利です。 デフォルト値は **0** です。 • **automaticFormulaCalculation** – フォーミュラを自動的に計算させるかどうかを決める Boolean 値です。 • **bottomMargin** – 下余白をインチ単位で決める数値です。 • **copies** – コピー数を決める数値です。 • **defaultColumnWidth** – デフォルトの列幅を決める数値です。 単位は文字数で値は 256 倍です。

- **defaultRowHeight** – デフォルトの行の高さを決める数値です。 単位はポイントで値は 1/20 です。
- **displayZeroValues** – ゼロの値を表示するかどうかを決める Boolean 値です。
- **fitHeight** – シートが印刷される際の縦方向のページ数を決める数値です。
- **fitToPages** – 印刷をページサイズにあわせるかどうかを決める Boolean 値です。
- **fitWidth** – シートが印刷される際の横方向のページ数を決める数値です。
- **footerMargin** – ページフッターの余白をインチ単位で決める数値です。
- **headerMargin** – ページヘッダーの余白をインチ単位で決める数値です。
- **hidden** – ワークシートを非表示にするかどうかを決める Boolean です。
- **horizontalCentre** – ワークシートを水平向きの中央に配置するかどうかを決める Boolean です。
- **horizontalFreeze** – ペインを水平に固定させる列を決める数値です。
- **horizontalPrintResolution** – 水平印字解像度を決める数値です。
- **leftMargin** – 左余白をインチ単位で決める数値です。
- **normalMagnification** – 通常倍率をパーセンテージで決める数値です。これはズーム係数やスケール係数ではありません。
- **orientation** – このシートを印刷する場合の用紙の向きを決める文字列値です。 **landscape** または **portrait** のいずれかです。
- **pageBreakPreviewMagnification** – 改ページプレビューの倍率をパーセンテージで決める数値です。
- **pageBreakPreviewMode** – ページをプレビューモードで表示させるかどうかを決める Boolean です。
- **pageStart** – 印刷を開始するページのページ番号を決める数値です。
- **paperSize** – 印刷時に使用する用紙の大きさを決める文字列値です。 値には **a4**、 **a3**、 **letter**、 **legal** が使用可能です。 用紙の大きさの詳細については jxl.format.PaperSize をご覧ください。
- **password** – このシートのパスワードを決める文字列値です。
- **passwordHash** – パスワードハッシュを決める文字列値です。 シートのコピー時のみ使用されます。
- **printGridLines** – 罫線を印刷するかどうかを決める Boolean です。
- **printHeaders** – ヘッダーを印刷するかどうかを決める Boolean です。

- **sheetProtected** – シートを読み取り専用にするかどうかを決める Boolean です。
- **recalculateFormulasBeforeSave** – シートの保護時にフォーミュラを再計算するかどうかを決める Boolean です。 デフォルト値は **false** です。
- **rightMargin** – 右余白をインチ単位で決める数値です。
- **scaleFactor** – このシートの印刷時に使用するスケール係数 (パーセンテージ) を決める数値です。
- **selected** – ワークブックを開いたときに自動的にこのシートが選択されるかどうかを決める Boolean 値です。
- **showGridLines** – 罫線を表示するかどうかを決める Boolean です。
- **topMargin** – 上余白をインチ単位で決める数値です。
- **verticalCentre** – ワークシートを垂直向きの中央に配置するかどうかを決める Boolean です。
- **verticalFreeze** – ペインを垂直に固定させる行を決める数値です。
- **verticalPrintResolution** – 垂直印字解像度を決める数値です。
- **zoomFactor** – ズーム係数を決める数値です。これは画面上のビューに関連するため、スケール係数と混同しないようにしてください。

子となるエレメント

- **<e:printArea/>** – ゼロまたはそれ以上の印刷領域の定義です (「print area とタイトル」を参照)。
- **<e:printTitle/>** – ゼロまたはそれ以上の印刷タイトルの定義です (「print area とタイトル」を参照)。
- **<e:headerFooter/>** – ゼロまたはそれ以上のヘッダーとフッターの定義です (「header と footer」を参照)。
- 0 またはそれ以上のワークシートコマンドです (「ワークシートコマンド」を参照)。

ファセット

- **header** – 列のヘッダー上部 (あれば)、 データブロックの冒頭に配置する内容です。
- **footer** – 列のフッター下部 (あれば)、 データブロックの末尾に配置する内容です。

```
<e:workbook>
  <e:worksheet name="foo" startColumn="1" startRow="1">
    <e:column value="#{personList}" var="person">
      <f:facet name="header">
        <e:cell value="Last name"/>
      </f:facet>
      <e:cell value="#{person.lastName}"/>
    </e:column>
```

```
  </e:worksheet>
<e:workbook>
```

これは B2 セルから表示が開始される「foo」という名前の worksheet を定義します。

19.5. COLUMN

column は worksheet の子であり、 cell、 image、 formula、 hyperlink の親です。worksheet のデータ列挙を制御します。 書式については「列の設定」 を参照してください。

<e:column>	**Attributes** • **none** **子となるエレメント** • **<e:cell/>** – ゼロまたはそれ以上のセルです (「cell」 を参照)。 • **<e:formula/>** – ゼロまたはそれ以上のフォーミュラです (「formula」 を参照)。 • **<e:image/>** – ゼロまたはそれ以上のイメージです (「image」 を参照)。 • **<e:hyperLink/>** – ゼロまたはそれ以上のハイパーリンクです (「hyperlink」 を参照)。 **ファセット** • **header** – **<e:cell>**、 **<e:formula>**、 **<e:image>** または **<e:hyperLink>** のうちいずれかひとつを含み、 列のヘッダーとして使用されます。 • **footer** – **<e:cell>**、 **<e:formula>**、 **<e:image>** または **<e:hyperLink>**、 のうちいずれかひとつを含み、 列のフッターとして使用されます。

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <f:facet name="header">
        <e:cell value="Last name"/>
      </f:facet>
      <e:cell value="#{person.lastName}"/>
    </e:column>
  </e:worksheet>
<e:workbook>
```

これは、ヘッダーと出力の列挙からなる列を定義します。

19.6. CELL

cell は column 内 (値の列挙) または worksheet の内側 (**column** 属性と **row** 属性を用いた直接的な配置) にネストされ、 通常データテーブルの **var** 属性を伴う **EL** 式でその値の出力を行います。 「セルの設定」 を参照してください。

<e:cell>	**Attributes** • **column** – セルが属している列を示す数値です。 デフォルトは内部カウンタです。 値は 0 ベースとなるので注意してください。 • **row** – セルを配置する場所の行を示す数値です。 デフォルトは内部カウンタです。 値は 0 ベースとなるので注意してください。 • **value** – 表示値を定義する文字列です。 通常、 含んでいるデータテーブルの var 属性を参照する **EL** 式になります。 • **comment** – セルに付けられたコメントを定義する文字列値です。 • **commentHeight** – ピクセル単位によるコメントの高さです。 • **commentWidth** – ピクセル単位によるコメントの幅です。 **子となるエレメント** • 0 またはそれ以上の検証条件です (「validation」 を参照)。 **ファセット** • **none**

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <f:facet name="header">
        <e:cell value="Last name"/>
      </f:facet>
      <e:cell value="#{person.lastName}"/>
    </e:column>
  </e:worksheet>
</e:workbook>
```

これは、ヘッダーと出力の列挙からなる列を定義します。

19.6.1. validation

Validation は cell または formula の内側にネストされます。 セルのデータに制約を加えます。

<e:numericValidation>	**Attributes** • **value** – 検証の制限 (適用できる場合は下限) を示す数値です。 • **value2** – 検証の上限 (適用できる場合) を示す数値です。 • **condition** – 検証の条件を決める文字列値です。 ◦ **equal** – セルの値が value 属性で定義した値と一致する必要があります。 ◦ **greater_equal** – セルの値が value 属性で定義した値より大きいまたは同等である必要があります。 ◦ **less_equal** – セルの値が value 属性で定義した値より小さいまたは同等である必要があります。 ◦ **less_than** – セルの値が value 属性で定義した値より小さい値である必要があります。 ◦ **not_equal** – セルの値が value 属性で定義した値と一致しない必要があります。 ◦ **between** – セルの値が value と value2 の属性で定義した値の間である必要があります。 ◦ **not_between** – セルの値が value と value2 の属性で定義した値の間の値にならない必要があります。 **子となるエレメント** • **none** **ファセット** • **none**

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <e:cell value="#{person.age">
        <e:numericValidation condition="between" value="4" value2="18"/>
      </e:cell>
    </e:column>
  </e:worksheet>
</e:workbook>
```

これは、値が 4 と 18 の間の値でなければならないという数値に関する検証条件をセルに付加します。

<e:rangeValidation>	**Attributes** • **startColumn** – 検証対象となる最初の列を示す数値です。 • **startRow** – 検証対象となる最初の行を示す数値です。 • **endColumn** – 検証対象となる最後の列を示す数値です。 • **endRow** – 検証対象となる最後の行を示す数値です。 子となるエレメント • **none** ファセット • **none**

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <e:cell value="#{person.position">
        <e:rangeValidation startColumn="0" startRow="0" endColumn="0"
           endRow="10"/>
      </e:cell>
    </e:column>
  </e:worksheet>
</e:workbook>
```

これは、 値が A1:A10 の範囲にある値のどれかでなければならないという検証条件をセルに付加します。

<e:listValidation>	**Attributes** • **none** 子となるエレメント • **0 アイテム以上の listvalidation アイテム** ファセット • **none**

e:listValidation は複数の e:listValidationItem タグを保持するための単なるコンテナです。

<e:listValidationItem>	**Attributes** • **value** – 検証対象となる値です。 子となるエレメント • **none** ファセット • **none**

```
<e:workbook>
  <e:worksheet>
    <e:column value="#{personList}" var="person">
      <e:cell value="#{person.position">
        <e:listValidation>
          <e:listValidationItem value="manager"/>
          <e:listValidationItem value="employee"/>
        </e:listValidation>
      </e:cell>
    </e:column>
  </e:worksheet>
</e:workbook>
```

これは、 値が「manager」もしくは「employee」でなければならないという検証条件をセルに付加します。

19.6.2. 書式マスク

書式マスクは cell または formula 内の **mask** 属性で定義されます。 書式マスクは数値と日付の 2 種類あります。

19.6.2.1. 数値マスク

書式マスクがあった場合、 例えば **format1**、 **accounting_float** など、 それが内部書式に従うかどうかがチェックされます (jxl.write.NumberFormats を参照してください)。

マスクが内部リストの一部ではない場合はカスタムマスクとして扱われ (たとえば **0.00** など)、 自動的に一番近いものに変換されます （ java.text.DecimalFormat を参照)。

19.6.2.2. 日付マスク

書式マスクがあった場合、たとえば **format1**、 **format2** など、 それが内部書式に従うかどうかがチェックされます (jxl.write.DecimalFormats を参照)。

マスクが内部リストの一部ではない場合はカスタムマスクとして扱われ (**dd.MM.yyyy** など)、 自動的に一番近いものに変換されます (java.text.DateFormat を参照)。

19.7. FORMULA

formula は column の中 (値の列挙) または worksheet の内側 (**column** 属性と **row** 属性を用いた直接的な配置) にネストされ、 ある範囲のセル値の計算や関数の適用を行います。 formula は本質的にセルのため、 使用可能な属性については「cell」 を参照してください。 テンプレートを適用することができ、 通常のセルと同様に独自のフォント定義などを持たせることができます。

セルの formula は **value** 属性内に通常の **Microsoft Excel** の表記法で配置されます。 クロスシートの formula を行う場合、 worksheet に対してフォーミュラを参照する前にその worksheet が存在していなければなりません。 値は文字列になります。

```
<e:workbook>
  <e:worksheet name="fooSheet">
    <e:cell column="0" row="0" value="1"/>
  </e:worksheet>
  <e:worksheet name="barSheet">
  <e:cell column="0" row="0" value="2"/>
    <e:formula column="0" row="1" value="fooSheet!A1+barSheet1!A1">
      <e:font fontSize="12"/>
    </e:formula>
  </e:worksheet>
</e:workbook>
```

これは、FooSheet および BarSheet の worksheet のそれぞれの A1 セルを加算する B2 のフォーミュラを定義します。

19.8. IMAGE

image は、 column 内 (値の列挙) または worksheet の内側 (**startColumn/startRow** 属性と**rowSpan/columnSpan** 属性を用いた直接的な配置) にネストされます。 span タグはオプションのため省略するとその画像はリサイズされずに挿入されます。

<e:image>	**Attributes** • **startColumn** – イメージの開始列を示す数値です。 デフォルトは内部カウンタです。 数値は 0 ベースです。 • **startRow** – イメージの開始行を示す数値です。 デフォルトは内部カウンタです。 数値は 0 ベースです。 • **columnSpan** – イメージが占める列の長さを示す浮動値です。 デフォルトではイメージのデフォルトの幅を使用します。 • **rowSpan** – イメージが占める行の長さを示す浮動値です。 デフォルトではイメージのデフォルトの高さを使用します。 • **URI** – イメージに対する URI を示す文字列です。 **子となるエレメント** • **none** **ファセット** • **none**

```
<e:workbook>
  <e:worksheet>
    <e:image startRow="0" startColumn="0" rowSpan="4" columnSpan="4"
       URI="http://foo.org/logo.jpg"/>
  </e:worksheet>
</e:workbook>
```

これは、任意のデータに基づき画像を A1:E5 の領域に定義します。

19.9. HYPERLINK

Hyperlink は column 内 (値の列挙) または worksheet の内側 (**startColumn/startRow** 属性と **endColumn/endRow** 属性を用いた直接的な配置) にネストされます。 URIにリンクのナビゲーションを追加します。

<e:hyperlink>	**Attributes** • **startColumn** – ハイパーリンクの開始列を示す数値です。 デフォルトは内部カウンタです。 数値は 0 ベースです。 • **startRow** – ハイパーリンクの開始行を示す数値です。 デフォルトは内部カウンタです。 数値は 0 ベースです。 • **endColumn** – ハイパーリンク終了列を示す数値です。 デフォルトは内部カウンタです。 数値は 0 ベースです。 • **endRow** – ハイパーリンクの終了行を示す数値です。 デフォルトは内部カウンタです。 数値は 0 ベースです。 • **URL** – リンクへの URL を示す文字列値です。 • **description** – リンクを表している文字列値です。 **子となるエレメント** • **none** **ファセット** • **none**

```
<e:workbook>
  <e:worksheet>
    <e:hyperLink startRow="0" startColumn="0" endRow="4" endColumn="4"
       URL="http://seamframework.org" description="The Seam Framework"/>
  </e:worksheet>
</e:workbook>
```

これは、Seam Framework を指すメッセージ付きハイパーリンクを A1:E5 の領域に定義します。

19.10. HEADER と FOOTER

header と footer は worksheet の子であり、 コマンドとして解析される文字列を持つファセットを含

みます。

<e:header>	**Attributes** • **none** **子となるエレメント** • **none** **ファセット** • **left** – ヘッダー左部分の内容です。 • **center** – ヘッダー中央部分の内容です。 • **right** – ヘッダー右部分の内容です。

<e:footer>	**Attributes** • **none** **子となるエレメント** • **none** **ファセット** • **left** – フッター左部分の内容です。 • **center** – フッター中央部分の内容です。 • **right** – フッター右部分の内容です。

ファセットは文字列値を含むため、 以下のように # で区切られた各種のコマンドを含めることができます。

#date#	現在の日付を挿入します。
#page_number#	現在のページ番号を挿入します。
#time#	現在の時刻を挿入します。
#total_pages#	総ページ数を挿入します。
#worksheet_name#	ワークシートの名前を挿入します。
#workbook_name#	ワークシートの名前を挿入します。
#bold#	強調文字に切り替えます。 次の #bold# まで有効となります。
#italics#	斜体文字に切り替えます。 次の #italic# まで有効となります。

#underline#	下線を引きます。 次の #underline# まで有効となります。
#double_underline#	二重下線を引きます。 次の #double_underline# まで有効となります。
#outline#	アウトラインフォントに切り替えます。 次の #outline# まで有効となります。
#shadow#	影付き文字に切り替えます。 次の #shadow# まで有効となります。
#strikethrough#	取り消し線を引きます。 次の #strikethrough# まで有効となります。
#subscript#	下付き文字に切り替えます。 次の #subscript# まで有効となります。
#superscript#	上付き文字に切り替えます。 次の #superscript# まで有効となります。
#font_name#	フォント名を設定します。 フォントに Verdana を設定する場合は **#font_name=Verdana#** のようにします。
#font_size#	フォントサイズを設定します。フォントサイズを 12 に設定する場合は **#font_size=12#** のように設定します。

```
<e:workbook>
  <e:worksheet>
    <e:header>
      <f:facet name="left">
        This document was made on #date# and has #total_pages# pages.
      </f:facet>
      <f:facet name="right"> #time# </f:facet>
    </e:header>
  <e:worksheet>
</e:workbook>
```

19.11. PRINT AREA とタイトル

print area と title は worksheet や worksheet template の子となり、 印刷範囲および印刷タイトルを設定します。

<e:printArea>	**Attributes** • **firstColumn** – 領域の左上隅となる列を示す数値です。 値は 0 ベースとなります。 • **firstRow** – 領域の左上隅となる行を示す数値です。 値は 0 ベースとなります。 • **lastColumn** – 領域の右下隅となる列を示す数値です。 値は 0 ベースとなります。 • **lastRow** – 領域の右下隅となる行を示す数値です。 値は 0 ベースとなります。 **子となるエレメント** • **none** **ファセット** • **none**

```
<e:workbook>
  <e:worksheet>
    <e:printTitles firstRow="0" firstColumn="0" lastRow="0"
       lastColumn="9"/>
    <e:printArea firstRow="1" firstColumn="0" lastRow="9"
lastColumn="9"/>
  </e:worksheet>
</e:workbook>
```

これは A1:A10 の領域を印刷タイトルとし B2:J10 の領域を印刷範囲とします。

19.12. ワークシートコマンド

ワークシートコマンドは workbook の子となり、通常 1 回だけ実行されます。

19.12.1. グループ化

列と行のグループ化を行います。

<e:groupRows>	**Attributes** • **startRow** – グループ化を開始する行を示す数値です。 値は 0 ベースです。 • **endRow** – グループ化を終了する行を示す数値です。 値は 0 ベースです。 • **collapse** – グループ化を最初に折り畳んでおくかどうかを示す Boolean です。 **子となるエレメント** • **none** **ファセット** • **none**

<e:groupColumns>	**Attributes** • **startColumn** – グループ化を開始する列を示す数値です。 値は 0 ベースです。 • **endColumn** – グループ化を終了する列を示す数値です。 値は 0 ベースです。 • **collapse** – グループ化を最初に折り畳んでおくかどうかを示す Boolean です。 **子となるエレメント** • **none** **ファセット** • **none**

```
<e:workbook>
  <e:worksheet>
    <e:groupRows startRow="4" endRow="9" collapse="true"/>
    <e:groupColumns startColumn="0" endColumn="9" collapse="false"/>
  </e:worksheet>
</e:workbook>
```

5 行目 から 10 行目までと 5 列目から 10 列目までをグループ化することで、 行は最初は折り畳まれます (ただし、列は折り畳みません)。

19.12.2. 改ページ

改ページを行います。

<e:rowPageBreak>	**Attributes** • **row** – 改ページを行う行を示す数値です。 値は 0 ベースになります。 **子となるエレメント** • **none** **ファセット** • **none**

```
<e:workbook>
  <e:worksheet>
    <e:rowPageBreak row="4"/>
  </e:worksheet>
</e:workbook>
```

これで、5行目で改ページを行います。

19.12.3. 結合

セルを結合します。

<e:mergeCells>	**Attributes** • **startRow** – 結合を開始する行を示す数値です。 値は 0 ベースになります。 • **startColumn** – 結合を開始する列を示す数値です。 値は 0 ベースです。 • **endRow** – 結合を終了する行を示す数値です。 値は 0 ベースになります。 • **endColumn** – 結合を終了する列を示す数値です。 値は 0 ベースになります。 **子となるエレメント** • **none** **ファセット** • **none**

```
<e:workbook>
  <e:worksheet>
    <e:mergeCells startRow="0" startColumn="0" endRow="9" endColumn="9"/>
  </e:worksheet>
</e:workbook>
```

これは、 A1:J10 の範囲のセルを結合します。

19.13. データテーブルエクスポータ

専用の XHTML 文書を作成せず既存の JSF データテーブルをエクスポートしたい場合は、**org.jboss.seam.excel.excelExporter.export** コンポーネントを実行し、 データテーブルの ID を Seam の EL パラメータとして渡すことができます。 たとえば、 次のようなデータテーブルがあるとします。

```
<h:form id="theForm">
  <h:dataTable id="theDataTable" value="#{personList.personList}"
     var="person">
    ...
  </h:dataTable>
</h:form>
```

これを **Microsoft Excel** のスプレッドシートに表示させたい場合、 以下をフォームに配置します。

```
<h:commandLink value="Export"
   action="#{excelExporter.export('theForm:theDataTable')}" />
```

また、 ボタンや s:link、 その他にも好きなメソッドを付けてエクスポータを実行することができます。

フォーマット処理については「フォントとレイアウト」を参照してください。

19.14. フォントとレイアウト

出力の外観は CSS スタイルとタグの属性の組み合わせで制御されます。 CSS スタイル属性は親から子へ流れるため、 1 つのタグを使ってそのタグに定義されたすべての属性を **styleClass** と **style** のシートで適用することができます。

空白やセミコロンなど特殊な文字を使用するフォーマットマスクまたはフォントがある場合、**xls-format-mask:'$;$'** などの **''** 文字で CSS 文字列をエスケープすることができます。

19.14.1. スタイルシートへのリンク

外部スタイルシートは **e:link** タグによって参照されます。 **e:link** タグはあたかも **workbook** タグの子であるかのように文書内に配置されます。

<e:link>	**Attributes** • **URL** – スタイルシートの URL です。 **子となるエレメント** • **none** **ファセット** • **none**

```
<e:workbook>
```

```
  <e:link URL="/css/excel.css"/>
</e:workbook>
```

/css/excel.css で示されるスタイルシートを参照します。

19.14.2. フォント

以下の XLS-CSS 属性のグループはフォントとその属性を定義します。

xls-font-family	フォント名です。 ここに入力したフォントがご使用のシステムによってサポートされていることを確認してください。
xls-font-size	フォントサイズを示す数値です。
xls-font-color	フォントの色です (jxl.format.Colour を参照)。
xls-font-bold	フォントを太字にするかどうか決める Boolean です。 有効な値は **true** と **false** です。
xls-font-italic	フォントを斜体にするかどうかを決める Boolean です。 有効な値は **true** と **false** です。
xls-font-script-style	フォントの上付きや下付きを設定します (jxl.format.ScriptStyle を参照)。
xls-font-underline-style	フォントの下線を設定します (jxl.format.UnderlineStyle を参照)。
xls-font-struck-out	フォントに取り消し線を引くかどうかを決める Boolean です。 有効な値は **true** と **false** です。
xls-font	フォント関連の全ての値の設定を略記で行います。 フォント名を最後にします (フォント名に空白があるフォントを使用する場合は **'Times New Roman'** のように一重引用符でそのフォントを囲みます)。ここでは斜体、太字、 取り消し線のテキストは **italic**、 **bold**、 **struckout** で定義します。 例えば、 **style="xls-font: red bold italic 22 Verdana"**

19.14.3. ボーダー

以下の XLS-CSS 属性のグループはセルのボーダーを定義します。

xls-border-left-color	セルの左端ボーダーの色です (jxl.format.Colour を参照)。
xls-border-left-line-style	セルの左端ボーダーの線スタイルです (jxl.format.LineStyle を参照)。
xls-border-left	セルの左端ボーダーの色と線スタイルの設定を略記で行います。 例えば **style="xls-border-left: thick red"** のようになります。
xls-border-top-color	セルの上端ボーダーの色です (jxl.format.Colour を参照)。

xls-border-top-line-style	セルの上端ボーダーの線スタイルです (jxl.format.LineStyle を参照)。
xls-border-top	セルの上端ボーダーの色と線スタイルの設定を略記で行います。 例えば **`style="xls-border-top: red thick"`** のようになります。
xls-border-right-color	セルの右端ボーダーの色です (jxl.format.Colour を参照)。
xls-border-right-line-style	セルの右端ボーダーの線スタイルです (jxl.format.LineStyle を参照)。
xls-border-right	セルの右端ボーダーの色と線スタイルの設定を略記で行います。 例えば **`style="xls-border-right: thick red"`** のようになります。
xls-border-bottom-color	セルの下端ボーダーの色です (jxl.format.Colour を参照)。
xls-border-bottom-line-style	セルの下端ボーダーの線スタイルです (jxl.format.LineStyle を参照)。
xls-border-bottom	セルの下端ボーダーの色と線スタイルの設定を略記で行います。 例えば **`style="xls-border-bottom: thick red"`** のようになります。
xls-border	セルの上下左右すべてのボーダーの色と線スタイルの設定を略記で行います。 例えば **`style="xls-border: thick red"`** のようになります。

19.14.4. 背景

以下の XLS-CSS 属性のグループはセルの背景を定義します。

xls-background-color	背景の色です (jxl.format.LineStyle を参照)。
xls-background-pattern	背景のパターンです (jxl.format.Pattern を参照)。
xls-background	背景の色とパターンの設定を略記で行います。

19.14.5. 列の設定

以下の XLS-CSS 属性のグループは列のプロパティを定義します。

xls-column-width	列の幅です。 約 5000 の値から開始して必要に応じて調整することをお勧めします。 XHTML モードで **`e:column`** により使用されます。
xls-column-widths	各列の幅です。 約 5000 の値から開始して必要に応じて調整することをお勧めします。 Excel エクスポータにより使用され、 データテーブルの **`style`** 属性に置かれます。 数値を使用するか、 列の迂回には「*」を使用してください。 たとえば、 **`style="xls-column-widths: 5000, 5000, *, 10000"`**
xls-column-autosize	列のサイズを自動的に決定するかどうかを判断します。 有効な値は **`true`** と **`false`** です。

xls-column-hidden	列を非表示にするかどうかを決めます。 有効な値は **true** と **false** です。
xls-column-export	列をエクスポートに表示させるかどうかを決めます。 有効な値は **true** と **false** です。 デフォルトでは**true** に設定されます。

19.14.6. セルの設定

以下の XLS-CSS 属性のグループはセルのプロパティを定義します。

xls-alignment	セル値のアライメントを行います (jxl.format.Alignment を参照)。
xls-force-type	セルの強制されたデータタイプを決定する文字列値です。 有効な値は **general**、**number**、**text**、**date**、**formula**、**bool** です。 データタイプは自動的に検出されるためこの属性を使うことはまれです。
xls-format-mask	セルのフォーマットマスクです (「書式マスク」 を参照)。
xls-indentation	セルの内容の字下げを決める数値です。
xls-locked	セルをロックするかどうかを設定します。workbook レベルの **locked** と併用します。 有効な値は **true** または **false** です。
xls-orientation	セル値の方向を設定します (jxl.format.Orientation を参照)。
xls-vertical-alignment	セル値の垂直方向の配置を設定します (jxl.format.VerticalAlignment を参照)。
xls-shrink-to-fit	セル値をセルに合わせて縮小するかどうかを設定します。 有効な値は **true** と **false** です。
xls-wrap	セルで新しい行を折り返すかどうかを決めます。 有効な値は **true** と **false** です。

19.14.7. データテーブルエクスポータ

データテーブルエクスポータにおいても XHTML モードの場合と同じ XLS-CSS 属性を使用しますが、その列幅は例外的にデータテーブルの **xls-column-widths** 属性を用いて定義されます (UIColumn が **style** や **styleClass** の属性をサポートしないためです)。

19.14.8. 制限

現在の Seam バージョンには CSS サポートに既知の制限がいくつかあります。

- **.xhtml** 文書を使用する場合、 スタイルシートは **<e:link>** タグで参照されなければなりません。
- データテーブルエクスポータの場合は、 CSS はスタイルの属性で与えられなければなりません。 外部のスタイルシートはサポートされません。

19.15. 国際化

使用されるリソースバンドルのキーは 2 種類のみです。 いずれも無効なデータ形式用で無効な値を定義するパラメータをとります。

- **`org.jboss.seam.excel.not_a_number`** – 数値であると想定された値が、実際にはそうではなかった場合のエラーメッセージです。
- **`org.jboss.seam.excel.not_a_date`** – 日付であると想定された値が、実際にはそうではなかった場合のエラーメッセージです。

19.16. リンクおよびその他のドキュメント

Seam での **Microsoft Excel** 機能のコアは JExcelAPI ライブラリをベースとし、 http://jexcelapi.sourceforge.net/ でご覧頂けます。 ほとんどの機能や制限はその JExcelAPI から継承されています。

注記

JExcelAPI は Seam ではありません。 Seam 関連の問題はすべて JBoss Seam JIRA の **Excel** モジュールで報告して頂くのが最適です。

第20章 電子メール

Seam には電子メールの送信およびテンプレート作成用のオプションコンポーネントが含まれています。

電子メールのサポートは **jboss-seam-mail.jar** により提供されます。 この **JAR** にはメールの作成に使用されるメール JSF コントロール、および **mailSession** マネージャコンポーネントが含まれます。

電子メールサポートのデモが Seam にありますので、 **examples/mail** プロジェクトをご覧ください。 正しいパッケージの方法を示すデモ、 および現在対応している主要な機能が数点含まれています。

Seam の統合テスト環境で電子メールのシステムをテストすることができます。 「Seam メールの統合テスト」 を参照してください。

20.1. メッセージの作成

Seam は Facelets を使用して電子メールのテンプレートを作成します。

```
<m:message xmlns="http://www.w3.org/1999/xhtml"
           xmlns:m="http://jboss.com/products/seam/mail"
           xmlns:h="http://java.sun.com/jsf/html">

  <m:from name="Peter" address="peter@example.com" />
  <m:to name="#{person.firstname} #{person.lastname}">
    #{person.address}
  </m:to>
  <m:subject>Try out Seam!</m:subject>

  <m:body>
    <p><h:outputText value="Dear #{person.firstname}" />,</p>
    <p>You can try out Seam by visiting
    <a href="http://labs.jboss.com/jbossseam">
      http://labs.jboss.com/jbossseam
      </a>.
    </p>
    <p>Regards,</p>
    <p>Pete</p>
  </m:body>

</m:message>
```

<m:message> タグはメッセージ全体を囲み、 Seam に電子メールのレンダリングを開始するよう指示します。 **<m:message>** タグ内では、 メッセージの送信者の指定に **<m:from>** タグ、 受信者の指定に **<m:to>** タグ、さらに **<m:subject>** タグを使用します (EL は通常の Facelets にあるためそれが使用される点に注意してください)。

<m:body> タグは電子メールの本文を囲みます。 HTML 正規タグを本文内や JSF コンポーネント内に使用することができます。

m:message がレンダリングされると、 **mailSession** が電子メールを送信するよう呼び出されます。 電子メールを送信するには Seam にそのビューをレンダリングするよう指示します。

```
@In(create=true)
private Renderer renderer;

public void send() {
    try {
        renderer.render("/simple.xhtml");
        facesMessages.add("Email sent successfully");
    } catch (Exception e) {
        facesMessages.add("Email sending failed: " + e.getMessage());
    }
}
```

たとえば、 無効な電子メールアドレスを入力すると例外が送出され、その例外はキャッチされユーザーに表示します。

20.1.1. 添付ファイル

Seam はファイルの処理に関してほとんどの Java の標準タイプに対応するため、 電子メールへのファイルの添付は簡単です。

たとえば、 **jboss-seam-mail.jar** を送信するには

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar"/>
```

Seam はファイルをクラスパスからロードして電子メールにそれを添付します。 デフォルトでは、 **jboss-seam-mail.jar** という名前で添付されますが、 **fileName** 属性を追加して編集すると添付ファイルの名前を変更することができます。

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar"
      fileName="this-is-so-cool.jar"/>
```

java.io.File、 **java.net.URL** を添付することもできます。

```
<m:attachment value="#{numbers}"/>
```

または、 **byte[]** あるいは **java.io.InputStream**

```
<m:attachment value="#{person.photo}" contentType="image/png"/>
```

byte[] や **java.io.InputStream** には添付ファイルの MIME タイプを指定する必要があります。 この情報はファイルの一部とはならないためです。

通常のタグの前後を **<m:attachment>** タグで囲むことで、Seam 生成の PDF や標準の JSF ビューを添付することができます。

```
<m:attachment fileName="tiny.pdf">
  <p:document>
    A very tiny PDF
  </p:document>
</m:attachment>
```

複数のファイル一式を添付する場合 – 例えばデータベースからロードした複数の写真一式など – **<ui:repeat>** を使うことができます。

```
<ui:repeat value="#{people}" var="person">
  <m:attachment value="#{person.photo}" contentType="image/jpeg"
                fileName="#{person.firstname}_#{person.lastname}.jpg"/>
</ui:repeat>
```

添付したイメージをインラインで表示させるには

```
<m:attachment value="#{person.photo}" contentType="image/jpeg"
              fileName="#{person.firstname}_#{person.lastname}.jpg"
              status="personPhoto" disposition="inline" />
<img src="cid:#{personPhoto.contentId}" />
```

cid:#{...} タグはイメージ検索が試行されたときに添付ファイルの検査が行われるよう指定します。 **cid – Content-ID** – が一致しなければなりません。

ステータスオブジェクトにアクセスする前に、添付ファイルを宣言しなければいけません。

20.1.2. HTML /Text 代替部分

ほとんどのメールリーダーは HTML に対応していますが、一部でサポートしていないメールリーダーもあります。 メール本文にプレーンテキストを入れることができます。

```
<m:body>
  <f:facet name="alternative">
    Sorry, your email reader can't show our fancy email. Please go to
    http://labs.jboss.com/jbossseam to explore Seam.
  </f:facet>
</m:body>
```

20.1.3. 複数の受信者

登録ユーザーなど、 複数の受信者で構成されるグループに電子メールを送信したいことがよくあります。 すべての受信者のメールタグを **<ui:repeat>** の中に置くことができます。

```
<ui:repeat value="#{allUsers} var="user">
  <m:to name="#{user.firstname} #{user.lastname}"
        address="#{user.emailAddress}"/>
</ui:repeat>
```

20.1.4. 複数のメッセージ

パスワードのリセットなど、 若干異なる内容のメッセージを各受信者に送信する必要がある場合もあります。 最適な方法としては、 メッセージ全体を **<ui:repeat>** 内に配置することです。

```
<ui:repeat value="#{people}" var="p">
  <m:message>
    <m:from name="#{person.firstname} #{person.lastname}">
      #{person.address}
    </m:from>
    <m:to name="#{p.firstname}">#{p.address}</m:to>
    ...
  </m:message>
</ui:repeat>
```

20.1.5. テンプレートの作成

メールテンプレートのサンプルでは Facelets のテンプレートが Seam のメールタグと動作することを示しています。

template.xhtml には次の内容が含まれています。

```
<m:message>
  <m:from name="Seam" address="do-not-reply@jboss.com" />
  <m:to name="#{person.firstname} #{person.lastname}">
    #{person.address}
  </m:to>
  <m:subject>#{subject}</m:subject>
  <m:body>
    <html>
      <body>
        <ui:insert name="body">
          This is the default body, specified by the template.
        </ui:insert>
      </body>
    </html>
  </m:body>
</m:message>
```

templating.xhtml には次の内容が含まれています。

```
<ui:param name="subject" value="Templating with Seam Mail"/>
<ui:define name="body">
  <p>
    This example demonstrates that you can easily use
    <i>facelets templating</i> in email!
  </p>
</ui:define>
```

電子メールには Facelets のソースタグも使用できます。 ソースタグは **WEB-INF/lib** の JAR ファイルに入れて置く必要があります。 Seam Mail を使用する場合は **web.xml** からの **.taglib.xml** の参照は信頼性に欠けるためです (非同期でメールを送信すると Seam Mail は JSF あるいは Servelt コンテキストにアクセスできないため、 **web.xml** の設定パラメータを認識しません)。

メール送信時にさらに Facelets や JSF の設定を行うためには、 Renderer コンポーネントを上書きしてプログラム的に設定を行う必要があります。これは上級ユーザーのみ行うようにしてください。

20.1.6. 国際化

Seam は国際化メッセージの送信に対応しています。デフォルトでは、 JSF によるエンコーディングが使用されますが、 テンプレートで上書きすることができます。

```
<m:message charset="UTF-8">
  ...
</m:message>
```

本文、 件名、 受信者名および送信者名はコード化されます。 テンプレートのエンコーディングを設定して、 Facelets にページの解析をする際に必ず正しい文字セットを使用させるようにする必要があります。

```
<?xml version="1.0" encoding="UTF-8"?>
```

20.1.7. その他のヘッダー

Seam は電子メールのヘッダーにもサポートを提供しています (「タグ」参照)。電子メールの重要度を設定し、受信者の受け取り確認を求めることができます。

```
<m:message xmlns:m="http://jboss.com/products/seam/mail"
           importance="low" requestReadReceipt="true"/>
```

または、 **<m:header>** タグを使うとメッセージにあらゆるヘッダーを追加することができます。

```
<m:header name="X-Sent-From" value="JBoss Seam"/>
```

20.2. 電子メールの受信

EJB を使用する場合は、MDB (メッセージ駆動型 Bean) を使って電子メールを受信することができます。JBoss は JCA アダプタ (**mail-ra.rar**) を提供していますが、以下のように**mail-ra.rar** を設定することも可能です。

```
@MessageDriven(activationConfig={
@ActivationConfigProperty(propertyName="mailServer",
                          propertyValue="localhost"),
@ActivationConfigProperty(propertyName="mailFolder",
                          propertyValue="INBOX"),
@ActivationConfigProperty(propertyName="storeProtocol",
                          propertyValue="pop3"),
@ActivationConfigProperty(propertyName="userName",
                          propertyValue="seam"),
@ActivationConfigProperty(propertyName="password",
                          propertyValue="seam")
})
@ResourceAdapter("mail-ra.rar")
@Name("mailListener")
public class MailListenerMDB implements MailListener {

    @In(create=true)
    private OrderProcessor orderProcessor;

    public void onMessage(Message message) {
        // Process the message
        orderProcessor.process(message.getSubject());
    }

}
```

受信されたメッセージは **onMessage(Message message)** を呼び出します。 ほとんどの Seam アノテーションは MDB の内側で動作しますが、 永続コンテキストにはアクセスしないでください。

20.3. 設定

アプリケーションに電子メールサポートを含めるためには、 **jboss-seam-mail.jar** を **WEB-INF/lib** ディレクトリに配置してください。 JBoss AS を使用する場合はこれ以上の設定は必要ありません。 JBoss AS を使用しない場合は JavaMail API と Java Activation Framework のコピーがあることを確認してください。 Seam で配信されるバージョンはそれぞれ **lib/mail.jar** と **lib/activation.jar** です。

注記

Seam Mail モジュールには **seam-ui** パッケージの使用とビューテクノロジーとして Facelets を使用する必要があります。ライブラリの今後のバージョンでは JSP の使用にも対応する可能性があります。

mailSession コンポーネントは「実際の」SMTP サーバと通信するときに JavaMail を使用します。

20.3.1. mailSession

Java EE 5 環境で作業している場合、 JavaMail セッションが JNDI ルックアップで使用できる場合があります。 これ以外は Seam 設定のセッションを使用します。

mailSession コンポーネントのプロパティについては「メール関連のコンポーネント」で詳しく説明しています。

20.3.1.1. JBoss AS の JNDI ルックアップ

JBoss AS の **deploy/mail-service.xml** で JNDI にバインドしている JavaMail セッションを設定します。 デフォルトのサービス設定は使用するネットワークに応じて変更する必要があります。 http://wiki.jboss.org/wiki/Wiki.jsp?page=JavaMail でサービスに関する詳細な記載をご覧ください。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:mail="http://jboss.com/products/seam/mail">
  <mail:mail-session session-jndi-name="java:/Mail"/>
</components>
```

Seam に JNDI から **java:/Mail** にバインドされるメールセッションを取得するよう指示します。

20.3.1.2. Seam 設定のセッション

メールセッションは **components.xml** で設定できます。 ここでは **smtp.example.com** を SMTP サーバーとして使用するよう Seam に指示します。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:core="http://jboss.com/products/seam/core"
            xmlns:mail="http://jboss.com/products/seam/mail">
  <mail:mail-session host="smtp.example.com"/>
</components>
```

20.4. タグ

電子メールは **http://jboss.com/products/seam/mail** の名前空間内でタグを使って生成されます。 ドキュメントには常にメッセージのルートに **message** タグがあるはずです。 メッセージタグは Seam による電子メール生成の準備を行います。

Facelets の標準のテンプレート作成タグは通常とおりに使用できます。 **body** 内ではすべての JSF タグを使用することができます。 スタイルシートや Javascript などの外部リソースへのアクセスがタグに必要な場合は、必ず **urlBase** を設定してください。

<m:message>

メールメッセージのルートタグです。

- **importance** – メールメッセージの重要度を設定します。 有効な値は **low**、 **normal**、 **high** です。 デフォルトは **normal** です。
- **precedence** – メッセージの優先度を設定します (**bulk** など)。
- **requestReadReceipt** – これを設定すると、受け取り確認が追加され **From:** アドレスに受け取り確認が送信されます。 デフォルトでは **false** に設定されています。
- **urlBase** – これを設定するとその値が **requestContextPath** の前に置かれ、電子メール中に **<h:graphicImage>** などのコンポーネントを使用できます。
- **messageId** – メッセージ ID を明示的に設定します。

<m:from>

電子メールに **From:** のアドレスを設定します。1 つの電子メールに対して 1 つのアドレスしか設定できません。

- **name** –電子メール発信者の名前です。
- **address** – 電子メール発信者のメールアドレスです。

<m:replyTo>

電子メールに **Reply-to:** のアドレスを設定します。1 つのメールに対して 1 つのアドレスしか設定できません。

- **address** – 電子メール発信者の電子メールアドレスです。

<m:to>

電子メールに受信者を追加します。 受信者が複数の場合は複数の **<m:to>** タグを使用します。 このタグは **<ui:repeat>** などの繰り返しタグ内に安全に配置できます。

- **name** – 受信者の名前です。
- **address** – 受信者の電子メールアドレスです。

<m:cc>

電子メールに CC の受信者を追加します。 CC が複数の場合は複数の <m:cc> タグを使用します。 このタグは <ui:repeat> などの繰り返しタグ内に安全に配置できます。

- **name** – 受信者の名前です。
- **address** – 受信者の電子メールアドレスです。

<m:bcc>

電子メールに BCC の受信者を追加します。 BCC が複数の場合は複数の **<m:bcc>** タグを使用します。 このタグは **<ui:repeat>** などの繰り返しタグ内に安全に配置できます。

- **name** – 受信者の名前です。
- **address** – 受信者の電子メールアドレスです。

<m:header>

電子メールにヘッダーを追加します (例、 **X-Sent-From: JBoss Seam**)。

- **name** – 追加するヘッダー名です (例、 **X-Sent-From**)。
- **value** – 追加するヘッダーの値です (例、 **JBoss Seam**)。

<m:attachment>

電子メールに添付ファイルを追加します。

- **value** – 添付するファイルです。
  - **String** – **String** はクラスパス内のファイルへのパスと解釈されます。
  - **java.io.File** – EL 式は **File** オブジェクトを参照することができます。
  - **java.net.URL** – EL 式は **URL** オブジェクトを参照することができます
  - **java.io.InputStream** – EL 式は **InputStream** を参照することができます。 この場合 **fileName** と **contentType** の両方を指定する必要があります。
  - **byte[]** – EL 式は **byte[]** を参照することができます。 この場合、 **fileName** と **contentType** の両方を指定する必要があります。

  値属性が省略される場合

  - このタグに **<p:document>** タグが含まれる場合、 記載されたドキュメントが生成され電子メールに添付されます。 **fileName** を指定してください。
  - このタグに他の JSF タグが含まれる場合、そこから HTML ドキュメントが生成され電子メールに添付されます。 **fileName** を指定してください。
- **fileName** – 添付ファイルに使用するファイル名を指定します。
- **contentType** – 添付ファイルの MIME タイプを指定します。

<m:subject>

電子メールに件名を設定します。

<m:body>

電子メールの本文を設定します。 **alternative** ファセットに対応します。 HTML 電子メールが生成されると HTML をサポートしていないメールリーダ用に代替となるテキストを含ませることができます。

- **type – plain** に設定するとプレーンテキストの電子メールを生成します。 これ以外は HTML 電子メールを生成します。

第21章 非同期性とメッセージング

Seam では Web 要求からの作業を非同期で容易に行うことができます。 Java EE での非同期は通常 JMS とつながっています。 サービス要件が厳密で明確に定義されている場合にはこれは適切な方法です。 Seam コンポーネントからの JMS メッセージの送信は簡単です。

しかし、多くのユースケースで JMS は必要以上にパワフルです。 Seam は選択した **ディスパッチャ** に対してシンプルで非同期なメソッドとイベントの機能を階層化します。

- **`java.util.concurrent.ScheduledThreadPoolExecutor`** (デフォルト)
- EJB タイマーサービス (EJB 3.0 環境向け)
- Quartz

21.1. 非同期性

非同期のイベントやメソッドの呼び出しは、基礎となるディスパッチャのメカニズムと同等のサービスが期待されます。 **`ScheduledThreadPoolExecutor`** をベースとするデフォルトのディスパッチャは効率的な働きをしますが、 永続非同期のタスクに対応していないためタスクが実際に実行されるかは保証されません。 EJB 3.0 に対応する環境で作業している場合は次の行を **`components.xml`** に追加して、 コンテナの EJB タイマーサービスによりタスクが非同期に処理されるようにします。

```
<async:timer-service-dispatcher/>
```

Seam で非同期メソッドを使用する場合、 タイマーサービスを直接操作する必要はありません。 ただし重要なことは、EJB3 実装には永続タイマーを使用するオプションがあるため、タスクが最終的には処理されることが保証されるという点です。

別の方法としては、 オープンソースの Quartz ライブラリを使って非同期メソッドを管理する方法です。 この場合、 **EAR** に Quartz ライブラリ **JAR (`lib` ディレクトリ)** を同梱してから **`application.xml`** で Java モジュールとして宣言します。 Quartz ディスパッチャはクラスパスに Quartz プロパティファイルを追加すると設定可能になります。このファイルは **`seam.quartz.properties`** という名前にしてください。 また、 Quartz ディスパッチャをインストールする場合は次の行を **`components.xml`** に追加する必要があります。

```
<async:quartz-dispatcher/>
```

Quartz **`Scheduler`**、 EJB3 **`Timer`**、 デフォルトの **`ScheduledThreadPoolExecutor`** の Seam API は非常によく似ているため、 **`components.xml`** に 1 行追加するだけで「プラグアンドプレイ」が可能です。

21.1.1. 非同期メソッド

非同期呼び出しにより、呼び出し側に対してメソッド呼び出しを非同期に (別のスレッドで) 処理することが可能です。通常クライアントに直ちに応答を返し、コスト高の作業をバックグラウンドで処理させたい場合には非同期呼び出しが使用されます。このパターンはクライアントが処理結果をサーバへ自動的にポーリングできるような AJAX を使用するアプリケーションでとても効果的です。

EJB コンポーネントの場合、Bean の実装にアノテーションを付与して、メソッドが非同期に処理されるよう指定します。JavaBean コンポーネントの場合は、コンポーネント実装クラスにアノテーションを付与します。

```
@Stateless
```

```
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
{
  @Asynchronous
  public void processPayment(Payment payment) {
    //do some work!
  }
}
```

非同期性の使用は **Bean** クラスには透過的です。また、クライアントにも透過的です。

```
@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
  @In(create=true) PaymentHandler paymentHandler;
  @In Bill bill;

  public String pay() {
    paymentHandler.processPayment( new Payment(bill) );
    return "success";
  }
}
```

非同期メソッドは新しいイベントコンテキストで処理され、呼び出し側のセッションまたは対話コンテキストの状態にはアクセスできません。しかしビジネスプロセスコンテキストは **伝播されます**。

実行を遅らせるために **@Duration**、**@Expiration**、**@IntervalDuration** のアノテーションを使って非同期メソッドの呼び出しをスケジュールすることができます。

```
@Local
public interface PaymentHandler {
  @Asynchronous
  public void processScheduledPayment(Payment payment,
                                      @Expiration Date date);

  @Asynchronous
  public void processRecurringPayment(Payment payment,
                                      @Expiration Date date,
                                      @IntervalDuration Long interval);
}
```

```
@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
  @In(create=true) PaymentHandler paymentHandler;
  @In Bill bill;

  public String schedulePayment() {
    paymentHandler.processScheduledPayment(new Payment(bill),
                                           bill.getDueDate() );
    return "success";
  }
```

```
  public String scheduleRecurringPayment() {
    paymentHandler.processRecurringPayment(new Payment(bill),
                                           bill.getDueDate(), ONE_MONTH
);
    return "success";
  }
}
```

クライアントとサーバーはいずれも呼び出しに関連付けられた **Timer** オブジェクトにアクセスすることができます。 以下に示す **Timer** オブジェクトは EJB3 ディスパッチャで使用される EJB3 タイマーです。 デフォルトの **ScheduledThreadPoolExecutor** の場合、 タイマーは JDK から **Future** を返します。 Quartz ディスパッチャの場合は **QuartzTriggerHandle** を返します。 これについては次項で説明していきます。

```
@Local
public interface PaymentHandler
{
  @Asynchronous
  public Timer processScheduledPayment(Payment payment,
                                       @Expiration Date date);
}
```

```
@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler {
  @In Timer timer;

  public Timer processScheduledPayment(Payment payment,
                                       @Expiration Date date) {
    //do some work!
    return timer; //note that return value is completely ignored
  }
}
```

```
@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
  @In(create=true) PaymentHandler paymentHandler;
  @In Bill bill;

  public String schedulePayment() {
    Timer timer =
      paymentHandler.processScheduledPayment(new Payment(bill),
                                             bill.getDueDate());
    return "success";
  }
}
```

非同期メソッドは呼び出し側に他のどんな値も返すことができません。

21.1.2. Quartz ディスパッチャを使った非同期メソッド

Quartz ディスパッチャでは上記のような **@Asynchronous**、 **@Duration**、 **@Expiration**、 **@IntervalDuration** のアノテーションが使用できます。 また、 他にもいくつかのアノテーションに対応しています。

@FinalExpiration アノテーションは反復タスクの終了日を指定します。 **QuartzTriggerHandle** をインジェクトできる点に注意してください。

```
@In QuartzTriggerHandle timer;

// Defines the method in the "processor" component
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                           @IntervalDuration Long
interval,
                                           @FinalExpiration Date
endDate,
                                           Payment payment) {
  // do the repeating or long running task until endDate
}

... ...

// Schedule the task in the business logic processing code
// Starts now, repeats every hour, and ends on May 10th, 2010
Calendar cal = Calendar.getInstance ();
cal.set (2010, Calendar.MAY, 10);
processor.schedulePayment(new Date(), 60*60*1000, cal.getTime(), payment);
```

このメソッドは **QuartzTriggerHandle** オブジェクトを返し、 これを使用してスケジューラの停止、 一時停止、 再開を行うことができます。 **QuartzTriggerHandle** オブジェクトはシリアライズ可能であるため、 長期間に渡りこれを維持する必要がある場合はデータベースに保存することができます。

```
QuartzTriggerHandle handle=
    processor.schedulePayment(payment.getPaymentDate(),
                              payment.getPaymentCron(),
                              payment);
payment.setQuartzTriggerHandle( handle );
// Save payment to DB

// later ...

// Retrieve payment from DB
// Cancel the remaining scheduled tasks
payment.getQuartzTriggerHandle().cancel();
```

@IntervalCron アノテーションはタスクのスケジューリングに Unix cron ジョブ構文をサポートします。 たとえば、 次の非同期メソッドは 3 月の毎水曜日の 2:10pm と 2:44pm に実行されます。

```
// Define the method
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                           @IntervalCron String cron,
                                           Payment payment) {
  // do the repeating or long running task
```

```
}

... ...

// Schedule the task in the business logic processing code
QuartzTriggerHandle handle =
     processor.schedulePayment(new Date(), "0 10,44 14 ? 3 WED", payment);
```

@IntervalBusinessDay アノテーションは「第X日営業日」というシナリオの呼び出しに対応します。 たとえば、 次の非同期メソッドは毎月第2営業日の 14:00 に実行されます。 デフォルトではすべての週末とアメリカ合衆国の祝日を営業日から除外します。

```
// Define the method
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                   @IntervalBusinessDay NthBusinessDay
nth,
                                   Payment payment) {
  // do the repeating or long running task
}

... ...

// Schedule the task in the business logic processing code
QuartzTriggerHandle handle =
    processor.schedulePayment(new Date(),
                              new NthBusinessDay(2, "14:00", WEEKLY),
                              payment);
```

NthBusinessDay オブジェクトには呼び出しトリガーの設定が含まれます。**additionalHolidays** プロパティで祝日を追加指定することができます (会社固有の休み、 アメリカ合衆国の祝日以外など)。

```
public class NthBusinessDay implements Serializable {
  int n;
  String fireAtTime;
  List<Date> additionalHolidays;
  BusinessDayIntervalType interval;
  boolean excludeWeekends;
  boolean excludeUsFederalHolidays;

  public enum BusinessDayIntervalType { WEEKLY, MONTHLY, YEARLY }

  public NthBusinessDay () {
    n = 1;
    fireAtTime = "12:00";
    additionalHolidays = new ArrayList<Date> ();
    interval = BusinessDayIntervalType.WEEKLY;
    excludeWeekends = true;
    excludeUsFederalHolidays = true;
  }
      ... ...
}
```

@IntervalDuration、 **@IntervalCron**、 **@IntervalNthBusinessDay** のアノテーションは互いに矛盾します。 同じメソッド内で使用しようとすると **RuntimeException** エラーの原因になります。

21.1.3. 非同期イベント

コンポーネント駆動のイベントも非同期にすることができます。 非同期処理するためにイベントを引き起こす場合は **Events** クラスの **raiseAsynchronousEvent()** メソッドを呼び出します。 指定時刻に起きるイベントをスケジュールするには、**raiseTimedEvent()** メソッドを呼び出し、スケジュールオブジェクトを渡します (デフォルトのディスパッチャまたはタイマーサービスのディスパッチャの場合は **TimerSchedule** を使用します)。 コンポーネントは通常通りに非同期のイベントを監視することができますが、 非同期スレッドに伝播されるのはビジネスプロセスコンテキストのみです。

21.1.4. 非同期の呼び出しによる例外処理

各非同期ディスパッチャは例外がそれを通じて伝播されるとそれぞれ異なった動作をします。 たとえば、 **java.util.concurrent** は繰り返す呼び出しの実行がこれ以上起きないよう一時停止し、EJB3 タイマーサービスがその例外を吸収します。 したがって、 非同期の呼び出しから伝播する例外がディスパッチャに到達する前に Seam によってその例外はキャッチされます。

デフォルトでは、 非同期実行から伝播する例外はすべてエラーレベルでキャッチされログ記録されます。 **org.jboss.seam.async.asynchronousExceptionHandler** コンポーネントを無効にするとこの動作をグローバルにカスタマイズすることができます。

```
@Scope(ScopeType.STATELESS)
@Name("org.jboss.seam.async.asynchronousExceptionHandler")
public class MyAsynchronousExceptionHandler
             extends AsynchronousExceptionHandler {
  @Logger Log log;

  @In Future timer;

  @Override
  public void handleException(Exception exception) {
    log.debug(exception);
    timer.cancel(false);
  }

}
```

上記は **java.util.concurrent** ディスパッチャを使ってその制御オブジェクトをインジェクトし、例外が送出された場合には今後の呼び出しをすべて取り消すようにしています。

また、 コンポーネント上にメソッド **public void handleAsynchronousException(Exception exception);** メソッドを実装すると個別のコンポーネントに対しこの動作を変更することができます。 たとえば、

```
public void handleAsynchronousException(Exception exception) {
  log.fatal(exception);
}
```

21.2. SEAM でのメッセージング

Seam コンポーネントからの JMS メッセージの送受信は簡単に行えます。

21.2.1. 設定

JMS メッセージを送信するよう Seam のインフラストラクチャを設定するには、 まず Seam にメッセージの送信先となるトピックおよびキューに関する情報、 要件に応じて **QueueConnectionFactory** や **TopicConnectionFactory** の場所を指示する必要があります。

デフォルトでは Seam は JBossMQ でのデフォルト接続ファクトリとなる **UIL2ConnectionFactory** を使用します。 別の JMS プロバイダを使用する場合は、 **seam.properties**、 **web.xml**、 **components.xml** のいずれかで **queueConnection.queueConnectionFactoryJndiName** と **topicConnection.topicConnectionFactoryJndiName** の一方あるいは両方を設定する必要があります。

Seam 管理の **TopicPublisher** と **QueueSender** をインストールするには、**components.xml** にトピックとキューも記載する必要があります。

```
<jms:managed-topic-publisher name="stockTickerPublisher"
     auto-create="true" topic-jndi-name="topic/stockTickerTopic"/>

<jms:managed-queue-sender name="paymentQueueSender"
     auto-create="true" queue-jndi-name="queue/paymentQueue"/>
```

21.2.2. メッセージ送信

設定が完了したら、 JMS **TopicPublisher** と **TopicSession** をコンポーネントにインジェクトすることができます。

```
@Name("stockPriceChangeNotifier")
public class StockPriceChangeNotifier {
  @In private TopicPublisher stockTickerPublisher;

  @In private TopicSession topicSession;

  public void publish(StockPrice price) {
    try {
      stockTickerPublisher.publish(topicSession
                                   .createObjectMessage(price));
    } catch (Exception ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

あるいは、 キューの場合は次のようになります。

```
@Name("paymentDispatcher")
public class PaymentDispatcher {
  @In private QueueSender paymentQueueSender;

  @In private QueueSession queueSession;

  public void publish(Payment payment) {
    try {
```

```
      paymentQueueSender.send(queueSession.createObjectMessage(payment));
    } catch (Exception ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

21.2.3. メッセージ駆動型 Bean を使用したメッセージの受信

EJB3 メッセージ駆動型 Bean を利用してメッセージ処理が可能です。 メッセージ駆動型 Bean は Seam コンポーネントとすることも可能です。 この場合、 イベントスコープでアプリケーションスコープの Seam コンポーネントのインジェクトが可能です。 次に支払いプロセッサに委任する支払いレシーバーの例を示します。

注記

@In アノテーションで **create** 属性を **true** に設定する必要がある場合があります。 これにより Seam はインジェクトされるコンポーネントのインスタンスを作成できます (コンポーネントが自動作成に対応しない場合にのみ必要となります。 つまり、 **@Autocreate** アノテーションが付与されません)。

まず、 メッセージを受信するメッセージ駆動型 Bean を作成します。

```
@MessageDriven(activationConfig =
    {@ActivationConfigProperty(propertyName = "destinationType",
                               propertyValue = "javax.jms.Queue"),
     @ActivationConfigProperty(propertyName = "destination",
                               propertyValue = "queue/paymentQueue")
})

@Name("paymentReceiver")
public class PaymentReceiver implements MessageListener
{
  @Logger private Log log;

  @In(create = true) private PaymentProcessor paymentProcessor;

  @Override
  public void onMessage(Message message)
  {
    try {
      paymentProcessor.processPayment((Payment) ((ObjectMessage)
                                        message).getObject());
    } catch (JMSException ex) {
      log.error("Message payload did not contain a Payment object", ex);
    }
  }
}
```

次に、 レシーバーによる支払い処理の委任先に Seam コンポーネントを実装します。

```
@Name("paymentProcessor")
public class PaymentProcessor {
  @In private EntityManager entityManager;
```

```
  public void processPayment(Payment payment) {
    // perhaps do something more fancy
    entityManager.persist(payment);
  }
}
```

メッセージ駆動型 Bean でトランザクションの動作を実行したい場合は、 XA データソースで作業するようにしてください。 そうでない場合は、データベーストランザクションがコミットする場合にはデータベースの変更をロールバックできなくなりますが、 その後のメッセージ動作は失敗します。

21.2.4. クライアントでのメッセージの受信

Seam Remoting によりクライアント側の JavaScript から JMS トピックにサブスクライブすることができます。 詳細は 24章 *リモーティング*をご覧ください。

第22章 キャッシュ

ほぼすべてのエンタープライズアプリケーションで主要なボトルネックとなるのがデータベースであり、ランタイム環境で最も拡張性に乏しい層であるため、データベースへのアクセス回数を低減するためにできることはすべてアプリケーションパフォーマンスの飛躍的な向上につながります。

適切に設計された Seam アプリケーションでは次のように何層にもわたる豊富なキャッシング戦略を実現し、アプリケーションのすべての層で利用することができるようになっています。

- データベース用のキャッシュを持っています。これは非常に重要ですが、アプリケーション層のキャッシュのような拡張性はありません。
- ORM ソリューション (Hibernate または別の JPA 実装) で提供されるデータベースの 2 次データキャッシュがあります。クラスタ環境でキャッシュデータをデータベースおよびその他のクラスタの両方とトランザクション的な永続性を持たせるのは、効率的な実装を行うという点で非常にコスト高となる場合があります。したがって、この 2 次キャッシュにはほとんど更新されることがないデータを格納するための使用と多くのユーザーとの共有に最適となります。従来のステートレスなアーキテクチャでは、この空間は対話的な状態の保存によく使用されます (非効率的)。
- 対話状態のキャッシュとなる Seam の対話コンテキストです。対話コンテキスト内のコンポーネントは、現在のユーザーのインタラクションに関連した状態を保持します。
- Seam 管理永続コンテキストは現在の対話に読み込まれたデータのキャッシュとして動作します (対話スコープのステートフルセッション Bean に関連付けられた EJB コンテナ管理の永続コンテキストを、Seam 管理永続コンテキストの代わりに使用することが可能です)。Seam によりクラスタ環境で Seam 管理永続コンテキストの複製が最適化され、楽観的ロック機能によりデータベースに一貫性のあるトランザクションを提供します。1 つの永続コンテキストに何千ものオブジェクトを読み込まない限り、このキャッシュによるパフォーマンスに及ぶ影響は最小限となります。
- Seam のアプリケーションコンテキストを使用するとトランザクションでない状態をキャッシュすることができます。ここに保持される状態はクラスタ内の他のノードからは見えません。
- アプリケーション内の Seam **`cacheProvider`** コンポーネントは、JBossCache または Ehcache を Seam 環境に統合します。キャッシュがクラスタモード内での実行をサポートする場合は、ここに保持される状態は他のノードにも見えます。
- 最後に、レンダリングされた JSF ページの断片をキャッシュすることができます。ORM の 2 次キャッシュと違い、データが更新されたときに自動的に無効にはならないため、明示的に無効化するアプリケーションコードを書くか、適切な有効期限ポリシーを設定する必要があります。

2 次キャッシュは非常に複雑な概念ですので、詳細はお使いの ORM ソリューションのドキュメントを参照してください。本項では **`cacheProvider`** コンポーネントを使用し直接キャッシュを行う方法、または **`<s:cache>`** コントロールを使用し保存されたページ断片としてキャッシュする方法について説明します。

22.1. SEAM でのキャッシュの使用

組み込みの **`cacheProvider`** コンポーネントは以下のインスタンスを管理します。

JBoss Cache 3.2.x

`org.jboss.cache.Cache`

EhCache

net.sf.ehcache.CacheManager

キャッシュ内に配置される不変の Java オブジェクトはすべてそこに格納され、 クラスタ全体に渡り複製されます (複製に対応しかつ有効な場合)。 変更可能なオブジェクトをキャッシュに持つためには、使用するキャッシュプロジェクトの関連文書を読み、格納されたオブジェクトに加えられたキャッシュの変更を通知する方法を調べてください。

cacheProvider を使うには、 プロジェクトにキャッシュ実装に関する JAR を含ませる必要があります。

JBoss Cache 3.2.x

- **jbosscache-core.jar** – JBoss Cache 3.2.x
- **jgroups.jar** – JGroups 2.6.x

Ehcache

- **ehcache.jar** – Ehcache 1.2.3

Seam の **EAR** デプロイメントでは、キャッシュ **JAR** と設定は直接 **EAR** に行くことが推奨されます。

また、JBossCache を使う場合は設定ファイルが必要となります。適切なキャッシュ設定を持つ **cache-configuration.xml** をクラスパスに置きます。例えば、EJB JAR または**WEB-INF/classes** です。JBossCache の設定に関する詳細は、JBossCache の文書を参照してください。

サンプルの **cache-configuration.xml** は **examples/blog/resources/META-INF/cache-configuration.xml** にあります。

Ehcache は設定ファイルがなくてもデフォルト設定で動作します。

使用中の設定ファイルを変更するには、 **components.xml**でキャッシュの設定を行います。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:cache="http://jboss.com/products/seam/cache">
  <cache:jboss-cache-provider
        configuration="META-INF/cache/cache-configuration.xml" />
</components>
```

これでいずれの Seam コンポーネントにもキャッシュをインジェクトすることができます。

```
@Name("chatroomUsers")
@Scope(ScopeType.STATELESS)

public class ChatroomUsers {
  @In CacheProvider cacheProvider;
  @Unwrap public Set<String> getUsers() throws CacheException   {
    Set<String> userList =
      (Set<String>) cacheProvider.get("chatroom", "userList");
    if (userList==null) {
      userList = new HashSet<String>();
      cacheProvider.put("chatroom", "userList", userList);
```

```
    } return userList;
  }
}
```

アプリケーションに使用できるキャッシュを複数設定する場合は、 **components.xml** を使用して複数のキャッシュプロバイダを設定します。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:cache="http://jboss.com/products/seam/cache">
  <cache:jboss-cache3-provider name="myCache"
        configuration="myown/cache.xml"/>
  <cache:jboss-cache3-provider name="myOtherCache"
        configuration="myother/cache.xml"/>
</components>
```

22.2. ページ断片のキャッシュ

Seam では **<s:cache>** タグが JSF におけるページ断片のキャッシュに関する問題を解決してくれます。 **<s:cache>** は **pojoCache** を内部的に使用するため前述の手順を行っておく必要があります。 **JAR** を **EAR** に配置してから追加の設定オプションを編集します。 これで使用できるようになります。

<s:cache> はあまり更新のないレンタリングされたコンテンツを保存します。 たとえば、 ブログのウェルカムページでは最新のブログエントリが表示されます。

```
<s:cache key="recentEntries-#{blog.id}" region="welcomePageFragments">
  <h:dataTable value="#{blog.recentEntries}" var="blogEntry">
    <h:column>
      <h3>#{blogEntry.title}</h3>
      <div>
        <s:formattedText value="#{blogEntry.body}"/>
      </div>
    </h:column>
  </h:dataTable>
</s:cache>
```

key を指定することによって各ページ断片の複数のバージョンを保存することができます。 この例では、 1 ブログに対して 1 キャッシュバージョンが存在します。 **region** には、すべてのバージョンを保存するキャッシュまたは領域ノードを指定します。異なるノードは異なる有効期限ポリシーを持つ場合があります。

<s:cache> の問題は基礎的データがいつ更新されるか認識できないことです。 このため、 変更が発生した場合はキャッシュされた断片を手作業で削除する必要があります。

```
public void post() {
  ...
  entityManager.persist(blogEntry);
  cacheProvider.remove("welcomePageFragments",
                       "recentEntries-" + blog.getId());
}
```

変更を直ちにユーザーに見せる必要がないのであれば、 キャッシュノードで有効期限を短く設定しても良いでしょう。

第23章 WEB サービス

Seam は JBossWS (JWS) と統合することで標準の Java EE の Web サービスに対話型 Web サービスのサポートなど Seam のコンテキストフレームワークの利点を十分に活用させることができます。 本章では Seam 環境向け Web サービスの設定について説明していきます。

23.1. 設定とパッケージング

Seam に Web サービス要求のコンテキストを作成させる場合、 まずそれらの要求へのアクセス権がなければなりません。 **org.jboss.seam.webservice.SOAPRequestHandler** は **SOAPHandler** の実装であり、 Web サービス要求のスコープの間 Seam コンポーネントのライフサイクルを管理します。

standard-jaxws-endpoint-config.xml (設定ファイル) は、Web サービスクラスを含む **JAR** ファイルの **META-INF** ディレクトリに配置されるはずです。 このファイルには以下のような SOAP ハンドラの設定が含まれています。

```
<jaxws-config xmlns="urn:jboss:jaxws-config:2.0"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:javaee="http://java.sun.com/xml/ns/javaee"
              xsi:schemaLocation=
                "urn:jboss:jaxws-config:2.0 jaxws-config_2_0.xsd">

  <endpoint-config>
    <config-name>Seam WebService Endpoint</config-name>

    <pre-handler-chains>
      <javaee:handler-chain>
        <javaee:protocol-bindings>
          ##SOAP11_HTTP
        </javaee:protocol-bindings>
        <javaee:handler>
          <javaee:handler-name>
            SOAP Request Handler
          </javaee:handler-name>
          <javaee:handler-class>
            org.jboss.seam.webservice.SOAPRequestHandler
          </javaee:handler-class>
        </javaee:handler>
      </javaee:handler-chain>

    </pre-handler-chains>

  </endpoint-config>

</jaxws-config>
```

23.2. 対話型 WEB サービス

Seam では SOAP 要求と応答のメッセージの両方で SOAP ヘッダーエレメントを使い、 その対話 ID を消費者側からサービスへ、 またサービスから消費者側へと伝えています。 以下は対話 ID を含む Web サービス要求の一例です。

```
<soapenv:Envelope
         xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
          xmlns:seam="http://seambay.example.seam.jboss.org/">

  <soapenv:Header>
    <seam:conversationId
xmlns:seam='http://www.jboss.org/seam/webservice'>
      2
    </seam:conversationId>
  </soapenv:Header>

  <soapenv:Body>
    <seam:confirmAuction/>
  </soapenv:Body>

</soapenv:Envelope>
```

上記の SOAP メッセージには **conversationId** エレメントが含まれ、 要求の対話 ID を含んでいます。 この例の場合は **2** です。 Web サービスを使う Web サービスのクライアントは多種多様で、なおかつさまざまな言語で記述されているため、 ひとつの対話のスコープ内で使われるそれぞれの Web サービス間の対話 ID の伝播を実装するのは開発者の責任となります。

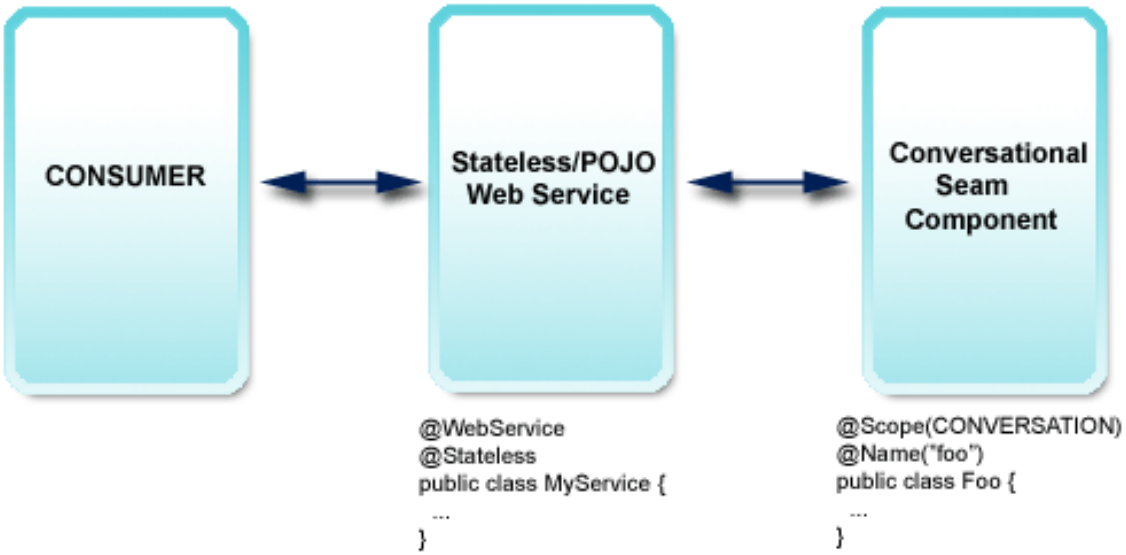
conversationId ヘッダーエレメントは **http://www.jboss.org/seam/webservice** の名前空間に適したものでなければいけません。 そうでないと、 Seam はその要求から対話 ID を読み取ることができなくなります。 上記の要求メッセージに対する応答の例を示します。

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
  <env:Header>
    <seam:conversationId
xmlns:seam='http://www.jboss.org/seam/webservice'>
      2
    </seam:conversationId>
  </env:Header>
  <env:Body>
    <confirmAuctionResponse
       xmlns="http://seambay.example.seam.jboss.org/"/>
  </env:Body>
</env:Envelope>
```

応答メッセージには要求と同じ **conversationId** エレメントが含まれているので留意してください。

23.2.1. 推奨される方法

Web サービスはステートレスセッション Bean または POJO で実装する必要があります。そのため 対話型 Web サービスには対話型 Seam コンポーネントの外観に Web サービスを実装させることをお勧めします。



Web サービスをステートレスセッション Bean で記述する場合は、 それに **@Name** アノテーションを付与することで Seam コンポーネントに変換することも可能です。 これにより Web サービスのクラス自体で Seam のバイジェクションやその他の機能を使用することができます。

23.3. WEB サービスの例

ここで例示するコードは seamBay サンプルアプリケーションからのコードです。これは Seam の **/examples** ディレクトリにあり、前項で述べた推奨される方法に添っています。 まず、 Web サービスのクラスとその Web サービスのメソッドから見てみましょう。

```
@Stateless
@WebService(name = "AuctionService", serviceName = "AuctionService")
public class AuctionService implements AuctionServiceRemote
{
  @WebMethod
  public boolean login(String username, String password)
  {
    Identity.instance().setUsername(username);
    Identity.instance().setPassword(password);
    Identity.instance().login();
    return Identity.instance().isLoggedIn();
  }

  // snip
}
```

この Web サービスは JSR-181 で定義されている通り、 **javax.jws** パッケージの JWS アノテーションが付与されたステートレスセッション Bean です。**@WebService** アノテーションがコンテナにこのクラスが Web サービスを実装することを伝えます。**login()** メソッドにある **@WebService** アノテーションはメソッドを Web サービスメソッドとして識別します。 **@WebService** アノテーションの **name** と **serviceName** の属性はオプションです。

Web サービスがステートレスセッション Bean である場合は、 Web サービスメソッドとして公開される各メソッドも Web サービスクラスのリモートインターフェース内で宣言される必要があります。 上記の例では、 **AuctionServiceRemote** インターフェースが **@WebMethod** でアノテーションが付与されているため、 **login()** メソッドを宣言しなければなりません。

上記の例にあるように、 Web サービスは Seam の組み込みの **Identity** コンポーネントに委譲する

login() メソッドを実装します。 推奨方法で提示しているように、 Web サービスは単にファサードとして記述されています。 実際の作業は Seam コンポーネント内で行われます。 つまり、 Web サービスと他のクライアント間でビジネスロジックは効率的に再利用されるということです。

次の例では、 この Web サービスメソッドは **AuctionAction.createAuction()** メソッドに委譲することで新しい対話を開始しています。

```
@WebMethod
public void createAuction(String title, String description, int
categoryId)
{
  AuctionAction action =
    (AuctionAction) Component.getInstance(AuctionAction.class, true);
  action.createAuction();
  action.setDetails(title, description, categoryId);
}
```

以下は、 **AuctionAction** からのコードです。

```
@Begin
public void createAuction()
{
  auction = new Auction();
  auction.setAccount(authenticatedAccount);
  auction.setStatus(Auction.STATUS_UNLISTED);
  durationDays = DEFAULT_AUCTION_DURATION;
}
```

ここでは、ファサードとして動作し実際の作業を対話型 Seam コンポーネントに委譲することで、Web サービスが長期実行の対話に参加する方法を示しています。

23.4. RESTEASY を使用した RESTFUL HTTP WEB サービス

Seam は JAX-RS 仕様 (JSR 311) の RESTEasy 実装を統合します。ご使用の Seam アプリケーションに統合する機能を以下から決定することができます。

- RESTEasy ブートストラップと設定、リソースの自動削除、およびプロバイダ
- SeamResourceServlet による HTTP/REST 要求、**web.xml** での外部のサーブレットおよび設定の必要性はなし
- Seam コンポーネントとしてのリソースの記述、Seam の完全なライフサイクルの管理とバイジェクション

23.4.1. RESTEasy 設定と要求

まず、RESTEasy ライブラリと **jaxrs-api.jar** をダウンロードします。それらを統合ライブラリ (**jboss-seam-resteasy.jar**) やご使用のアプリケーションに必要な他のライブラリとともにデプロイします。

seam-gen ベースのプロジェクトでは、これは、**jaxrs-api.jar**、**resteasy-jaxrs.jar**、および **jboss-seam-resteasy.jar** を **deployed-jars.list** (war デプロイメント) または **deployed-jars-ear.list** (ear デプロイメント) ファイルに追加することによって行えます。JBDS ベースのプ

ロジェクトの場合は、上記のライブラリを **EarContent/lib** (ear デプロイメント) または **WebContent/WEB-INF/lib** (war デプロイメント) フォルダにコピーし、IDE でプロジェクトをリロードします。

@javax.ws.rs.Path でアノテーション付与されたすべてのクラスは、起動時に自動的に発見され HTTP リソースとして登録されます。Seam は 組み込み**SeamResourceServlet** を使って自動的に HTTP 要求を受け入れ、提供します。リソースの URI は以下のように構築されます。

- URI は例として **/seam/resource** の **SeamResourceServlet** の **web.xml** でマップされたパターンで始まります。この設定を変更して、異なるベースでご使用の RESTful リソースを公開します。これは **グローバルな** 変更で、他の Seam リソース (**s:graphicImage**) もこのベースパスで提供されます。

- Seam の RESTEasy 統合ではベースパスに設定可能な文字列を追加します (デフォルトは **/rest**)。このため、サンプルではリソースの完全なベースパスは**/seam/resource/rest** となるでしょう。今後の REST API のアップグレードに備えてバージョン番号を追加するなど、ご使用のアプリケーションのこの文字列をさらに記述的なものに変更することをお薦めします。これにより旧のクライアントは旧の URI ベースを維持することができます。

- 最後に、リソースは定義された **@Path** で使用可能です。たとえば、**@Path("/customer")** でマップされたリソースは、**/seam/resource/rest/customer** で使用可能となります。

次のリソースの定義は、URI **http://your.hostname/seam/resource/rest/customer/123** を使用した GET 要求に対するプレーンテキスト表現を返します。

```
@Path("/customer")
public class MyCustomerResource {

    @GET
    @Path("/{customerId}")
    @Produces("text/plain")
    public String getCustomer(@PathParam("customerId") int id) {
         return ...;
    }

}
```

これらのデフォルトで望ましい場合は、追加で設定する必要はありません。ただし、必要ならばご使用の Seam アプリケーションで RESTEasy を設定することが可能です。まず、**resteasy** 名前空間を XML 設定ファイルのヘッダーにインポートします。

```
<components
   xmlns="http://jboss.com/products/seam/components"
   xmlns:resteasy="http://jboss.com/products/seam/resteasy"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
     "http://jboss.com/products/seam/resteasy
      http://jboss.com/products/seam/resteasy-2.2.xsd
      http://jboss.com/products/seam/components
      http://jboss.com/products/seam/components-2.2.xsd">
```

```
<resteasy:application resource-path-prefix="/restv1"/>
```

リソースへの完全なベースパスは **/seam/resource/restv1/{resource}** です。**@Path** の定義とマッピングは変わらない点に注意してください。これはアプリケーション全体に及ぶスイッチであり、通常は HTTP API のバージョニングに使用されます。

リソースに完全なパスをマップしたい場合は、ベースパスのストリッピングを無効にできます。

```
<resteasy:application strip-seam-resource-path="false"/>
```

ここでリソースのパスが **@Path("/seam/resource/rest/customer")** にマップされました。この機能を無効にすることで、リソースクラスのマッピングを特定のデプロイメントシナリオにバインドします。これは **推奨されません**。

Seam はクラスパスをすべてのデプロイされた **@javax.ws.rs.Path** リソースまたは **@javax.ws.rs.ext.Provider** クラスにスキャンします。以下のようにスキャンを無効にして、これらのクラスを手動で設定することが可能です。

```
<resteasy:application
     scan-providers="false"
     scan-resources="false"
     use-builtin-providers="true">

  <resteasy:resource-class-names>
    <value>org.foo.MyCustomerResource</value>
    <value>org.foo.MyOrderResource</value>
         <value>org.foo.MyStatelessEJBImplementation</value>
  </resteasy:resource-class-names>

  <resteasy:provider-class-names>
    <value>org.foo.MyFancyProvider</value>
  </resteasy:provider-class-names>

 </resteasy:application>
```

use-built-in-providers のスイッチは RESTEasy 組み込みプロバイダを有効 (デフォルト) または無効にします。これらはプレーンテキスト、JSON および JAXB マーシャリングを提供するため、有効にしておくことが推奨されます。

RESTEasy はリソースとして純粋な EJB (Seam コンポーネントでない EJB) に対応します。**web.xml** で移植可能でない方法で JNDI の名前を設定する代わりに (RESTEasy のドキュメントを参照)、上記に示したとおり **components.xml** でビジネスインターフェースではなく、単純に EJB 実装クラスを一覧にすることができます。EJB の **@Local** インターフェースに Bean 実装クラスではなく **@Path**、**@GET** などでアノテーションを付与する必要がある点に注意してください。これによりご使用のアプリケーションをグローバルな Seam **jndi-pattern** のスイッチをオン **<core:init/>** にした状態で、デプロイメントで移植可能に保つことができます。純粋な (Seam コンポーネントでない) EJB リソースはリソースのスキャンが有効であっても見つからず、常に手動で一覧化する必要がある点に注意してください。繰り返しになりますが、このパラグラフ全体は Seam コンポーネントでなく、かつ **@Name** アノテーションを持っていない EJB リソースに関連しているだけです。

最後に、メディアタイプと言語 URI 拡張子を設定できます。

```
<resteasy:application>

    <resteasy:media-type-mappings>
      <key>txt</key>
      <value>text/plain</value>
```

```
    </resteasy:media-type-mappings>

    <resteasy:language-mappings>
       <key>deutsch</key><value>de-DE</value>
    </resteasy:language-mappings>

</resteasy:application>
```

この定義は **.txt.deutsch** の URI サフィックスを追加の **Accept**、**Accept-Language** ヘッダーの値、**text/plain**、**de-DE** にマップします。

23.4.2. Seam コンポーネントとしてのリソースとプロバイダ

リソースとプロバイダインスタンスは、デフォルトで RESTEasy により管理されます。リソースクラスは RESTEasy によりインスタンスが作成され単一の要求を提供し、その後破棄されます。これはデフォルトの JAX-RS ライフサイクルです。プロバイダはアプリケーション全体に対し 1 度インスタンスが作成されます。これらはステートレスなシングルトンです。

リソースとプロバイダは Seam コンポーネントとしても記述可能で、Seam のより効果的なライフサイクル管理、バイジェクション、セキュリティの能力の利点を活用できます。以下のようにリソースクラスを Seam のコンポーネントにします。

```
@Name("customerResource")
@Path("/customer")
public class MyCustomerResource {

    @In
    CustomerDAO customerDAO;

    @GET
    @Path("/{customerId}")
    @Produces("text/plain")
    public String getCustomer(@PathParam("customerId") int id) {
         return customerDAO.find(id).getName();
    }

}
```

これで **customerResource** インスタンスは、要求がサーバーに到達したときに Seam により処理されます。このコンポーネントはイベントスコープであるため、そのライフサイクルは JAX-RS のライフサイクルとまったく同じです。ただし、Seam JavaBean コンポーネントはインジェクションに完全に対応しており、他のすべてのコンポーネントとコンテキストに完全にアクセスすることができます。セキュリティ、アプリケーション、ステートレスなリソースコンポーネントもまた対応しています。これら 3 つのスコープにより、ステートレスな Seam の中間層の HTTP 要求処理アプリケーションを効果的に作成できます。

インターフェースにアノテーションを付与することで、その実装には JAX-RS アノテーションを付けない状態に保つことが可能です。

```
@Path("/customer")
public interface MyCustomerResource {

    @GET
    @Path("/{customerId}")
    @Produces("text/plain")
```

```
    public String getCustomer(@PathParam("customerId") int id);

}
```

```
@Name("customerResource")
@Scope(ScopeType.STATELESS)
public class MyCustomerResourceBean implements MyCustomerResource {

    @In
    CustomerDAO customerDAO;

    public String getCustomer(int id) {
         return customerDAO.find(id).getName();
    }

}
```

SESSION スコープの Seam コンポーネントを使用できます。ただしデフォルトでは、セッションは短くなり単一の要求となります。言い換えると HTTP 要求が RESTEasy 統合コードにより処理されている場合、HTTP セッションが作成されるため Seam コンポーネントはそのコンテキストを活用できます。要求が処理されたら、Seam はセッションを調べて、セッションがその単一の要求を提供するためだけに作成されたかを決定します (要求を持つセッション識別子はありません。要求のために存在したセッションもありません)。セッションがこの要求のためだけに作成された場合は、このセッションは要求後に破棄されます。

Seam アプリケーションがイベント、アプリケーション、またはステートレスなコンポーネントのみを使用すると仮定しましょう。この手順はサーバー上の使用可能な HTTP セッションの消費を防ぎます。Seam と RESTEasy の統合は、デフォルト設定でセッションが使用されないと仮定しています。したがって、各 REST 要求がタイムアウトしたときのみ削除されるセッションを開始するため、貧弱なセッションは増加します。

RESTful Seam アプリケーションがセッション状態を REST HTTP 要求全体に渡って保存する必要がある場合は、設定ファイルでこの動作を無効にします。

```
<resteasy:application destroy-session-after-request="false"/>
```

これで各 REST HTTP 要求は **Session.instance().invalidate()** によりコード内でタイムアウトまたは明示的な無効化によってのみ削除される新しいセッションを作成します。要求全体に渡りセッションコンテキストを活用したい場合は、HTTP 要求とともに有効なセッション識別子を渡すことはご自身の責任となります。

対話スコープのリソースコンポーネントと対話マッピングは現在サポートされていませんが、Seam の今後のバージョンではサポートされる予定です。

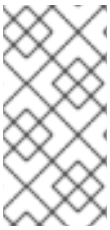
プロバイダクラスは Seam コンポーネントとなることも可能です。アプリケーションによるスコープかステートレスのどちらかでなければなりません。

リソースとプロバイダは他の Seam コンポーネントのように、EJB または JavaBean となることが可能です。

EJB Seam コンポーネントは REST リソースとしてサポートされています。EJB 実装クラスではなく必ずローカルビジネスインターフェースに JAX-RS でアノテーションを付与するようにします。EJB は **STATELESS** である必要があります。

注記

RESTEasy コンポーネントはホットデプロイメントをサポートしません。したがって、これらのコンポーネントを **src/hot** フォルダーに置かないでください。代わりに **src/main** フォルダーを使用してください。



注記

3.4.1 項の JAX RS 仕様で定義されたサブリソースは、この時点では Seam コンポーネントインスタンスとなることはできません。ルートリソースクラスのみが Seam コンポーネントとして登録可能です。言い換えると、ルートリソースメソッドから Seam コンポーネントインスタンスを返さないようにしてください。

23.4.3. リソースをセキュアにする

components.xml で HTTP ベーシック認証およびダイジェスト認証に対し Seam 認証フィルタを有効にできます。

```
<web:authentication-filter url-pattern="/seam/resource/rest/*" auth-
type="basic"/>
```

認証ルーチンの記述方法は「セキュリティ」の章をご確認ください。

認証が成功すると、一般的な **@Restrict** および **@PermissionCheck** のアノテーションを使用した承認ルールが有効になります。クライアント **Identity** へのアクセス、パーミッションマッピングとの連携などが可能です。承認に関するすべての Seam が持つ通常のセキュリティ機能は使用可能です。

23.4.4. HTTP 応答への例外のマップ

JAX-RS 仕様の 3.3.4 項では、JAX RS がチェック例外と非チェック例外を処理する方法について定義しています。Seam と RESTEasy を統合することで、Seam の **pages.xml** 内の HTTP 応答コードに例外をマップすることが可能です。**pages.xml** をすでに使用している場合は、これは多くの JAX RS 例外のマッパークラスよりも維持し易いです。

Seam 内で処理される例外については、Seam フィルタは HTTP 要求に対し実行する必要があります。REST 要求を扱わない要求 URI パターンとして **ではなく**、**web.xml** のすべての要求をフィルタする必要があります。次のサンプルはすべての HTTP 要求をインターセプトし、Seam 例外処理を有効にします。

```
<filter>
    <filter-name>Seam Filter</filter-name>
    <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>Seam Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

リソースメソッドで投げられた非チェック例外 **UnsupportedOperationException** を **501 Not Implemented** HTTP ステータス応答に変換するには、次を **pages.xml** 記述子に追加します。

```
<exception class="java.lang.UnsupportedOperationException">
```

```
    <http-error error-code="501">
        <message>The requested operation is not supported</message>
    </http-error>
</exception>
```

カスタム例外またはチェック例外は同じように処理されます。

```
<exception class="my.CustomException" log="false">
    <http-error error-code="503">
        <message>Service not available: #
{org.jboss.seam.handledException.message}</message>
    </http-error>
</exception>
```

例外が発生した場合 HTTP エラーをクライアントに送る必要はありません。Seam により リダイレクトとして例外を Seam アプリケーションのビューにマップすることが可能です。この機能は通常、REST API リモートクライアントではなく人間のクライアント (Web ブラウザ) に対して使用されるため、**pages.xml** の競合する例外マッピングに注意する必要があります。

HTTP レスポンスはサーブレットコンテナを通るため、**web.xml** 設定に **<error-page>** マッピングがある場合は、追加のマッピングを適用することが可能です。次に HTTP ステータスコードは **200 OK** のステータスを持つレンダリングされた HTML エラーページにマップされます。

23.4.5. RESTful API によるエンティティの公開

Seam により RESTful の方法を使用して、アプリケーションデータにアクセスすることは実に簡単です。Seam が導入した改善点のひとつとして、純粋な HTTP 呼び出しによるリモートアクセスに対し SQL データーベースの一部を公開することができることです。このため、Seam/RESTEasy 統合モジュールは 2 つのコンポーネント **ResourceHome** と **ResourceQuery**を提供します。これは Seam Application Framework (13章*Seam アプリケーションフレームワーク*) により提供された API から利点を受けています。これらのコンポーネントにより、ドメインモデルエンティティクラスを HTTP API にバインドすることができます。

23.4.5.1. ResourceQuery

ResourceQuery は RESTful Web サービスのような機能をクエリするエンティティを公開します。デフォルトでは、基礎となるシンプルな Query コンポーネントは、任意のエンティティクラスのインスタンスの一覧を返し、自動的に作成されます。もしくは、ResourceQuery コンポーネントはより高度なクラスの既存の Query コンポーネントにリンクすることができます。次のサンプルではどれほど簡単に ResourceQuery を設定できるか示しています。

```
<resteasy:resource-query
   path="/user"
   name="userResourceQuery"
   entity-class="com.example.User"/>
```

この単一の XML エレメントにより、ResourceQuery コンポーネントが設定されます。設定は簡単です。

- コンポーネントは **com.example.User** インスタンスの一覧を返します。
- コンポーネントは URI パス **/user** の HTTP 要求を処理します。
- コンポーネントはデフォルトではデータを (クライアントの好みに基づき) XML または JSON に変換します。対応する MIME タイプのセットは **media-types** 属性を使用して変更できま

す。例えば以下のとおりです。

```
<resteasy:resource-query
   path="/user"
   name="userResourceQuery"
   entity-class="com.example.User"
   media-types="application/fastinfoset"/>
```

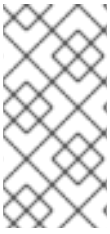
XML を使用してコンポーネントを設定するのを好まない場合は、別の方法として拡張機能によってコンポーネントを設定できます。

```
@Name("userResourceQuery")
@Path("user")
public class UserResourceQuery extends ResourceQuery<User>
{
}
```

Query は読み取り専用の動作で、リソースは GET 要求にのみ応答します。さらに Web サービスのクライアントは、ResourceQuery を使用することで次のパスパラメータを使ってクエリの結果のセットを操作できます。

パラメータの名前	例	詳細
start	/user?start=20	20 番目のエントリで始まるデーターベースクエリの結果のサブセットを返します。
show	/user?show=10	10 エントリに制限されたデーターベースクエリの結果のサブセットを返します。

例えば、HTTP GET 要求を **/user?start=30&show=10** に送り、行 30 で始まる 10 行を表すエントリの一覧を取得します。



注記

RESTEasyは JAXB を使用してエンティティをマーシャルします。よって、それらをワイヤで転送するには、**@XMLRootElement** でエンティティクラスにアノテーションを付与する必要があります。詳細は JAXB および RESTEasy のドキュメントを参照してください。

23.4.5.2. ResourceHome

リモートアクセスに対し ResourceQuery が Query の API を使用可能にするのと同じように、Home コンポーネントに対しては ResourceHome が使用可能にします。以下の表は 2 つの API (HTTP と Home) がどのようにバインドされているかを示しています。

表23.1 ResourceHome でのバインディング

HTTP メソッド	パス	機能	ResourceHome メソッド
GET	{path}/{id}	読み取り	getResource()
POST	{path}	作成	postResource()
PUT	{path}/{id}	更新	putResource()
DELETE	{path}/{id}	削除	deleteResource()

- HTTP 要求を /user/{userId} に送ることで、特定のユーザーインスタンスを「GET」、「PUT」、「DELETE」できます。
- POST 要求を **/user** に送ることで、新しいユーザーエンティティインスタンスを作成、永続化します。通常はそれを永続層に任せて、識別子値、URI が付いたエンティティインスタンスを提供します。そのため URI は HTTP レスポンスのヘッダー **Location** のクライアントに送られます。

ResourceHome の設定は ResourceQuery と非常に似ています。異なる点は、基礎となる Home コンポーネントとエンティティ識別子プロパティの Java タイプを明示的に指定する必要があることです。

```
<resteasy:resource-home
   path="/user"
   name="userResourceHome"
   entity-home="#{userHome}"
   entity-id-class="java.lang.Integer"/>
```

繰り返しになりますが、XML ではなく ResourceHome のサブクラスを記述できます。

```
@Name("userResourceHome")
@Path("user")
public class UserResourceHome extends ResourceHome<User, Integer>
{

   @In
   private EntityHome<User> userHome;

   @Override
   public Home<?, User> getEntityHome()
   {
      return userHome;
   }
}
```

ResourceHome および ResourceQuery コンポーネントのさらなる例は、**Seam Tasks** サンプルのアプリケーションを見てください。Seam/RESTEasy 統合と jQuery Web クライアントを共に使用する方法について分かりやすく説明しています。さらに、**Restbay** サンプルでさらにコードサンプルを見ることができます。これは主にテスト目的で使用されます。

23.4.6. リソースとプロバイダのテスト

Seam にはユーティリティクラスをテストするユニットが含まれ、RESTful アーキテクチャに対しユニットテストを作成するのに役立ちます。**SeamTest** クラスを通常どおり拡張し、**ResourceRequestEnvironment.ResourceRequest** を使用して HTTP 要求 / 応答サイクルをエミュレートします。

```
import org.jboss.seam.mock.ResourceRequestEnvironment;
import org.jboss.seam.mock.EnhancedMockHttpServletRequest;
import org.jboss.seam.mock.EnhancedMockHttpServletResponse;
import static
org.jboss.seam.mock.ResourceRequestEnvironment.ResourceRequest;
import static org.jboss.seam.mock.ResourceRequestEnvironment.Method;

public class MyTest extends SeamTest {

   ResourceRequestEnvironment sharedEnvironment;

   @BeforeClass
   public void prepareSharedEnvironment() throws Exception {
       sharedEnvironment = new ResourceRequestEnvironment(this) {
            @Override
            public Map<String, Object> getDefaultHeaders() {
               return new HashMap<String, Object>() {{
                   put("Accept", "text/plain");
               }};
            }
         };
   }

   @Test
   public void test() throws Exception
   {
      //Not shared: new ResourceRequest(new
ResourceRequestEnvironment(this), Method.GET, "/my/relative/uri)

      new ResourceRequest(sharedEnvironment, Method.GET,
"/my/relative/uri)
      {
         @Override
         protected void prepareRequest(EnhancedMockHttpServletRequest
request)
         {
            request.addQueryParameter("foo", "123");
            request.addHeader("Accept-Language", "en_US, de");
         }

         @Override
         protected void onResponse(EnhancedMockHttpServletResponse
response)
         {
            assert response.getStatus() == 200;
            assert response.getContentAsString().equals("foobar");
         }

      }.run();
   }
}
```

–

このテストはローカル呼び出しを実行するだけで、TCP を通じて **SeamResourceServlet** と通信しません。モック要求は Seam サーブレットに渡されフィルタされて、そのレスポンスはテストアサーションのために使用可能となります。**ResourceRequestEnvironment** の共有インスタンスの **getDefaultHeaders()** メソッドを上書きすることで、テストクラスの各テストメソッドに対して要求ヘッダーを設定できます。

ResourceRequest は **@Test** メソッドまたは **@BeforeMethod** コールバックで実行する必要があります。**@BeforeClass** のような他のコールバックでは実行できません。

第24章 リモーティング

Seam は Web ページからコンポーネントへのリモートアクセスに Asynchronous JavaScript and XML (AJAX) を使用します。 この機能のフレームワークの開発にはほとんど労力を必要としません。コンポーネントを単純なアノテーションで AJAX によりアクセス可能にします。本章では AJAX が有効な Web ページの作成に必要な手順、 そして Seam Remoting フレームワークに関する詳細についても説明していきます。

24.1. 設定

リモーティングの機能を使用するには、 まず **web.xml** ファイル内で Seam Resource Servlet を設定する必要があります。

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.servlet.SeamResourceServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

次のステップは Web ページに必要な JavaScript をインポートすることです。 インポートされるスクリプトは 2 つ以上必要です。 最初のスクリプトにはリモーティングの機能を有効にするクライアント側フレームワークの全コードが含まれます。

```
<script type="text/javascript"
        src="seam/resource/remoting/resource/remote.js">
</script>
```

2 つ目のスクリプトは、 呼び出したいコンポーネントのスタブと型定義を含みます。 これはコンポーネントのローカルインターフェースに応じて動的に生成され、 インターフェースのリモート可能なメソッドの呼び出しに使用できる全クラスの型定義を含みます。 スクリプトの名前にはコンポーネントの名前が反映されます。 例えば、 **@Name("customerAction")** アノテーションをステートレスセッション Bean に付与する場合、 スクリプトタグは以下のようになります。

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?customerAction">
</script>
```

同じページから 1 つ以上のコンポーネントにアクセスしたい場合は、 スクリプトタグのパラメータとしてそれらをすべて含めます。

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?
customerAction&accountAction">
</script>
```

必要な Javascript のインポートに **s:remote** タグを使用することもできます。 インポートしたいコンポーネントやクラス名はそれぞれコンマで区切ります。

```
<s:remote include="customerAction,accountAction"/>
```

24.2. SEAM オブジェクト

クライアント側からのコンポーネントとのやりとりは **remote.js** で定義される **Seam** Javascript オブジェクトで行われます。 コンポーネントに対する非同期呼び出しに使用されます。 オブジェクトはコンポーネントと連携するメソッドを含む **Seam.Component** そしてリモート要求を実行するメソッドを含む **Seam.Remoting** の 2 つの機能に区分されます。 このオブジェクトに精通する最も容易な方法は簡単なサンプルから始めることです。

24.2.1. Hello World サンプル

手順24.1 Hello World の例

1. **Seam** オブジェクトがどのように動作するかを見るために、 まず**helloAction** と呼ばれる新しい Seam コンポーネントを作成します。

```
@Stateless
@Name("helloAction")
public class HelloAction implements HelloLocal {
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}
```

2. 新しいコンポーネント用にローカルのインターフェースも生成する必要があります。 **@WebRemote** アノテーションはリモーティングでメソッドへのアクセスを可能にするために必要となるので特に注意してください。

```
@Local
public interface HelloLocal {
    @WebRemote
    public String sayHello(String name);
}
```

3. 記述する必要があるサーバー側のコードはこれだけです。次に新しい **Web** ページを作成して **helloAction** コンポーネントをインポートします。

```
<s:remote include="helloAction"/>
```

4. このページにボタンを追加して対話式のユーザーエクスペリエンスにします。

```
<button onclick="javascript:sayHello()">Say Hello</button>
```

5. また、 ボタンをクリックしたときに動作を実行するスクリプトが必要になります。

```
<script type="text/javascript">
function sayHello() {
     var name = prompt("What is your name?");
     Seam.Component.getInstance("helloAction").sayHello(name,
sayHelloCallback);
```

```
}

function sayHelloCallback(result) {
    alert(result);
}
</script>
```

6. アプリケーションをデプロイしてページを見てみます。 ボタンをクリックしてプロンプトが出たら名前を入力します。 呼び出しが成功であることを確認する「Hello」メッセージがメッセージボックスに表示されます (Seam の **/examples/remoting/helloworld** ディレクトリにこの Hello World サンプルの全ソースコードがあります)。

Javascript コードから 2 つのメソッドを実装していることがわかります。 最初のメソッドはユーザーに対して名前を入力するよう促しリモート要求を行います。 以下の行を見てみましょう。

```
Seam.Component.getInstance("helloAction").sayHello(name,
sayHelloCallback);
```

この行の最初の部分 (**Seam.Component.getInstance("helloAction")**) は **helloAction** コンポーネントのプロキシまたは **スタブ** を返します。 この行の残りの部分 (**sayHello(name, sayHelloCallback);**) はスタブに対してコンポーネントのメソッドを呼び出します。

コード行全体で行っていることは、 コンポーネントの **sayHello** メソッドを呼び出し、 パラメータとして **name** を渡しています。2 番目のパラメータ **sayHelloCallback** はこのコンポーネントの **sayHello** メソッドのパラメータではありません。 Seam Remoting フレームワークは、要求に対する応答を受けたらそれを **sayHelloCallback** Javascript メソッドに渡されるべきことを指示しています (このコールバックパラメータはオプションとなるため、 **void** 戻りタイプでメソッドを呼び出している場合、または要求の結果を気にする必要がない場合は、 そのままにしておいて構いません)。

sayHelloCallback メソッドがリモート要求に対する応答を受け取ると、メソッド呼び出しの結果を表示する警報メッセージを表示します。

24.2.2. Seam.Component

Seam.Component Javascript オブジェクトは Seam コンポーネントと連携するクライアント側メソッドをいくつか提供します。 主となる 2 つのメソッド、 **newInstance()** と **getInstance()** については本項の後半で詳しく記載しています。 **newInstance()** は常にコンポーネントタイプの新しいインスタンスを作成し、 **getInstance()** はシングルトンのインスタンスを返すことが主な違いとなります。

24.2.2.1. Seam.Component.newInstance()

エンティティまたは JavaBean コンポーネントの新しいインスタンスを作成するためにこのメソッドを使用します。返されるオブジェクトはそのサーバー側と同じ getter / setter のメソッドを持ちます。 また、 そのフィールドに直接アクセスすることも可能です。 たとえば

```
@Name("customer")
@Entity
public class Customer implements Serializable
{
    private Integer customerId;
    private String firstName;
    private String lastName;
```

```
    @Column public Integer getCustomerId() {
        return customerId;
    }

    public void setCustomerId(Integer customerId} {
        this.customerId = customerId;
    }

    @Column public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @Column public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

クライアント側の **Customer** を作成するには、 以下のコードを記述します。

```
var customer = Seam.Component.newInstance("customer");
```

ここから **customer** オブジェクトのフィールドを設定することができます。

```
customer.setFirstName("John"); // Or you can set the fields directly
// customer.lastName = "Smith";
```

24.2.2.2. Seam.Component.getInstance()

getInstance() メソッドを使って **Seam** セッション **Bean** コンポーネントのスタブを参照します。 次にこれを使ってコンポーネントに対して遠隔的にメソッドを実行することができます。 このメソッドは指定コンポーネントのシングルトンを返すため、 続けて同じコンポーネント名で **2** 回呼び出すとそのコンポーネントの同じインスタンスが返されます。

前述の例を続行するためには、 新しい **customer** を作成し保存したい場合は、**customerAction** コンポーネントの **saveCustomer()** メソッドにそれを渡します。

```
Seam.Component.getInstance("customerAction").saveCustomer( customer);
```

24.2.2.3. Seam.Component.getComponentName()

このメソッドにオブジェクトを渡すと、 それがコンポーネントの場合はコンポーネント名を返し、そうでない場合には **null** を返します。

```
if (Seam.Component.getComponentName(instance) == "customer")
    alert("Customer");
```

```
else if (Seam.Component.getComponentName(instance) == "staff")
    alert("Staff member");
```

24.2.3. Seam.Remoting

Seam Remoting のクライアント側の機能のほとんどは **Seam.Remoting** オブジェクト内に含まれます。 そのメソッドの多くは直接呼び出す必要はないはずですが、 言及する価値のある重要なものがいくつかあります。

24.2.3.1. Seam.Remoting.createType()

アプリケーションが Seam コンポーネントではない JavaBean のクラスを含むまたは使用する場合、クライアント側でこれらのタイプを作成してパラメータとしてコンポーネントメソッドに渡す必要がある場合があります。 タイプのインスタンスを作成するには **createType()** メソッドを使用します。 パラメータとして、完全修飾の Java クラス名を渡してください。

```
var widget = Seam.Remoting.createType("com.acme.widgets.MyWidget");
```

24.2.3.2. Seam.Remoting.getTypeName()

このメソッドは **Seam.Component.getComponentName()** と同等となる非コンポーネントです。 オブジェクトインスタンスにタイプの名前を返します。 または、 タイプが既知でない場合は **null** を返します。 この名前は、タイプの Java クラス完全修飾名です。

24.3. EL 式の評価

Seam Remoting は EL 式の評価にも対応し、これはサーバーからのデータ取得に便利なもうひとつの方法です。 **Seam.Remoting.eval()** 関数を使用して EL 式をサーバー上で遠隔に評価してその結果値をクライアント側のコールバックメソッドに返すことができます。 この関数は 2 つのパラメータを受け取ります。1 番目のパラメータは評価対象となる EL 式であり、2番目のパラメータはその式の値を付けて呼び出すコールバックメソッドです。 次に例を示します。

```
function customersCallback(customers) {
    for (var i = 0; i < customers.length; i++) {
        alert("Got customer: " + customers[i].getName());
    }
}

Seam.Remoting.eval("#{customers}", customersCallback);
```

この例では、 **#{customers}** の式が Seam によって評価され、 その式の値 (この場合 **Customer** オブジェクトの一覧) が**customersCallback()** メソッドに返されます。 このようにして返されるオブジェクトは Javascript で動作できるよう **s:remote** でそれ自体のタイプがインポートされていなければなりません。 **customer** オブジェクトの一覧と動作させるには、 **customer** タイプをインポートする必要があります。

```
<s:remote include="customer"/>
```

24.4. クライアントのインタフェース

上記の設定のセクションでは、 コンポーネントのスタブは **seam/resource/remoting/interface.js** タグまたは **s:remote** タグのいずれかを使用してペー

ジにインポートされます。

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?customerAction">
</script>
```

```
<s:remote include="customerAction"/>
```

このスクリプトを含ませることでコンポーネントのインターフェース定義に加えて、 コンポーネントのメソッド実行に必要なその他のコンポーネントやタイプが生成され、 リモーティングフレームワークで使用できるようになります。

生成できるスタブは **実行可能** スタブと **タイプ** スタブの 2 種類です。 実行可能スタブは動作を持ち、セッション Bean コンポーネントに対してメソッドを実行します。 タイプスタブは状態を保持し、 パラメータとして渡されるか、結果として返されることができるタイプを表します。

生成されるスタブの種類は Seam コンポーネントのタイプによります。 コンポーネントがセッション Bean なら実行可能スタブが生成されます。 エンティティや JavaBean となる場合にはスタブのタイプが生成されます。 ただし、 コンポーネントが JavaBean でそのメソッドのいずれにも **@WebRemote** アノテーションが付く場合、 実行可能なスタブが生成されます。 これにより、 セッション Bean にアクセスできない非 EJB 環境で JavaBean コンポーネントのメソッドを呼び出すことができるようになります。

24.5. コンテキスト

Seam Remoting コンテキストにはリモート要求または応答のサイクルの一部として送受信される追加情報が含まれます。 現段階では対話 ID だけしか含んでいませんが、将来拡張される可能性があります。

24.5.1. 対話 ID の設定と読み込み

対話スコープ内でリモート呼び出しを使用する予定である場合は、 Seam Remoting コンテキスト内の対話 ID の読み込みと設定が行える必要があります。リモート要求の後に対話 ID を読み込む場合には、**`Seam.Remoting.getContext().getConversationId()`** を呼び出します。 要求の前に対話 ID を設定する場合には、 **`Seam.Remoting.getContext().setConversationId()`** を呼び出します。

対話 ID が明示的に **`Seam.Remoting.getContext().setConversationId()`** で設定されない場合、 リモート呼び出しによって返される最初の有効な対話 ID が自動的に割り当てられます。 ページ内で複数の対話を使用する場合は、 それぞれの呼び出しの前に対話 ID を明示的に設定する必要があるかもしれません。1 つの対話だけを使用する場合は、明示的に ID を設定する必要はありません。

24.5.2. 現在の対話スコープ内のリモート呼び出し

現在のビューの対話スコープ内でリモート呼び出しを行う必要がある場合があります。 これを行うにはリモート呼び出しを行う前に明示的に対話 ID をビューのそれに設定する必要があります。 次の JavaScript はリモート呼び出しに使用されている対話 ID を現在のビューの対話 ID に設定します。

```
Seam.Remoting.getContext().setConversationId( #{conversation.id} );
```

24.6. バッチ要求

Seam Remoting により、1 つの要求で複数のコンポーネント呼び出しが実行できるようになります。ネットワークトラフィックを低減する必要がある場合にこの機能を使用することをお勧めします。

Seam.Remoting.startBatch() メソッドは新しいバッチを起動します。 バッチ起動後に実行されるコンポーネント呼び出しはすべて待ち行列に入れられ、 即時送信は行われません。 必要とされるすべてのコンポーネント呼び出しがバッチに追加されると、 **Seam.Remoting.executeBatch()** メソッドは待ち行列にあるすべての呼び出しを含む単一の要求をサーバーに送信し、 その呼び出しは順番に実行されることになります。 呼び出しが実行されるとすべての戻り値を含む単一の応答がクライアントに返され、 コールバック機能が実行と同じ順番で起動されます。

新しいバッチを起動したけれど送信しないことにした場合、 **Seam.Remoting.cancelBatch()** メソッドは待ち行列に入れられたすべての呼び出しを破棄してそのバッチモードを終了します。

バッチが利用されているサンプルは、 **/examples/remoting/chatroom** を参照ください。

24.7. データタイプの取り扱い

24.7.1. プリミティブ / 基本タイプ

本項では基本データタイプのサポートについて説明します。 サーバー側ではこれらの値は一般的にそのプリミティブタイプか該当のラッパークラスと互換性があります。

24.7.1.1. String 型

String パラメータ値を設定するには、 Javascript String オブジェクトを使用します。

24.7.1.2. Number 型

Java でサポートされているすべての数値タイプに対応します。 クライアント側では数値は常にその String 表現としてシリアライズされます。 サーバー側で適切な目的タイプに変換されます。 プリミティブまたはラッパーいずれかのタイプへの変換は、 **Byte**、 **Double**、 **Float**、 **Integer**、 **Long**、 **Short** の各タイプに対してサポートされます。

24.7.1.3. Boolean 型

Boolean はクライアント側では Javascript の Boolean 値で表現され、 サーバー側では Java Boolean で表現されます。

24.7.2. JavaBeans

一般的に Seam エンティティ、 JavaBean コンポーネント、 または non-component クラスのいずれかになります。 適切なメソッドを使って Seam コンポーネントの新しいオブジェクトのインスタンス **Seam.Component.newInstance()**、またはその他の場合は **Seam.Remoting.createType()** の新しいインスタンスを作成します。

これら 2 つのメソッドのどちらかによって生成されるオブジェクトだけがパラメータ値として使用されるはずで、このパラメータは既に存在している有効なタイプのひとつにはなりません。 以下のように厳密にパラメータタイプを決定できないコンポーネントメソッドがあるかもしれません。

```
@Name("myAction")
public class MyAction implements MyActionLocal {
    public void doSomethingWithObject(Object obj) {
        // code
    }
}
```

この場合、 **myAction** のインターフェースはそのいずれのメソッドからも直接参照されないため **myWidget** を含みません。 したがって **MyWidget** コンポーネントのインスタンスを明示的にインポートしない限りそれを渡すことができません。

```
<s:remote include="myAction,myWidget"/>
```

これにより **myWidget** オブジェクトが **Seam.Component.newInstance("myWidget")** で作成されるようになり、 **myAction.doSomethingWithObject()** に渡されます。

24.7.3. 日付と時刻

日付の値はミリ秒単位で正確な String 表示にシリアライズされます。 クライアント側では Javascript Date オブジェクトを使って日付値と動作します。 サーバー側では **java.util.Date** (または **java.sql.Date** や **java.sql.Timestamp** などの下位クラス) を使用します。

24.7.4. Enum

クライアント側では、 Enum は String と同様に扱われます。 Enum パラメータの値を設定する場合は enum の String 表現を使います。 次のコンポーネントを例として参照してください。

```
@Name("paintAction")
public class paintAction implements paintLocal {
    public enum Color {red, green, blue, yellow, orange, purple};
    public void paint(Color color) {
        // code
    }
}
```

paint() メソッドを **red** の色を使って呼び出すには、 String リテラルとしてパラメータ値を渡します。

```
Seam.Component.getInstance("paintAction").paint("red");
```

逆もまた同じことが言えます。 つまり、 コンポーネントメソッドが enum パラメータを返す場合 (または返されるオブジェクトグラフのどこかに enum フィールドを含む場合)、 クライアント側では String として表示されます。

24.7.5. コレクション

24.7.5.1. Bag

Bag は array、 collection、 list、 set などすべてのコレクションタイプを対象としますが map は対象外です。 map については次項を参照してください。 これらは呼び出される場合、返される場合いずれでも Javascript 配列でクライアント側に実装されます。 サーバー側にあるこのリモーティングフレームワークでは bag をコンポーネントメソッド呼び出しの適切なタイプに変換することが可能です。

24.7.5.2. Map

Seam Remoting フレームワークでは、ネイティブのサポートがない JavaScript のシンプルな map サポートが提供されます。 リモート呼び出しに対してパラメータとして使用できる map を作成するには、新しい **Seam.Remoting.Map** オブジェクトを作成します。

```
var map = new Seam.Remoting.Map();
```

この JavaScript 実装では Map と動作することを目的とした基本的なメソッド、 **size()**、 **isEmpty()**、 **keySet()**、 **values()**、 **get(key)**、 **put(key, value)**、 **remove(key)**、 **contains(key)** を提供します。 それぞれのメソッドは同じ名前の Java メソッドと同等です。 メソッドが **keySet()** および **values()** でコレクションを返すと、 そのキーまたは値オブジェクトを含む Javascript array オブジェクトが返されます。

24.8. デバッグ機能

バグの追跡を支援する目的でデバッグモードを有効にすることができます。 ポップアップウィンドウ内でクライアントとサーバーの間で送信されるすべてのパケットの内容を表示します。 デバッグモードを有効にするには次のいずれかを行います。 JavaScript 内で **setDebug()** メソッドを実行する方法は次のとおりです。

```
Seam.Remoting.setDebug(true);
```

components.xml で設定を行う方法は以下のとおりです。

```
<remoting:remoting debug="true"/>
```

デバッグモードをオフにするには **setDebug(false)** を呼び出します。 独自のメッセージをデバッグログに書き込みたい場合は、 **Seam.Remoting.log(message)** を呼び出します。

24.9. 例外の処理

リモートコンポーネントメソッドを呼び出すときに、コンポーネント呼び出し中に例外が発生した場合には、例外ハンドラを指定して応答を処理することができます。 例外ハンドラ機能を指定するには、それへの参照を JavaScript 内のコールバックパラメータの後ろに含ませます。

```
var callback = function(result) {
    alert(result);
};
var exceptionHandler = function(ex) {
    alert("An exception occurred: " + ex.getMessage());
};
Seam.Component.getInstance("helloAction")
    .sayHello(name, callback, exceptionHandler);
```

定義したコールバックハンドラがない場合にはその場所に **null** を指定しなければなりません。

```
var exceptionHandler = function(ex) {
    alert("An exception occurred: " + ex.getMessage());
};
Seam.Component.getInstance("helloAction")
    .sayHello(name, null, exceptionHandler);
```

例外ハンドラに渡される例外オブジェクトは、ひとつのメソッド **getMessage()** を公開し、 **@WebRemote** メソッドで送出される例外に属する例外メッセージを返します。

24.10. ロード中のメッセージ

画面の右上隅に出てくるデフォルトのロード中メッセージのカスタムな表示を修正、 定義または削除することもできます。

24.10.1. メッセージの変更

デフォルトの「Please Wait...」というメッセージを変更するには、
Seam.Remoting.loadingMessage の値を設定します。

```
Seam.Remoting.loadingMessage = "Loading...";
```

24.10.2. ロード中のメッセージの非表示

ロード中のメッセージを完全に表示させないようにするには、 **displayLoadingMessage()** および **hideLoadingMessage()** の実装を何も行わない機能で上書きします。

```
// don't display the loading indicator
Seam.Remoting.displayLoadingMessage = function() {};
Seam.Remoting.hideLoadingMessage = function() {};
```

24.10.3. カスタムのロード中インジケータ

ロード中インジケータを上書きして動画のアイコンやその他好きなもの何でも表示させることができます。 **displayLoadingMessage()** と **hideLoadingMessage()** の各メッセージを独自の実装で上書きしてこれを行います。

```
Seam.Remoting.displayLoadingMessage = function() {
    // Write code here to display the indicator
};
Seam.Remoting.hideLoadingMessage = function() {
    // Write code here to hide the indicator
};
```

24.11. 返されるデータの制御

リモートメソッドが実行されると、 その結果は XML レスポンスにシリアライズされ、クライアントに返されます。次にこの応答はクライアントにより JavaScript オブジェクトにアンマーシャルされます。 他のオブジェクトへの参照を含む複雑なタイプの場合 (JavaBeans など)、参照されるオブジェクトもすべて応答の一部としてシリアライズされます。 これらのオブジェクトは他のオブジェクトを参照することができ、それらはまた別のオブジェクトを参照できるといった具合になります。返されるデータを制御しないままにしておくと、このオブジェクト「グラフ」は非常に膨大になる可能性があります。

このため、クライアントに対して機密情報が公開されないようにするために、 Seam Remoting はリモートメソッドの **@WebRemote** アノテーションの **exclude** フィールドを指定することでそのオブジェクトグラフを制約できます。このフィールドはドット (「.」) 表記を使って指定される 1 つ以上のパスを含む String 配列を受け取ります。 リモートメソッドを呼び出すと、 これらのパスと一致する結果のオブジェクトグラフにあるオブジェクトがシリアライズされる結果パケットから除外されます。

すべての例は次の **Widget** クラスに基づいています。

```
@Name("widget")
public class Widget {
```

```
    private String value;
    private String secret;
    private Widget child;
    private Map&lt;String,Widget&gt; widgetMap;
    private List&lt;Widget&gt; widgetList;

    // getters and setters for all fields
```

24.11.1. 通常のフィールドの制約

リモートメソッドが **Widget** のインスタンスを返すけれど**secret** フィールドには機密情報が含まれているため公開したくない場合は、次のように制約します。

```
@WebRemote(exclude = {"secret"})
public Widget getWidget();
```

値「**secret**」は返されるオブジェクトの **secret** フィールドを参照します。

ここで、返される **Widget** 値には **child** フィールドがあり、これも **Widget** になる点に注意してください。フィールドではなくこの **child** の **secret** 値を隠したい場合は、ドット表記を使用して結果となるオブジェクトグラフ内のこのフィールドのパスを指定することができます。

```
@WebRemote(exclude = {"child.secret"})
public Widget getWidget();
```

24.11.2. Map とコレクションの制約

オブジェクトグラフ内のオブジェクトは **Map** またはコレクション (**List**、 **Set**、 **Array** など) 内にも存在することができます。コレクションはその他のフィールドと同様に扱えます。 たとえば、**Widget** の **widgetList** フィールド内に他の **Widget** 一覧が含まれていて、 この一覧の**Widget** の **secret** フィールドを次のような表記で制約するとします。

```
@WebRemote(exclude = {"widgetList.secret"})
public Widget getWidget();
```

Map のキーまたは値を制約する場合の表記は少し異なります。**Map** のフィールド名の後ろに **[key]** を付け加えると **Map** のキーオブジェクト値を制約し、 **[value]** の場合は値オブジェクトの値を制約します。 次の例では **widgetMap** フィールドの値に制約された **secret** フィールドを持たせる方法を示しています。

```
@WebRemote(exclude = {"widgetMap[value].secret"})
public Widget getWidget();
```

24.11.3. 特定タイプのオブジェクトの制約

角括弧を使ってオブジェクトグラフ内のその場所に関係なくオブジェクトタイプのフィールドを制約することができます。 オブジェクトが Seam コンポーネントの場合はコンポーネント名を使用し、そうでない場合は完全修飾クラス名を使用します。

```
@WebRemote(exclude = {"[widget].secret"})
public Widget getWidget();
```

24.11.4. 制約同士の組み合わせ

制約同士はオブジェクトグラフ内で複数のパスからオブジェクトをフィルタするために組み合わせることもできます。

```
@WebRemote(exclude = {"widgetList.secret", "widgetMap[value].secret"})
public Widget getWidget();
```

24.12. トランザクション的な要求

デフォルトではリモート要求の間はトランザクションはアクティブになりません。 リモート要求中にデータベースを更新したい場合は **@Transactional** アノテーションを **@WebRemote** メソッドに付与する必要があります。

```
@WebRemote
@Transactional(TransactionPropagationType.REQUIRED)
    public void updateOrder(Order order) {
    entityManager.merge(order);
}
```

24.13. JMS MESSAGING

Seam Remoting は JMS Messaging に対して実験的に対応しています。 本項では現在実装されている JMS サポートについて記載していますが、 今後変更される可能性があるので注意してください。 現在この機能を実稼働環境下で使用することは推奨されていません。

24.13.1. 設定

JMS トピックをサブスクライブする前に、 まず Seam Remoting でサブスクライブさせることができるトピック一覧を設定する必要があります。 **seam.properties**、 **web.xml** または **components.xml** の **org.jboss.seam.remoting.messaging.subscriptionRegistry.allowedTopics** 配下にあるトピックを一覧表示させます。

```
<remoting:remoting poll-timeout="5" poll-interval="1"/>
```

24.13.2. JMS Topic のサブスクライブ

次の例では JMS Topic へのサブスクライブ方法を示しています。

```
function subscriptionCallback(message) {
    if (message instanceof Seam.Remoting.TextMessage)
        alert("Received message: " + message.getText());
}
Seam.Remoting.subscribe("topicName", subscriptionCallback);
```

Seam.Remoting.subscribe() メソッドは 2 つのパラメータを受け取ります。1 つ目はサブスクライブする JMS Topic 名であり、2 つ目はメッセージが受け取られると呼び出すコールバック機能です。

サポートされているメッセージは 2 種類で、 テキストメッセージとオブジェクトメッセージです。 コールバック機能に渡されるメッセージタイプのテストをする場合は、 **instanceof** 演算子を使ってメッセージが **Seam.Remoting.TextMessage** なのか **Seam.Remoting.ObjectMessage** かをテストすることができます。 **TextMessage** はその **text** フィールドにテキスト値を含みます (オブジェク

トの **getText()** メソッドを呼び出してこの値をフェッチすることもできます)。**ObjectMessage** はその **value** フィールドにオブジェクト値を含みます (**getValue()** メソッドを呼び出してこの値をフェッチすることもできます)。

24.13.3. トピックのサブスクライブの中止

トピックのサブスクライブを中止するには、 **Seam.Remoting.unsubscribe()** を呼び出してトピック名を渡します。

```
Seam.Remoting.unsubscribe("topicName");
```

24.13.4. ポーリングプロセスの調整

ポーリングは 2 種類のパラメータで制御および修正が可能です。

Seam.Remoting.pollInterval は新しいメッセージに対して後続ポーリングが発生する間隔を制御します。 このパラメータは秒単位で表現され、 デフォルト設定は **10** です。

Seam.Remoting.pollTimeout も秒単位で表現されます。 サーバーへの要求がタイムアウトして空白の応答を送信するまでの新しいメッセージを待機する時間を制御します。 デフォルトは **0** 秒で、サーバーがポーリングされると配信できるメッセージがない場合は空白の応答が直ちに返されます。

pollTimeout 値を高く設定する場合は注意が必要です。メッセージを待機する必要がある各要求は、メッセージが受信されるまでまたはその要求がタイムアウトするまでサーバースレッドを使用します。こうした要求が同時に多数発生すると、 大量のサーバースレッドが使用される結果となります。

これらのオプションは **components.xml** で設定することを推奨しますが、 必要に応じて JavaScript で上書きすることができます。 次の例ではよりアグレッシブなポーリングメソッドを示しています。これらのパラメータをご使用のアプリケーションに適切な値に設定してください。

components.xml での設定

```
<remoting:remoting poll-timeout="5" poll-interval="1"/>
```

Java での設定

```
// Only wait 1 second between receiving a poll response and sending
// the next poll request.
Seam.Remoting.pollInterval = 1;
// Wait up to 5 seconds on the server for new messages
Seam.Remoting.pollTimeout = 5;
```

第25章 SEAM と GOOGLE WEB TOOLKIT

警告

Technology Preview の機能は Red Hat サブスクリプションレベルアグリーメント (SLA) では完全に対応していません。また、機能的に完全ではない場合があるため実稼働での使用を目的としていません。ただし、こうした機能により今後の新製品開発に早くアクセスすることができるため、開発段階でお客様が機能性をテストしたり、フィードバックをお寄せいただくことができます。Red Hat は 今後強化された Technology Preview の機能を一般的に利用できるよう検討しており、商業的に合理的な範囲でお客様がこうした機能を使用しているときに直面するすべての問題の解決に向けて努力します。

動的な AJAX (Asynchronous Java and XML) アプリケーションを Google Web Toolkit (GWT) を使って開発したいときのために、 Seam は GWT ウィジェットが直接 Seam コンポーネントと連携できる統合レイヤを備えています。

本項では、 GWT Tools に関しては熟知されていることを前提とし、 Seam の統合についてのみ焦点を絞って説明します。 詳しくは http://code.google.com/webtoolkit/ を参照ください。

25.1. 設定

Seam アプリケーションで GWT を使う場合に特に設定に変更を加える必要はありません。必要なことは Seam Resource Servlet をインストールするだけです。 詳細は *28章 Seam の設定と Seam アプリケーションのパッケージング* を参照してください。

25.2. コンポーネントの準備

GWT で Seam コンポーネントが呼び出されるよう準備するには、 まず呼び出したいメソッドの同期および非同期サービスの両インターフェースを作成しなければなりません。 両方のインターフェースとも GWT インターフェース **com.google.gwt.user.client.rpc.RemoteService** を拡張するはずです。

```
public interface MyService extends RemoteService {
  public String askIt(String question);
}
```

非同期インターフェースは宣言するメソッドごとに **AsyncCallback** パラメータが追加されている点以外はまったく同じになるはずです。

```
public interface MyServiceAsync extends RemoteService {
  public void askIt(String question, AsyncCallback callback);
}
```

非同期インターフェース (例では **MyServiceAsync**) は GWT で実装されるので、絶対に直接実装しないでください。

次のステップは同期インターフェースを実装する Seam コンポーネントの作成です。

```
@Name("org.jboss.seam.example.remoting.gwt.client.MyService")
public class ServiceImpl implements MyService {

  @WebRemote
  public String askIt(String question) {

    if (!validate(question)) {
      throw new IllegalStateException("Hey, this shouldn't happen, " +
                                      "I checked on the client, but " +
                                      "it's always good to double
check.");
    }
    return "42. Its the real question that you seek now.";
  }

  public boolean validate(String q) {
    ValidationUtility util = new ValidationUtility();
    return util.isValid(q);
  }
}
```

Seam コンポーネント名は GWT クライアントインターフェースの完全修飾名と一致しなければなりません (上記参照)。 一致しないと、クライアントが GWT 呼び出しを行っても Seam Resource Servlet はそれを見つけることができません。 GWT がアクセスできるようにするメソッドには **@WebRemote** アノテーションを付与する必要があります。

25.3. GWT ウィジェットを SEAM コンポーネントにつなげる

次に、 コンポーネントに非同期インターフェースを返すメソッドを記述します。 このメソッドはウィジェットクラス内にあり、 非同期クライアントのスタブへの参照を取得するためウィジェットにより使用されます。

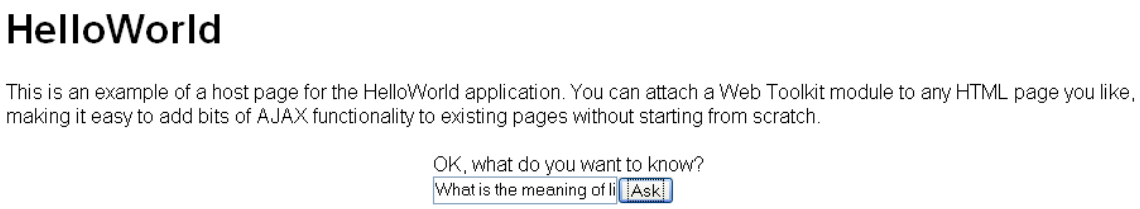
```
private MyServiceAsync getService() {
  String endpointURL = GWT.getModuleBaseURL() + "seam/resource/gwt";

  MyServiceAsync svc = (MyServiceAsync) GWT.create(MyService.class);
  ((ServiceDefTarget) svc).setServiceEntryPoint(endpointURL);
  return svc;
}
```

最後にクライアントスタブでメソッドを呼び出すウィジェットのコードを記述します。 次の例ではラベル、 テキスト入力フィールド、 ボタンで構成されるシンプルなユーザーインターフェースを作成します。

```
public class AskQuestionWidget extends Composite {
  private AbsolutePanel panel = new AbsolutePanel();

  public AskQuestionWidget() {
    Label lbl = new Label("OK, what do you want to know?");
    panel.add(lbl);
    final TextBox box = new TextBox();
    box.setText("What is the meaning of life?");
    panel.add(box);
    Button ok = new Button("Ask");
```

```
    ok.addClickListener(new ClickListener() {

      public void onClick(Widget w) {
        ValidationUtility valid = new ValidationUtility();
        if (!valid.isValid(box.getText())) {
          Window.alert("A question has to end with a '?'");
        } else {
          askServer(box.getText());
        }
      }
    });
    panel.add(ok);

    initWidget(panel);
  }

  private void askServer(String text) {
    getService().askIt(text, new AsyncCallback() {
    public void onFailure(Throwable t) {
      Window.alert(t.getMessage());
    }

    public void onSuccess(Object data) {
      Window.alert((String) data);
    }
  });
}

...
```

ボタンをクリックすると **askServer()** メソッドが呼び出され、 入力テキストの内容を渡します。 この例では、 入力値が正しい質問であるかも検証します。 **askServer()** メソッドは非同期クライアントスタブへの参照を取得し (**getService()** メソッドで返される)、 **askIt()** メソッドを呼び出します。 その結果はアラートウィンドウに表示されます (または呼び出しが失敗するとエラーメッセージが表示されます)。



この例の完全なコードは Seam ディストリビューションの **examples/remoting/gwt** ディレクトリにあります。

25.4. GWT と ANT ターゲット

GWT アプリケーションをデプロイするためには JavaScript に対してもコンパイルを行う必要があります。これによりコードを圧縮し、難読化します。コマンドラインの代わりに Ant ユーティリティや GWT で提供される GUI ユーティリティを使うことができます。 これらを使うためには Ant クラスパスに Ant タスクの **JAR** とダウンロードした GWTが必要です。

次を Ant ファイルの冒頭付近に配置します。

```
<taskdef uri="antlib:de.samaflost.gwttasks"
         resource="de/samaflost/gwttasks/antlib.xml"
         classpath="./lib/gwttasks.jar"/>
<property file="build.properties"/>
```

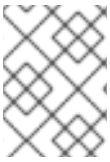
以下を含む **build.properties** ファイルを作成します。

```
gwt.home=/gwt_home_dir
```

GWT がインストールされたディレクトリを指していなければなりません。 次にターゲットを作成します。

```
<!-- the following are are handy utilities for doing GWT development.
     To use GWT, you will of course need to download GWT seperately -->

<target name="gwt-compile">
  <!-- in this case, we are "re homing" the gwt generated stuff, so
       in this case we can only have one GWT module - we are doing this
       deliberately to keep the URL short -->
  <delete>
    <fileset dir="view"/>
  </delete>
  <gwt:compile outDir="build/gwt"
               gwtHome="${gwt.home}"
               classBase="${gwt.module.name}"
               sourceclasspath="src"/>
    <copy todir="view">
      <fileset dir="build/gwt/${gwt.module.name}"/>
    </copy>
</target>
```

呼び出されると、 このターゲットは GWT アプリケーションをコンパイルしてそれを指定ディレクトリにコピーします (WAR の **webapp** セクションの場合が多い)。



注記

gwt-compile で生成されたコードは絶対に編集しないでください。 編集が必要な場合には GWT ソースディレクトリ内で行ってください。

GWT でアプリケーション開発を行う予定の場合は、GWT に含まれているホストモードブラウザの使用を強く推奨します。

第26章 SPRING FRAMEWORK 統合

Spring Framework は Seam inversion-of-control (IoC) モジュールの一部です。 これにより Spring ベースのプロジェクトを Seam に移行しやすくなり、 Spring アプリケーションで Seam の主要な機能となる対話や高度な永続コンテキスト管理を利用できます。

注記

Spring 統合コードは **jboss-seam-ioc** ライブラリに含まれています。 これは本章に記載されているすべての Seam と Spring の統合技術に必要となる依存ライブラリです。

Spring に対し Seam は次のような機能を提供します。

- Seam コンポーネントを Spring Bean にインジェクトする
- Spring Bean を Seam コンポーネントにインジェクトする
- Spring Bean を Seam コンポーネントに変換する
- Spring Bean を Seam コンテキストに配置できるようにする
- Seam コンポーネントで Spring WebApplicationContext を起動できるようにする
- Seam ベースのアプリケーションで Spring PlatformTransactionManagement の使用をサポートする
- Spring の **OpenEntityManagerInViewFilter** および **OpenSessionInViewFilter** の代替として Seam 管理の使用をサポートする
- Spring **TaskExecutors** で **@Asynchronous** 呼び出しの支援をサポートする

26.1. SEAM コンポーネントを SPRING BEAN にインジェクトする

Seam コンポーネントのインスタンスを Spring Bean に **<seam:instance/>** 名前空間ハンドラを使用してインジェクトします。 Seam 名前空間ハンドラを有効にするには、 Seam 名前空間をまず Spring Bean の定義ファイルに追加しなければなりません。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:seam="http://jboss.com/products/seam/spring-seam"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
         "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-
2.0.xsd
          http://jboss.com/products/seam/spring-seam
          http://jboss.com/products/seam/spring-seam-2.2.xsd">
```

これで以下のように、いずれの Seam コンポーネントもあらゆる Spring Bean にインジェクト可能となりました。

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty">
    <seam:instance name="someComponent"/>
```

```
  </property>
</bean>
```

コンポーネント名の代わりに EL 式を使用することができます。

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty">
    <seam:instance name="#{someExpression}"/>
  </property>
</bean>
```

以下のようにして Spring Bean ID を使って Seam コンポーネントインスタンスを Spring Bean にインジェクトできます。

```
<seam:instance name="someComponent" id="someSeamComponentInstance"/>

<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty" ref="someSeamComponentInstance">
</bean>
```

ただし、 Spring は Seam と異なり複数のコンテキストではステートフルコンポーネントモデルに対応するようには設計されていませんでした。 Spring インジェクションはメソッド呼出し時には発生しませんが、 Spring Bean がインスタンス化された時に発生します。

Bean がインスタンス化されるときに使用可能なインスタンスは Bean の寿命全体に渡り使用されます。 Seam の対話スコープのコンポーネントインスタンスを直接シングルトンの Spring Bean にインジェクトするとします。 このシングルトンは対話が終了した後もしばらくの間同じインスタンスへの参照を保持します。これは **スコープインピーダンス** と呼ばれます。

呼び出しがシステムを流れるように Seam バイジェクションは自然にスコープインピーダンスを維持します。 Spring では Seam コンポーネントのプロキシをインジェクトし、そのプロキシが呼び出された場合に参照を解決しなければなりません。

<seam:instance/> タグで自動的に Seam コンポーネントをプロキシできます。

```
<seam:instance id="seamManagedEM"
               name="someManagedEMComponent"
               proxy="true"/>

<bean id="someSpringBean" class="SomeSpringBeanClass">
  <property name="entityManager" ref="seamManagedEM">
</bean>
```

上記の例では Spring Bean から Seam 管理永続コンテキストを使用する方法の例を示しています。 Spring **OpenEntityManagerInView** フィルタの代替として Seam 管理永続コンテキストを使用するためのより堅牢な方法については、「Spring での Seam 管理永続コンテキストの使用」の項を参照してください。

26.2. SPRING BEAN を SEAM コンポーネントにインジェクトする

EL 式を使用するか Spring Beam を Seam コンポーネントにすることで Spring Bean を Seam コンポーネントのインスタンスにインジェクトすることができます。

最も容易な方法は EL を使って Spring Bean にアクセスすることです。

Spring の **DelegatingVariableResolver** は Spring の JavaServer Faces (JSF) との統合に役立ちます。 この **VariableResolver** は Bean ID を持つ EL を使って Spring Bean を JSF に対して利用できるようにします。 **DelegatingVariableResolver** を **faces-config.xml** に追加する必要があります。

```
<application>
  <variable-resolver>
    org.springframework.web.jsf.DelegatingVariableResolver
  </variable-resolver>
</application>
```

これで **@In** を使って Spring Bean をインジェクトできるようになります。

```
@In("#{bookingService}")
private BookingService bookingService;
```

Spring Bean はインジェクションに限定されません。プロセスとページフロー定義、 ワーキングメモリのアサーションなど、Seam で EL 式が使用されていれば常に Spring Bean を使用することができます。

26.3. SPRING BEAN を SEAM コンポーネントにする

<seam:component/> 名前空間ハンドラを使用すると、 あらゆる Spring Bean を Seam コンポーネントに変換することができます。 Seam コンポーネントにしたい Bean の宣言に **<seam:component/>** タグを追加するだけです。

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <seam:component/>
</bean>
```

デフォルトでは、 **<seam:component/>** は Bean 定義で与えられるクラスと名前を付けてステートレスな Seam コンポーネントを作成します。 ときおり、 **FactoryBean** が使用される場合など、 Spring Bean のクラス が Bean 定義に表示されるクラスとは異なることがあります。 このような場合は **class** を明示的に指定してください。 また、 名前付けに競合の可能性がある場合は Seam コンポーネント名も明示的に指定してください。

Spring Bean を特定の Seam スコープ内で管理したい場合は **<seam:component/>** の **scope** 属性を使用します。 指定される Seam スコープが **STATELESS** ではない場合、 Spring Bean を **prototype** にスコープする必要があります。既存の Spring Bean は通常基本的にステートレスな特徴を持っているので、 この属性は通常は不要です。

26.4. SEAM スコープの SPRING BEAN

Seam 統合パッケージでは Seam のコンテキストを Spring 2.0 スタイル **カスタムなスコープ**として使用することもできます。 これによりいずれの Seam コンテキスト内でもあらゆる Spring Bean を宣言することができます。ただし、 Spring のコンポーネントモデルはステートフル性に対応するようには設計されていなかったため、 この機能を使用する場合は十分に気を付けてください。 特に、 セッションスコープや対話スコープの Spring Bean のクラスタ化には問題があるため、 広いスコープの Bean やコンポーネントを狭いスコープの Bean にインジェクトする場合は注意が必要です。

Spring Bean ファクトリの設定で **<seam:configure-scopes/>** を指定し、 すべての Seam スコープがカスタムスコープとして Spring Bean に利用できるようにします。 Spring Bean を特定の Seam スコープに関連付けるには、 Bean 定義の **scope** 属性で目的のスコープを指定します。

```
<!-- Only needs to be specified once per bean factory-->
<seam:configure-scopes/>

...

<bean id="someSpringBean" class="SomeSpringBeanClass"
      scope="seam.CONVERSATION"/>
```

configure-scopes 定義内の **prefix** 属性を指定することによって、 スコープ名のプレフィックスを変更することができます (デフォルトのプレフィックスは **seam.** です)。

デフォルトではこの方法で登録される Spring コンポーネントのインスタンスは **@In** を使って参照される場合に自動的に作成されません。 インスタンスを自動作成させるには、インジェクションポイントで **@In(create=true)** を指定するか (特定の Bean を自動作成するため)、 **configure-scopes** の **default-auto-create** 属性を使って Seam スコープの Spring Bean がすべて自動作成されるようにします。

後者の方法では Seam スコープの Spring Bean を **<seam:instance/>** を使わずに他の Spring Bean にインジェクトすることができます。 ただし、スコープインピーダンスには十分注意する必要があります。 一般的には Bean 定義内で **<aop:scoped-proxy/>** を指定しますが、 Seam スコープの Spring Bean は **<aop:scoped-proxy/>** との互換性がありません。 したがって、Seam スコープ Spring Bean をシングルトンにインジェクトする場合は **<seam:instance/>** を使用してください。

```
<bean id="someSpringBean" class="SomeSpringBeanClass"
scope="seam.CONVERSATION"/>

...

<bean id="someSingleton">
  <property name="someSeamScopedSpringBean">
    <seam:instance name="someSpringBean" proxy="true"/>
  </property>
</bean>
```

26.5. SPRING の PLATFORMTRANSACTIONMANAGEMENT の使用

Spring の拡張可能なトランザクション管理は Java Persistence API (JPA)、 Hibernate、 Java Data Objects (JDO)、 Java Transaction API (JTA) などの多くのトランザクション API に対応します。 また、 ネストしたトランザクションなどの多くの高度な機能にも対応しています。 Spring は Websphere や Weblogic などの多くのアプリケーションサーバーの TransactionManagers との強い統合を実現し、 **REQUIRES_NEW** や **NOT_SUPPORTED** などの完全 Java EE トランザクション伝播のルールにも対応します。 詳細は Spring のドキュメント を参照してください。

Seam が Spring のトランザクションを使用するよう設定するには、 **SpringTransaction** コンポーネントを以下のように有効にします。

```
<spring:spring-transaction
        platform-transaction-manager="#{transactionManager}"/>
```

spring:spring-transaction コンポーネントは同期のコールバックに Spring トランザクション同期の機能を利用します。

26.6. SPRING での SEAM 管理永続コンテキストの使用

Seam の最もパワフルな機能として、その対話スコープや対話が生きている間 **EntityManager** をオープンにしておくという機能があります。 これによりエンティティの分離や再併合に関連する多くの問題が解消され、**LazyInitializationException** の発生を軽減できます。 Spring は単一の Web 要求 (**OpenEntityManagerInViewFilter**) のスコープを越えて永続コンテキストを管理する方法は提供していません。

Spring 開発者が Spring 提供の JPA ツールで Seam 管理永続コンテキストにアクセスできるようにすることで、Seam は対話スコープの永続コンテキストの機能を Spring アプリケーションにもたらしました (**PersistenceAnnotationBeanPostProcessor**、 **JpaTemplate** など)。

この統合により次のような機能を実現します。

- Spring 提供のツールを使った Seam 管理永続コンテキストへの透過的なアクセス
- Web 要求以外での Seam 対話スコープ永続コンテキストへのアクセス (非同期の Quartz ジョブなど)
- Spring 管理トランザクションで Seam 管理永続コンテキストを使用する機能 (手作業による永続コンテキストのフラッシュが必要)

Spring の永続コンテキスト伝播モデルは **EntityManagerFactory** ごとに 1 つのオープン **EntityManager** しか可能でないため、 Seam 統合は **EntityManagerFactory** を Seam 管理永続コンテキストでラップすることで動作します。

```
<bean id="seamEntityManagerFactory"
    class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
  <property name="persistenceContextName" value="entityManager"/>
</bean>
```

persistenceContextName は Seam 管理永続コンテキストコンポーネントの名前です。 デフォルトではこの **EntityManagerFactory** には Seam コンポーネント名と同等の **unitName** があります。この場合は **entityManager** です。 別の **unitName** を与えたい場合は次のようにして **persistenceUnitName** を与えることができます。

```
<bean id="seamEntityManagerFactory"
    class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
  <property name="persistenceContextName" value="entityManager"/>
  <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
```

これでこの **EntityManagerFactory** をいずれの Spring 提供のツールでも使用することができます。この場合は、Spring の **PersistenceAnnotationBeanPostProcessor** を Spring で使用するのと同じように使用することができます。

```
<bean class="org.springframework.orm.jpa.support
             .PersistenceAnnotationBeanPostProcessor"/>
```

実際の **EntityManagerFactory** を Spring で定義するが Seam 管理永続コンテキストを使用したい場合は、 **defaultPersistenceUnitName** プロパティを指定してデフォルトで使用したい

persistenctUnitName を **PersistenceAnnotationBeanPostProcessor** に指示することができます。

applicationContext.xml は次に似たようなものになります。

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="bookingDatabase"/>
</bean>
<bean id="seamEntityManagerFactory"
    class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
  <property name="persistenceContextName" value="entityManager"/>
  <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
<bean class="org.springframework.orm.jpa
             .support.PersistenceAnnotationBeanPostProcessor">
  <property name="defaultPersistenceUnitName"
            value="bookingDatabase:extended"/>
</bean>
```

component.xml は次に似たようなものになります。

```
<persistence:managed-persistence-context name="entityManager"
    auto-create="true" entity-manager-factory="#{entityManagerFactory}"/>
```

JpaTemplate および **JpaDaoSupport** は Spring ベースの永続コンテキストと通常の Seam 管理永続コンテキストではまったく同じ構成になります。

```
<bean id="bookingService"
      class="org.jboss.seam.example.spring.BookingService">
  <property name="entityManagerFactory" ref="seamEntityManagerFactory"/>
</bean>
```

26.7. SPRING での SEAM 管理 HIBERNATE セッションの使用

Seam への Spring 統合により Spring のツールを使った Seam 管理 Hibernate セッションへの完全アクセスに対応することもできます。 この統合は JPA 統合に非常によく似ています。 詳細は 「Spring での Seam 管理永続コンテキストの使用」 を参照してください。

Spring の伝播モデルは **EntityManagerFactory** ごとの 1 つのオープンな **EntityManager** のみ Spring ツールに対して利用できるようにします。このため、 Seam はプロキシの **SessionFactory** を Seam 管理の Hibernate セッションコンテキストでラップすることで統合を行います。

```
<bean id="seamSessionFactory"
    class="org.jboss.seam.ioc.spring.SeamManagedSessionFactoryBean">
  <property name="sessionName" value="hibernateSession"/>
</bean>
```

sessionName は **persistence:managed-hibernate-session** コンポーネントの名前です。 これでこの **SessionFactory** はいずれの Spring 提供ツールでも使用することができます。 この統合は **SeamManagedSessionFactory** で **getCurrentInstance()** を呼び出している場合であれば **SessionFactory.getCurrentInstance()** に対する呼び出しにも対応します。

26.8. SEAM コンポーネントとしての SPRING APPLICATION CONTEXT

Spring **ContextLoaderListener** を使ってアプリケーションの Spring **ApplicationContext** を起動することは可能ですが制約がいくつかあります。その制約とは、Spring **ApplicationContext** は **SeamListener** の後に起動させる必要があること、また Seam ユニットと統合テストで使用するために Spring **ApplicationContext** を起動することは複雑になる場合があることです。

これらの制約を克服するために Spring 統合には Spring **ApplicationContext** を起動できる Seam コンポーネントが含まれています。 このコンポーネントを使用するには、**<spring:context-loader/>** の定義を **components.xml** ファイルに配置します。 **config-locations** 属性で使用する Spring コンテキストファイルの場所を指定します。 複数の設定ファイルが必要な場合は、標準 **components.xml** の複数値のとおり、ネストした **<spring:config-locations/>** エレメントに配置することができます。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:spring="http://jboss.com/products/seam/spring"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation=
              "http://jboss.com/products/seam/components
               http://jboss.com/products/seam/components-2.2.xsd
               http://jboss.com/products/seam/spring
               http://jboss.com/products/seam/spring-2.2.xsd">

  <spring:context-loader config-locations=
                          "/WEB-INF/applicationContext.xml"/>

</components>
```

26.9. @ASYNCHRONOUS への SPRING TASKEXECUTOR の使用

Spring はコードを非同期に実行するために **TaskExecutor** と呼ばれる抽象を提供します。 Spring Seam 統合では **@Asynchronous** メソッド呼び出しを直ちに実行するために Spring の **TaskExecutor** を使用できます。 この機能を有効にするには **SpringTaskExecutorDispatchor** をインストールしてから次のように Spring Bean 定義の **taskExecutor** を与えます。

```
<spring:task-executor-dispatcher
        task-executor="#{springThreadPoolTaskExecutor}"/>
```

Spring の **TaskExecutor** は非同期イベントのスケジューリングには対応しないため、 代替となる Seam **Dispatcher** で処理することができます。

```
<!--
  Install a ThreadPoolDispatcher to handle scheduled asynchronous event
-->
<core:thread-pool-dispatcher name="threadPoolDispatcher"/>

<!-- Install the SpringDispatcher as default -->
<spring:task-executor-dispatcher
        task-executor="#{springThreadPoolTaskExecutor}"
        schedule-dispatcher="#{threadPoolDispatcher}"/>
```

第27章 HIBERNATE SEARCH

27.1. はじめに

Apache™ Lucene™ のようなフルテキスト検索エンジンにより、アプリケーションにフルテキストクエリと効率的なクエリを行うことが可能です。Apache Lucene を使用している Hibernate Search は数種類のアノテーションを追加したドメインモデルをインデックスし、データベースとインデックスの同期を処理し、フルテキストクエリに一致する通常の管理オブジェクトを返します。ただし、テキストのインデックスに対しドメインオブジェクトモデルを取り扱う検索を行う場合には次のような制限があります。インデックスの正確性を維持すること、インデックスの構造とドメインモデル間の一貫性、クエリの不整合を回避することなどです。しかし、検索スピードと効率面を考えれば、これらの制約を補ってあまりあるメリットがあります。

Hibernate Search はできるだけ自然に JPA および Hibernate と統合するよう設計されています。自然な流れとして JBoss Seam は Hibernate Search 統合を提供しています。

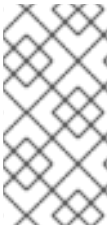
Hibernate Search プロジェクトに関する詳細は Hibernate Search documentation を参照してください。

27.2. 設定

Hibernate Search は **META-INF/persistence.xml** または **hibernate.cfg.xml** のいずれかのファイルで設定します。

Hibernate Search の設定はほとんどの設定パラメータで適切なデフォルト値が設定されています。以下に最低限の永続ユニットの設定を示します。

```
<persistence-unit name="sample">
  <jta-data-source>java:/DefaultDS</jta-data-source>
  <properties>
    [...]
    <!-- use a file system based index -->
    <property name="hibernate.search.default.directory_provider"
              value="org.hibernate.search.store.FSDirectoryProvider"/>
    <!-- directory where the indexes will be stored -->
    <property name="hibernate.search.default.indexBase"
              value="/Users/prod/apps/dvdstore/dvdindexes"/>
  </properties>
</persistence-unit>
```



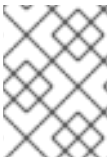
注記

Hibernate Search 3.1.x を使用する場合、より多くのイベントリスナーが必要になりますが、これらは Hibernate Annotations により自動的に登録されます。Hibernate EntityManager と Hibernate Annotations を使用せずにイベントリスナーを設定する方法は、『Hibernate Search Reference Guide』 を参照してください。

設定ファイルと共に次の **JAR** もデプロイする必要があります。

- **hibernate-search.jar**
- **hibernate-commons-annotations.jar**

- **lucene-core.jar**



注記

これらを **EAR** 内にデプロイする場合、 **application.xml** の更新を忘れずに行ってください。

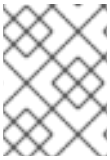
27.3. 使い方

Hibernate Search はアノテーションを使ってエンティティを Lucene のインデックスにマップします。 詳細については リファレンスマニュアル を参照してください。

Hibernate Search は API および JPA や Hibernate のセマンティックと完全に統合されています。 HQL ベースまたは検索条件ベースの問い合わせからの切り替えにはほとんどコードを必要としません。 アプリケーションは主に Hibernate の **Session** のサブクラスとなる **FullTextSession** API と連携します。

Hibernate Search が存在する場合は、JBoss Seam は **FullTextSession** をインジェクトします。

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{

  @In FullTextSession session;

  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    org.hibernate.Query query session.createFullTextQuery(luceneQuery,

Product.class);
    searchResults = query
                    .setMaxResults(pageSize + 1)
                    .setFirstResult(pageSize * currentPage)
                    .list();
  }
  [...]
}
```



注記

ここでは、 **FullTextSession** が **org.hibernate.Session** を拡張しているため通常の Hibernate Session として使用することができます。

JPA を使用した場合、より円滑な統合が提案されます。

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{
  @In FullTextEntityManager em;
  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
```

```
    javax.persistence.Query query = em.createFullTextQuery(luceneQuery,

Product.class);
    searchResults = query
                    .setMaxResults(pageSize + 1)
                    .setFirstResult(pageSize * currentPage)
                    .getResultList();
  }
  [...]
}
```

FullTextEntityManager は Hibernate Search が存在するところにインジェクトされます。 **FullTextEntityManager** は検索固有のメソッドで **EntityManager** を拡張します。 同様にして **FullTextSession** は **Session** を拡張します。

EJB 3.0 Session またはメッセージ駆動型 Beanのインジェクションが使用される場合 (つまりインジェクションが **@PersistenceContext** アノテーションを使用) は、宣言ステートメント内で **FullTextEntityManager** インターフェースを使うことで **EntityManager** インタフェースの置換はできません。 ただし、 インジェクトされる実装は **FullTextEntityManager** 実装になり、 ダウンキャストが可能です。

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{

  @PersistenceContext EntityManager em;

  public void search(String searchString) {
    org.apache.lucene.search.Query luceneQuery = getLuceneQuery();
    FullTextEntityManager ftEm = (FullTextEntityManager) em;
    javax.persistence.Query query =
      ftEm.createFullTextQuery(luceneQuery, Product.class);
    searchResults = query
                    .setMaxResults(pageSize + 1)
                    .setFirstResult(pageSize * currentPage)
                    .getResultList();
  }
  [...]
}
```

注記

Seam の外側で Hibernate Search を使用するのに慣れている方は、 Hibernate Search が Seam と統合されるときは **Search.createFullTextSession** を使用する必要がないことを覚えておいてください。

Hibernate Search の作業サンプルについては JBoss Seam ディストリビューションの Blog サンプルか DVDStore をご確認ください。

第28章 SEAM の設定と SEAM アプリケーションのパッケージング

設定は複雑でつまらない場合がありますが、 大部分はゼロから作る必要はありません。 Seam を ご使用の JavaServer Faces (JSF) 実装およびサーブレットコンテナと統合するために XML が数行必要な他は、 ほとんどの部分はアプリケーションの起動に seam-gen を使用するか、Seam で提供されるサンプルアプリケーションからコピーして貼り付けるだけです。

28.1. SEAM の基本設定

最初に、JSF と Seam を併用する場合に必要となる基本設定について見ていきます。

28.1.1. Seam と JSF、 サーブレットコンテナとの統合

まず Faces サーブレットを定義します。

```
<servlet>
   <servlet-name>Faces Servlet</servlet-name>
   <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>Faces Servlet</servlet-name>
   <url-pattern>*.seam</url-pattern>
</servlet-mapping>
```

(適宜 URL パターンを調整できます。)

また、 Seam には **web.xml** ファイルに次のエントリも必要になります。

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

このリスナーは Seam のブートストラップおよびセッションとアプリケーションのコンテキストの破棄を行います。

JSF 実装の中には Seam の対話伝播と動作するサーバー側状態保存を実装していないものがあります。フォームサブミット中の対話伝播に問題が見られる場合はクライアント側状態保存に切り替えてみてください。そのためには **web.xml** に次を追加します。

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
```

JSF 仕様ではビュー状態の値の可変性が不明瞭です。 Seam は JSF ビュー状態を使ってその **PAGE** スコープに戻るためこれが問題となる可能性があります。 JSF-RI (JSF 参照実装) でサーバー側状態保存を使用し、特定のページビューに対してページスコープの Bean にその正確な値を維持させたい場合、コンテキストパラメータを次のように指定する必要があります。

```
<context-param>
  <param-name>com.sun.faces.serializeServerState</param-name>
```

```
    <param-value>true</param-value>
</context-param>
```

これを指定しないとページスコープのコンポーネントはページの最新値を含むことになり、「戻る」ボタンを使用した場合に「戻る」ページの値を含みません (詳細は 本仕様に関する問題 を参照してください)。この設定はデフォルトでは有効になっていません。 JSF ビューを各要求でシリアライズ化すると全体的なパフォーマンスが低下するためです。

28.1.2. Facelet の使用

JavaServer Pages (JSP) に推奨の Facelets を使用するには次の行を **faces-config.xml** に追加します。

```
<application>
  <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
</application>
```

次に以下の行を **web.xml** に追加します。

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>
```

28.1.3. Seam Resource Servlet

Seam Resource Servlet は Seam Remoting、 CAPTCHA (章「セキュリティ」を参照) や JSF の UI コントロールで使用されるリソースを提供します。Seam Resource Servlet の設定には **web.xml** に以下のエントリが必要です。

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.servlet.SeamResourceServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

28.1.4. Seam Servlet フィルタ

Seam は基本操作には Servlet フィルタを必要としません。 ただし、フィルタの使用に依存する機能がいくつかあります。 Seam では他の組み込み Seam コンポーネントを設定する場合と同じようにして Servlet フィルタを追加、設定することができます。 この設定方法を利用するにはまず **web.xml** にマスターフィルタをインストールする必要があります。

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>
```

```
<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Seam マスターフィルタは **web.xml** で指定される 1 番目のフィルタで **なければなりません**。これでマスターフィルタが最初に実行されます。

Seam フィルタにはいくつかの共通の属性があります。 これらに加えて以降で説明するパラメータを **components.xml** で設定することができます。

- **url-pattern** – フィルタされる要求を指定するのに使用します。 デフォルトは全要求です。 **url-pattern** はワイルドカードサフィックスを許可するパターンです。
- **regex-url-pattern** – フィルタされる要求を指定するのに使用します。 デフォルトは全要求です。 **regex-url-pattern** は要求パスに対する実際の正規表現の一致です。
- **disabled** – 組み込みのフィルタの無効化に使用します。

これらのパターンは要求の URI パスに対して適合される点 (**HttpServletRequest.getURIPath()** を参照)、および Servlet コンテキスト名は適合が行われる前に削除される点に注意してください。

マスターフィルタを追加することにより、以下の組み込みフィルタが有効になります。

28.1.4.1. 例外処理

このフィルタは大部分のアプリケーションで必要とされ、 **pages.xml** に例外マッピングの機能を提供します。 また、キャッチされなかった例外が発生した場合にコミットされていないトランザクションのロールバックも行います (これは Web コンテナにより自動的に行われるはずですが、正しくこの動作を行わないアプリケーションサーバーもあります)。

デフォルトにより例外処理フィルタはすべての要求を処理しますが、以下のように **components.xml** に **<web:exception-filter>** エントリを追加してこれを変更することもできます。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:web="http://jboss.com/products/seam/web">
  <web:exception-filter url-pattern="*.seam"/>
</components>
```

28.1.4.2. リダイレクトによる対話の伝播

このフィルタにより Seam はブラウザリダイレクト全体に対話コンテキストを伝播することが可能です。あらゆるブラウザリダイレクトをインターセプトし、Seam の対話識別子を指定する要求パラメータを追加します。

リダイレクトフィルタもデフォルトですべての要求を処理しますが、 **components.xml** の記述を以下のように調節することも可能です。

```
<web:redirect-filter url-pattern="*.seam"/>
```

28.1.4.3. URL の書き換え

このフィルタにより Seam は **pages.xml** の設定に応じてビューの URL 書き換えを適用できます。 このフィルタはデフォルトではアクティブではありませんが、 **components.xml** に以下の設定を追加するとアクティブにできます。

```
<web:rewrite-filter view-mapping="*.seam"/>
```

view-mapping パラメータは **web.xml** ファイルにある Faces Servlet 用に定義された Servlet マッピングに一致しなければなりません。 省略すると書き換えフィルタはパターンが ***.seam** であるとみなします。

28.1.4.4. マルチパートフォームの送信

この機能は Seam の **ファイルアップロード** JSF コントロールを使用するときに必要です。 マルチパートフォームの要求を検出すると、 multipart/form-data 仕様 (RFC-2388) に従い処理を行います。設定を上書きするためには **components.xml** に以下を追加します。

```
<web:multipart-filter create-temp-files="true"
     max-request-size="1000000" url-pattern="*.seam"/>
```

- **create-temp-files** – **true** に設定するとアップロードされたファイルはメモリで保持されるのではなく一時ファイルに書き込まれます。 大容量ファイルのアップロードが予期されるときには考慮すべき重要な点となる場合があります。デフォルト設定は **false** です。
- **max-request-size** – ファイルのアップロード要求のサイズがこの値を越えるとその要求は中断されます。 デフォルト設定は **0** (サイズ制限なし) です (ファイルアップロードのサイズは要求の **Content-Length** ヘッダーを読み込んで決定されます)。

28.1.4.5. 文字エンコーディング

送信されたフォームデータの文字エンコーディングを設定するフィルタです。 デフォルトではこのフィルタはインストールされていないため、 有効にするには **components.xml** に以下のエントリが必要です。

```
<web:character-encoding-filter encoding="UTF-16"
     override-client="true" url-pattern="*.seam"/>
```

- **encoding** – 使用するエンコーディングタイプです。
- **override-client** – **true** に設定すると、 要求エンコーディングはその要求がすでにエンコーディングを指定しているか否かにかかわらず **encoding** で指定されているものに設定されます。 **false** に設定すると、 クライアントが要求エンコーディングをまだ指定していない場合にのみ設定されます。 デフォルト設定は **false** です。

28.1.4.6. RichFaces

RichFaces をプロジェクトに使用すると、Seam は RichFaces AJAX フィルタをその他すべての組み込みフィルタより先に自動的にインストールします。 このため、 **web.xml** に手作業で RichFaces Ajax を追加する必要はありません。

RichFaces Ajax フィルタは RichFaces **JAR** 群がプロジェクトにある場合にのみインストールされます。

デフォルト設定を上書きするには次のエントリを **components.xml** に追加します。 オプションは RichFaces Developer Guide に記載されているものと同じです。

```
<web:ajax4jsf-filter force-parser="true" enable-cache="true"
     log4j-init-file="custom-log4j.xml" url-pattern="*.seam"/>
```

- **force-parser** – JSF の全ページが Richfaces の XML 構文チェッカーにより強制的に検証されるようにします。 **false** に設定すると、AJAX の応答のみが検証され適格な XML に変換されます。 **force-parser** を **false** に設定するとパフォーマンスは向上しますが AJAX 更新で視覚アーティファクトが生じることがあります。
- **enable-cache** – フレームワーク生成のリソースのキャッシュ化を有効にします (javascript、CSS、 イメージなど)。 カスタムの javascript や CSS を開発している場合は **true** に設定するとブラウザにリソースをキャッシュさせないようにします。
- **log4j-init-file** – アプリケーションごとのログ記録の設定に使用されます。**log4j.xml** 設定ファイルにウェブアプリケーションコンテキストと相対的なパスを与えてください。

28.1.4.7. アイデンティティロギング

このフィルタは認証されたユーザー名を **log4j** マップ診断コンテキストに追加するため、 パターンに **%X{username}** を追加するとフォーマット化されたログ出力にそれを含めることができます。

デフォルトではロギングフィルタが全要求を処理します。 以下の例で示すように **<web:logging-filter>** のエントリを **components.xml** に追加するとこの動作を調整できます。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:web="http://jboss.com/products/seam/web">
  <web:logging-filter url-pattern="*.seam"/>
</components>
```

28.1.4.8. カスタムなサーブレットのコンテキスト管理

JSF Servlet 以外の Servlet に直接送信される要求は JSF のライフサイクルでは処理されません。 そこで、Seam は Seam コンポーネントにアクセスする必要のあるその他の Servlet に適用できる ServeIt フィルタを提供します。

このフィルタにより、カスタムな Servlet による Seam コンテキストとの通信を可能にします。各要求の最初に Seam コンテキストを設定し、要求の終了時にこれを破棄します。このフィルタは JSFの **FacesServlet** には **絶対に** 適用しないでください。 Seam は JSF 要求のコンテキスト管理にはフェーズリスナーを使用します。

デフォルトではこのフィルタはインストールされていないため、**components.xml** で有効にする必要があります。

```
<web:context-filter url-pattern="/media/*"/>
```

コンテキストフィルタは **conversationId** 要求パラメータで対話コンテキストの対話 ID が定義されることを期待します。必ず、要求に対話 IDを含めるようにしてください。

また、新たな対話 ID はクライアントに確実に伝播する必要があります。Seam は組み込みコンポーネント **conversation** のプロパティとして対話 ID を公開します。

28.1.4.9. カスタムフィルタの追加

Seam はフィルタをインストールすることができ、チェーン内にフィルタを配置する場所を指定できます (フィルタを **web.xml** で指定すると Servlet 仕様は明確な順序を提供しません)。**@Filter** アノテーションを Seam コンポーネントに追加します (Seam コンポーネントは **javax.servlet.Filter** を実装しなければなりません)。

```
@Startup
@Scope(APPLICATION)
@Name("org.jboss.seam.web.multipartFilter")
@BypassInterceptors
@Filter(within="org.jboss.seam.web.ajax4jsfFilter")
public class MultipartFilter extends AbstractFilter {...}
```

@Startup アノテーションを追加すると Seam 起動時にコンポーネントが使用可能となります。バイジェクションはここでは使用できません (**@BypassInterceptors**)。フィルタは RichFaces フィルタよりチェーンの下方にします (**@Filter(within="org.jboss.seam.web.ajax4jsfFilter")**)。

28.1.5. EJB コンテナと Seam の統合

Seam アプリケーション内の EJB コンポーネントは Seam と EJB コンテナの両方で管理されます。Seam は EJB コンポーネントの参照を解決し、ステートフルセッション Bean のコンポーネントのライフタイムを管理、インターセプタで各メソッドコールに参加します。Seam を EJB コンテナと統合するには最初にインターセプタチェーンを設定する必要があります。

SeamInterceptor を Seam EJB コンポーネントに適用します。このインターセプタはバイジェクション、対話区分、ビジネスプロセスのシグナルなどの操作を処理するサーバー側の組み込みインターセプター式に委譲します。アプリケーション全体に渡りこれを最も容易に行うためには、次のインターセプタ設定を **ejb-jar.xml** に追加します。

```
<interceptors>
  <interceptor>
    <interceptor-class>
      org.jboss.seam.ejb.SeamInterceptor
    </interceptor-class>
  </interceptor>
</interceptors>
<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>*</ejb-name>
    <interceptor-class>
      org.jboss.seam.ejb.SeamInterceptor
    </interceptor-class>
  </interceptor-binding>
</assembly-descriptor>
```

セッション Bean の JNDI 内の場所を Seam に指示する必要があります。各セッション Bean の Seam コンポーネントで **@JndiName** アノテーションを指定します。より適切な方法は、EJB 名から JNDI 名を判断できるようパターンを指定することです。ただし、EJB3 仕様ではグローバル JNDI をマッピングする標準的な方法は定義されていないため、このマッピングはベンダー固有であり、命名規則により異なる可能性もあります。このオプションは **components.xml** で指定します。

JBoss AS の場合、正しいパターンは次のとおりです。

```
<core:init jndi-name="earName/#{ejbName}/local" />
```

上記の **earName** は Bean がデプロイされる EAR 名です。 Seam は **#{ejbName}** をEJB 名に置き換えるため、 最後の部分はインターフェースのタイプを表します (ローカルまたはリモート)。

EAR コンテキストの外側では (JBoss Embeddable EJB3 コンテナを使用する場合など)、 **EAR** がないため最初の部分は省略されて次のようなパターンになります。

```
<core:init jndi-name="#{ejbName}/local" />
```

複雑な過程に見えますが実際には数点の手順だけです。

まず、 EJB コンポーネントがどのように JNDI に転送されるのか見ていきます。 XML の使用を避けるために JBoss AS は前述したパターン (**EAR** name/EJB name/interface タイプ) を使って自動的に EJB コンポーネントにグローバル JNDI 名を割り当てます。 次のうち値が空ではない最初の値が EJB 名となります。

- **ejb-jar.xml** 内の **<ejb-name>** エレメント
- **@Stateless** か **@Stateful** アノテーションの **name** 属性、 または
- Bean クラスの簡易名

例えば、 次の EJB Bean とインターフェースが定義されているとします。

```
package com.example.myapp;
import javax.ejb.Local;

@Local
public class Authenticator {
  boolean authenticate();
}


package com.example.myapp;
import javax.ejb.Stateless;

@Stateless
@Name("authenticator")
public class AuthenticatorBean implements Authenticator {
  public boolean authenticate() { ... }
}
```

EJB Bean クラスを **myapp** という名前で **EAR** にデプロイすると仮定すると、 JBoss AS で割り当てられるグローバル JNDI 名は **myapp/AuthenticatorBean/local** になります。 この EJB コンポーネントを **authenticator** という名前で Seam コンポーネントとして参照できるため、 Seam はその JNDI パターン (または **@JndiName** アノテーション) を使って JNDI 内で検索を行います。

他のアプリケーションサーバーの場合は EJB に EJB 参照を宣言する必要があります。 これにより JNDI 名が割り当てられます。 これには若干の XML が必要になります。 つまり、 Seam JNDI パターンを使用できるよう独自の JNDI 命名規則を確立する必要があるということです。 JBoss 規則に従うと便利な場合があります。

Seam を JBoss アプリケーションサーバー以外のサーバーと併用させる場合、 EJB 参照を 2 箇所で定義する必要があります。 Seam EJB コンポーネントを JSF (JSF ビュー内または JSF アクションリス

ナーとして) や Seam JavaBean コンポーネントで検索する場合は、EJB 参照を **web.xml** で宣言する必要があります。 次はこの例で必要となるEJB 参照です。

```
<ejb-local-ref>
  <ejb-ref-name>myapp/AuthenticatorBean/local</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local>org.example.vehicles.action.Authenticator</local>
</ejb-local-ref>
```

この参照は Seam アプリケーション内のコンポーネントのほとんどの使用に対応します。 Seam コンポーネントを **@In** アノテーションを付けて別の Seam EJB コンポーネントにインジェクトできるようにしたい場合は、この EJB 参照を 2 番目の場所となる **ejb-jar.xml** に定義する必要があります。 こちらの方が若干複雑です。

Seam が Seam EJB コンポーネントを検索して **@In** で定義されるインジェクションポイントを満たすと、 コンポーネントは JNDI 内で参照される場合にのみ見つかります。 JBoss は自動的に EJB を JNDI に登録するため、 常に Web および EJB コンテナに対して使用可能です。 他のコンテナの場合は EJB を明示的に定義する必要があります。

EJB 仕様を順守するアプリケーションサーバーは EJB 参照が常に明示的に定義されている必要があります。 これらはグローバルには宣言できません。 各 JNDI リソースを EJB コンポーネントに対して個別に指定する必要があります。

RegisterAction という解決済みの名前を持つ EJB があるとすると、 次の Seam インジェクションが適用されます。

```
@In(create = true) Authenticator authenticator;
```

このインジェクションを動作させるには、次のように **ejb-jar.xml** でリンクを確立する必要もあります。

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>RegisterAction</ejb-name>
      <ejb-local-ref>
      <ejb-ref-name>myapp/AuthenticatorAction/local</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <local>com.example.myapp.Authenticator</local>
      </ejb-local-ref>
    </session>
  </enterprise-beans>

  ...

</ejb-jar>
```

コンポーネントは **web.xml** で参照されたのと同じようにここで参照されます。 ここで識別することにより EJB コンテキストで参照が可能となり、 **RegisterAction** Bean がその参照を使用できるようになります。(**@In** により) 任意の Seam EJB コンポーネントの各インジェクションに対して 1 つの参照を別の Seam EJB コンポーネントに追加する必要があります。 **jee5/booking** サンプルでこの設定例を見ることができます。

特定の EJB を別の EJB に **@EJB** アノテーションを付けてインジェクトすることができますが、 Seam

EJB コンポーネントのインスタンスではなく EJB 参照をインジェクトすることになります。 Seam のインターセプタはいずれの **メソッド呼び出し** でも EJB コンポーネントに対して呼び出され、**@EJB** を使用することで Seam のサーバー側のインターセプタチェーンのみを呼び出すため、 **@EJB** インジェクションでは動作しなくなる Seam 機能がいくつかあります (セキュリティや同時実行を行う Seam の状態管理と Seam のクライアント側インターセプタチェーンなどがこれに該当する機能です)。ステートフルセッション Bean が **@EJB** を使ってインジェクトされると、 必ずしもアクティブなセッションや対話にバインドするとは限らないため、 **@In** を使ってインジェクトを行うことをお薦めします。

すべての EJB コンポーネントに対して明示的に JNDI 名の指定を必要とするアプリケーションサーバーがあり (Glassfish など)、 時には複数回に渡り指定を必要とすることがあります。 また、 JBoss AS 命名規則に従っている場合でも Seam で使用される JNDI パターンの変更を必要とする場合があります。 たとえば、 Glassfish ではグローバル JNDI 名の先頭に自動的にプレフィックス **java:comp/env** が付くため、 JNDI パターンを次のように定義する必要があります。

```
<core:init jndi-name="java:comp/env/earName/#{ejbName}/local" />
```

トランザクション管理には、 コンテナのトランザクションを完全に認識し、 **Events** コンポーネントで登録されるトランザクションの成功イベントを正しく処理できるような特殊な組み込みコンポーネントを使用することをお勧めします。 コンテナ管理トランザクションがいつ終了するのかを Seam に伝えるには次の行を **components.xml** ファイルに追加します。

```
<transaction:ejb-transaction/>
```

28.1.6. 注意点

統合における最後の要件として、 Seamコンポーネントがデプロイされるアーカイブにはすべて **seam.properties**、 **META-INF/seam.properties** または **META-INF/components.xml** ファイルを配置しておく必要があります。 Web アーカイブ (WAR) ファイルの場合は、コンポーネントがデプロイされる **WEB-INF/classes** ディレクトリの内側に **seam.properties** ファイルを配置します。

Seam は起動時に Seam コンポーネントの **seam.properties** ファイルを持つアーカイブをすべてスキャンします。 **seam.properties** ファイルは空でも構いませんが、 Seam がコンポーネントを認識できるようファイルを含ませなければなりません。 これは JVM (Java Virtual Machine) が持つ制約に対処する方法です。 **seam.properties** ファイルを持たせない場合は **components.xml** にすべてのコンポーネントを明示的に記載しなければならなくなります。

28.2. 代替の JPA プロバイダの使用

Seam にはデフォルトの JPA プロバイダとして Hibernate がパッケージ化され設定されています。 別の JPA プロバイダを使用する場合は Seam でそのプロバイダを設定する必要があります。

注記

これは対応策です。Seam の今後のバージョンでは、 カスタムな永続プロバイダの実装を追加しない限り、 代替の JPA プロバイダを使用するために設定変更を行う必要はなくなる予定です。

Seam に JPA プロバイダを認識させる方法は 2 種類あります。 1 つ目の方法はアプリケーションの **components.xml** を更新し、 汎用 **PersistenceProvider** が Hibernate バージョンより優先されるようにします。このファイルに次を追加するだけです。

```
<component name="org.jboss.seam.persistence.persistenceProvider"
```

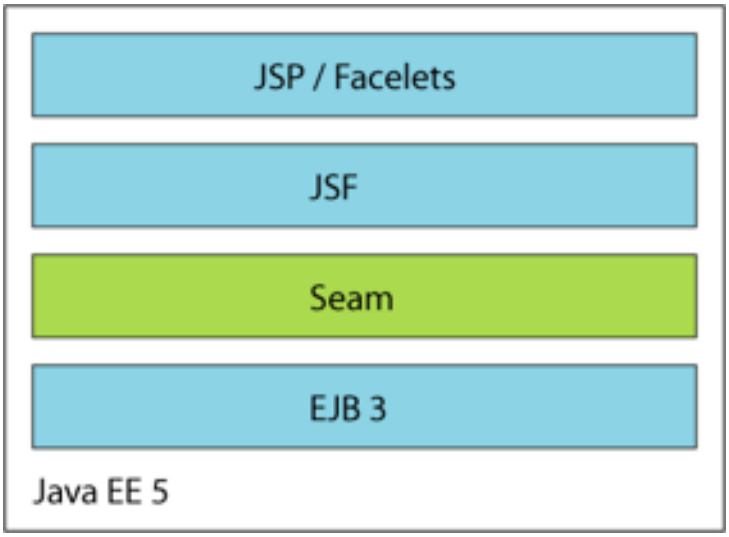
```
            class="org.jboss.seam.persistence.PersistenceProvider"
            scope="stateless">
</component>
```

JPA プロバイダの非標準の機能を利用したい場合は **PersistenceProvider** の独自の実装を記述する必要があります (**HibernatePersistenceProvider** を起点として利用できます)。次のように Seam にこの **PersistenceProvider** を使うよう指示します。

```
<component name="org.jboss.seam.persistence.persistenceProvider"
            class="org.your.package.YourPersistenceProvider">
</component>
```

あとは正しいプロバイダクラスおよび使用するプロバイダが必要とするプロパティで **persistence.xml** を更新するだけです。 必要となる **JAR** ファイル群を使用するアプリケーションでパッケージ化するのを忘れないようにしてください。

28.3. JAVA EE 5 での SEAM の設定



Java EE 5 環境で実行している場合は Seam の使用を開始するために必要な設定はこれだけです。

28.3.1. パッケージング

EAR へのパッケージ化が終了するとアーカイブは次のような構成になります。

```
my-application.ear/
    jboss-seam.jar
    lib/
        jboss-el.jar
    META-INF/
        MANIFEST.MF
        application.xml
    my-application.war/
        META-INF/
            MANIFEST.MF
        WEB-INF/
            web.xml
            components.xml
            faces-config.xml
            lib/
```

```
                jsf-facelets.jar
                jboss-seam-ui.jar
        login.jsp
        register.jsp
        ...
    my-application.jar/
        META-INF/
            MANIFEST.MF
            persistence.xml
        seam.properties
        org/
            jboss/
                myapplication/
                    User.class
                    Login.class
                    LoginBean.class
                    Register.class
                    RegisterBean.class
          ...
```

jboss-seam.jar を EJB モジュールとして **META-INF/application.xml** で宣言します。 **jboss-el.jar** を **EAR** の **lib** ディレクトリに配置して **EAR** のクラスパスに追加します。

jBPM または Drools を使用するには、 必要となる **JAR** 郡を **EAR** の **lib** ディレクトリに含めます。

推奨されているように Facelets を使用する場合は **jsf-facelets.jar** を **WAR** の **WEB-INF/lib** ディレクトリに含めます。

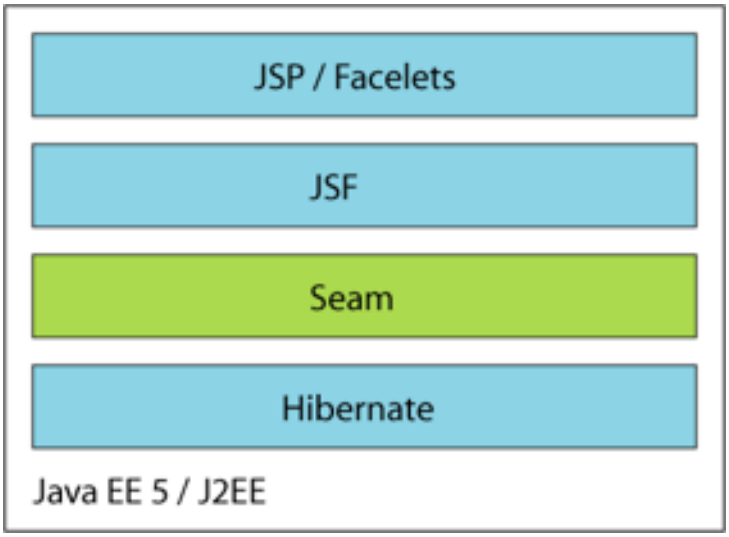
ほとんどのアプリケーションは Seam のタグライブラリを使用します。そのためには **jboss-seam-ui.jar** を WAR の **WEB-INF/lib** ディレクトリに含めます。 PDF や email のタグライブラリを使用する場合には、 **WEB-INF/lib** に **jboss-seam-pdf.jar** または **jboss-seam-mail.jar** を配置する必要もあります。

Seam デバッグページを使用する場合は、 **jboss-seam-debug.jar** を **WAR** の **WEB-INF/lib** ディレクトリに含めます。 Seam のデバッグページが正しく動作するのは Facelets を使用するアプリケーションのみです。

Seam にはサンプルのアプリケーションも数点同梱されています。 これらは EJB3 サポートのある Java EE コンテナならいずれでもデプロイが可能です。

28.4. J2EE での SEAM の設定

EJB3 永続の代わりに Hibernate 3 か JPA、 また セッション Bean の代わりに JavaBean を使用することができます。 Seam の宣言的な状態管理アーキテクチャの利点も活用できるため、 EJB3 への移行が容易になります。



Seam JavaBean コンポーネントはセッション Bean のような宣言的トランザクション境界設定は提供しません。 JavaBean で Hibernate を使用する場合はほとんどのアプリケーションが Seam 管理トランザクションを使用しますが、 JTA **`UserTransaction`** で手作業による管理、 または Seam の **`@Transactional`** アノテーションで宣言的に管理を行うこともできます。

Seam ディストリビューションには、予約サンプルアプリケーションの追加バージョンが含まれています。 ひとつは EJB3 の代わりに Hibernate3 と JavaBean を使用し、もう 1 つは JPA と JavaBean を使用します。 サンプルアプリケーションはいずれの J2EE アプリケーションサーバーにもデプロイ可能です。

28.4.1. Seam での Hibernate のブートストラップ

次の組み込みコンポーネントをインストールして、 Seam に **`hibernate.cfg.xml`** ファイルから Hibernate の **`SessionFactory`** をブートストラップさせます。

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>
```

Seam 管理の Hibernate **`Session`** をインジェクトにより使用可能にするには次のように**`managed session`** を設定します。

```
<persistence:managed-hibernate-session name="hibernateSession"
            session-factory="#{hibernateSessionFactory}"/>
```

28.4.2. Seam での JPA のブートストラップ

次の組み込みコンポーネントをインストールして、Seam に **`persistence.xml`** ファイルから JPA の **`EntityManagerFactory`** をブートストラップさせます。

```
<persistence:entity-manager-factory name="entityManagerFactory"/>
```

Seam 管理の JPA **`EntityManager`** をインジェクトにより使用可能にするには次のように管理永続コンテキストを設定します。

```
<persistence:managed-persistence-context name="entityManager"
            entity-manager-factory="#{entityManagerFactory}"/>
```

28.4.3. パッケージング

WAR としてパッケージするとアプリケーションは次のような構成になります。

```
my-application.war/
    META-INF/
        MANIFEST.MF
    WEB-INF/
        web.xml
        components.xml
        faces-config.xml
        lib/
            jboss-seam.jar
            jboss-seam-ui.jar
            jboss-el.jar
            jsf-facelets.jar
            hibernate3.jar
            hibernate-annotations.jar
            hibernate-validator.jar
            ...
            my-application.jar/
                META-INF/
                   MANIFEST.MF
                seam.properties
                hibernate.cfg.xml
                org/
                    jboss/
                        myapplication/
                            User.class
                            Login.class
                            Register.class
                            ...
    login.jsp
    register.jsp
    ...
```

TestNG など EE ではない環境に Hibernate をデプロイするには追加の設定が必要です。

28.5. JBOSS EMBEDDED のない JAVA SE での SEAM 設定

Seam を EE 環境の外側で使用するためには、 使用できる JTA がないので Seam にどのようにトランザクションを管理するのかを指示する必要があります。 JPA を使用している場合は Seam に JPA リソースローカルのトランザクション、 **EntityTransaction** などを使用するよう指示することができます。

```
<transaction:entity-transaction entity-manager="#{entityManager}"/>
```

Hibernate を使用している場合は、Seam に次のように Hibernate トランザクション API を使用するよう指示することができます。

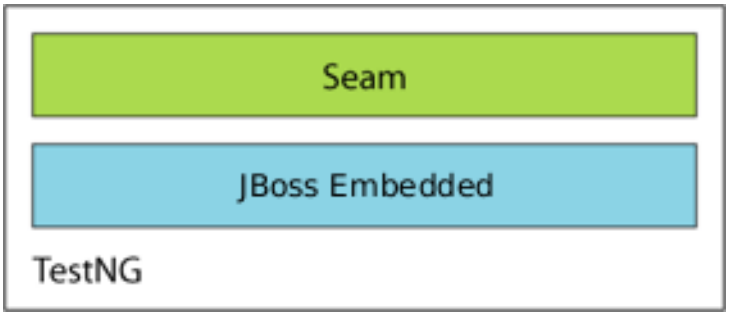
```
<transaction:hibernate-transaction session="#{session}"/>
```

また、 データソースも定義する必要があります。

28.6. JBOSS EMBEDDED を使用した JAVA SE での SEAM 設定

JBoss Embedded により Java EE 5 アプリケーションサーバーのコンテキストの外側で EJB 3 のコンポーネントを実行することができます。これは特にテストを行うときに便利です。

Seam 予約サンプルアプリケーションには TestNG 統合テストスィートが含まれ、 **SeamTest** を通じて Embedded JBoss で実行します。



28.6.1. パッケージング

Servlet エンジンでの **WAR** ベースのデプロイメントは次のような構成になります。

```
my-application.war/
    META-INF/
        MANIFEST.MF
    WEB-INF/
        web.xml
        components.xml
        faces-config.xml
        lib/
            jboss-seam.jar
            jboss-seam-ui.jar
            jboss-el.jar
            jsf-facelets.jar
            jsf-api.jar
            jsf-impl.jar
            ...
            my-application.jar/
                META-INF/
                    MANIFEST.MF
                    persistence.xml
                seam.properties
                org/
                    jboss/
                        myapplication/
                            User.class
                            Login.class
                            LoginBean.class
                            Register.class
                            RegisterBean.class
                            ...
    login.jsp
    register.jsp
    ...
```

28.7. SEAM での JBPM の設定

Seam の jBPM 統合はデフォルトではインストールされていません。 jBPM を有効にするには、 組み込みコンポーネントをインストールする必要があります。 また、 使用するプロセスとページフローの定義を **components.xml** に明示的に記載する必要があります。

```
<bpm:jbpm>
  <bpm:pageflow-definitions>
    <value>createDocument.jpdl.xml</value>
    <value>editDocument.jpdl.xml</value>
    <value>approveDocument.jpdl.xml</value>
  </bpm:pageflow-definitions>
  <bpm:process-definitions>
    <value>documentLifecycle.jpdl.xml</value>
  </bpm:process-definitions>
</bpm:jbpm>
```

ページフローしかない場合はこれ以上の設定は不要です。 ビジネスプロセスの定義がある場合は jBPM 設定および jBPM 用の Hibernate 設定も用意する必要があります。 Seam DVD Store demo には、Seam で機能するサンプルの **jbpm.cfg.xml** と **hibernate.cfg.xml** ファイルが含まれています。

```
<jbpm-configuration>
  <jbpm-context>
    <service name="persistence">
      <factory>
        <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
          <field name="isTransactionEnabled"><false/></field>
        </bean>
      </factory>
    </service>
    <service name="tx" factory="org.jbpm.tx.TxServiceFactory" />
    <service name="message"
     factory="org.jbpm.msg.db.DbMessageServiceFactory" />
    <service name="scheduler"
             factory="org.jbpm.scheduler.db.DbSchedulerServiceFactory" />
    <service name="logging"
             factory="org.jbpm.logging.db.DbLoggingServiceFactory" />
    <service name="authentication"
             factory="org.jbpm.security.authentication
                          .DefaultAuthenticationServiceFactory"/>
  </jbpm-context>
</jbpm-configuration>
```

jBPM トランザクションコントロールは無効である点に注意してください。 JTA のトランザクションは Seam または EJB3 のいずれかで制御してください。

28.7.1. パッケージング

jBPM 設定やプロセスおよびページフローの定義ファイルに対する明確なパッケージング形式はありません。他の標準パッケージング形式が開発されるかもしれませんが、Seam サンプルは EAR のルートにパッケージされており、次のような構成となります。

```
my-application.ear/
    jboss-seam.jar
    lib/
       jboss-el.jar
       jbpm-jpdl.jar
```

```
    META-INF/
        MANIFEST.MF
        application.xml
    my-application.war/
        META-INF/
            MANIFEST.MF
        WEB-INF/
            web.xml
            components.xml
            faces-config.xml
            lib/
                jsf-facelets.jar
                jboss-seam-ui.jar
        login.jsp
        register.jsp
        ...
    my-application.jar/
        META-INF/
            MANIFEST.MF
            persistence.xml
        seam.properties
        org/
            jboss/
                myapplication/
                    User.class
                    Login.class
                    LoginBean.class
                    Register.class
                    RegisterBean.class
                    ...
    jbpm.cfg.xml
    hibernate.cfg.xml
    createDocument.jpdl.xml
    editDocument.jpdl.xml
    approveDocument.jpdl.xml
    documentLifecycle.jpdl.xml
```

28.8. JBOSS ASで SFSB とセッションのタイムアウトの設定

ステートフルセッション Bean のタイムアウトは、HTTP セッションのタイムアウトより長く設定しなければなりません。 これをしないとユーザーの HTTP セッションが終了する前にステートフルセッション Bean がタイムアウトする可能性があります。 JBoss AS のデフォルトのセッション Bean タイムアウトは 30 分で、 これは **server/default/conf/standardjboss.xml** で設定されます。 これを変更するには **default** を希望の設定に置き換えます。

LRUStatefulContextCachePolicy キャッシュ設定で、**max-bean-life** の値を修正してデフォルトのステートフルセッション Bean のタイムアウトを変更します。

```
<container-cache-conf>
  <cache-policy>
    org.jboss.ejb.plugins.LRUStatefulContextCachePolicy
  </cache-policy>
  <cache-policy-conf>
    <min-capacity>50</min-capacity>
    <max-capacity>1000000</max-capacity>
```

```
    <remover-period>1800</remover-period>

    <!-- SFSB timeout in seconds; 1800 seconds == 30 minutes -->
    <max-bean-life>1800</max-bean-life>

    <overager-period>300</overager-period>
    <max-bean-age>600</max-bean-age>
    <resizer-period>400</resizer-period>
    <max-cache-miss-period>60</max-cache-miss-period>
    <min-cache-miss-period>1</min-cache-miss-period>
    <cache-load-factor>0.75</cache-load-factor>
  </cache-policy-conf>
</container-cache-conf>
```

JBoss Enterprise Application Platform 5.1 では、デフォルトの HTTP セッションタイムアウトを **server/default/deployer/jboss-web.deployer/conf/web.xml** で変更できます。**web.xml** ファイルの次のエントリはすべての Web アプリケーションのデフォルトセッションタイムアウトを制御します。

```
<session-config>
  <!-- HTTP Session timeout, in minutes -->
  <session-timeout>30</session-timeout>
</session-config>
```

使用するアプリケーション用にこの値を上書きするには、 このエントリの修正バージョンをアプリケーションの **web.xml** に含めるだけで可能です。

28.9. PORTLET での SEAM の実行

警告

Technology Preview の機能は Red Hat サブスクリプションレベルアグリーメント (SLA) では完全に対応していません。また、機能的に完全ではない場合があるため実稼働での使用を目的としていません。ただし、こうした機能により今後の新製品開発に早くアクセスすることができるため、開発段階でお客様が機能性をテストしたり、フィードバックをお寄せいただくことができます。Red Hat は 今後強化された Technology Preview の機能を一般的に利用できるよう検討しており、商業的に合理的な範囲でお客様がこうした機能を使用しているときに直面するすべての問題の解決に向けて努力します。

JBoss Portlet Bridge を使用するとポートレット内での Seam アプリケーションの実行が可能になります。 このブリッジはポートレットで JSF をサポートし、 Seam および RichFaces 用の拡張を含んでいます。 詳細は http://labs.jboss.com/ portletbridge を参照してください。

28.10. カスタムリソースのデプロイ

起動時に、Seam はリソースに対し **/seam.properties**、 **/META-INF/components.xml**、 または **/META-INF/seam.properties** を含んでいるすべての **JAR** をスキャンします。 たとえば、**@Name** アノテーションが付与されたクラスはすべて起動時に Seam コンポーネントとして登録されます。

Seam を使ってカスタムのリソースを処理することもできます。つまり、Seam は特定のアノテーションを処理できるということです。まず、次のように **/META-INF/seam-deployment.properties** ファイルで処理するアノテーションタイプの一覧を与えます。

```
# A colon-separated list of annotation types to handle
org.jboss.seam.deployment.annotationTypes=com.acme.Foo:com.acme.Bar
```

次に、アプリケーション起動時に **@Foo** アノテーションが付くすべてのクラスを収集します。

```
@Name("fooStartup")
@Scope(APPLICATION)
@Startup
public class FooStartup {

   @In("#{deploymentStrategy.annotatedClasses['com.acme.Foo']}")
   private Set<Class<Object>> fooClasses;

   @In("#{hotDeploymentStrategy.annotatedClasses['com.acme.Foo']}")
   private Set<Class<Object>> hotFooClasses;

   @Create
   public void create() {
      for (Class clazz: fooClasses) {
         handleClass(clazz);
      }
      for (Class clazz: hotFooClasses) {
         handleClass(clazz);
      }
   }

   public void handleClass(Class clazz) {
       // ...
   }

}
```

また、あらゆるリソースを処理するよう Seam を設定することもできます。たとえば、**.foo.xml** 拡張子が付くファイルを処理したい場合はカスタムのデプロイメントハンドラを記述することができます。

```
public class FooDeploymentHandler implements DeploymentHandler {
  private static DeploymentMetadata FOO_METADATA = new
DeploymentMetadata() {

    public String getFileNameSuffix() {
      return ".foo.xml";
    }
  };

  public String getName() {
    return "fooDeploymentHandler";
  }

  public DeploymentMetadata getMetadata() {
```

```
    return FOO_METADATA;
  }
}
```

これによりサフィックス **.foo.xml** が付くすべてのファイルの一覧が提供されます。

次に、 **/META-INF/seam-deployment.properties** でデプロイメントハンドラを Seam に登録します。

```
# For standard deployment
# org.jboss.seam.deployment.deploymentHandlers=
#    com.acme.FooDeploymentHandler

# For hot deployment
# org.jboss.seam.deployment.hotDeploymentHandlers=
#    com.acme.FooDeploymentHandler
```

コンマで区切った一覧を使うと複数のデプロイメントハンドラを登録することができます。

Seam はデプロイメントハンドラを内部的に使ってコンポーネントと名前空間をインストールするため、**handle()** は Seam のブートストラップで使用できるようかなり早くに呼び出されます。 アプリケーションによりスコープされたコンポーネントの起動中にデプロイメントハンドラに簡単にアクセスすることができます。

```
@Name("fooStartup")
@Scope(APPLICATION)
@Startup
public class FooStartup {
  @In("#{deploymentStrategy.deploymentHandlers['fooDeploymentHandler']}")
  private FooDeploymentHandler myDeploymentHandler;
  @In("#
{hotDeploymentStrategy.deploymentHandlers['fooDeploymentHandler']}")
  private FooDeploymentHandler myHotDeploymentHandler;
  @Create public void create() {
    for (FileDescriptor fd: myDeploymentHandler.getResources()) {
      handleFooXml(fd);
    }
    for (FileDescriptor f: myHotDeploymentHandler.getResources()) {
      handleFooXml(fd);
    }
  }

  public void handleFooXml(FileDescriptor fd) {
      // ...
  }
}
```

第29章 SEAM アノテーション

Seam ではアノテーションを使用して宣言的なプログラミングを実現することができます。アノテーションのほとんどは EJB 3.0 仕様で定義されています。 また、 データ検証用のアノテーションは Hibernate Validator パッケージで定義されています。 ただし、 Seam は Seam 独自のアノテーションセットを持っており、 これは本章で説明します。

これらのアノテーションはすべてパッケージ **org.jboss.seam.annotations** で定義されます。

29.1. コンポーネント定義のためのアノテーション

このアノテーショングループは Seam コンポーネントの定義に使用されます。 これらのアノテーションはコンポーネントクラスで見られます。

@Name

```
@Name("componentName")
```

クラスに対して Seam コンポーネント名を定義します。 このアノテーションは Seam の全コンポーネントに必要です。

@Scope

```
@Scope(ScopeType.CONVERSATION)
```

コンポーネントのデフォルトコンテキストを定義します。**EVENT**、 **PAGE**、 **CONVERSATION**、 **SESSION**、 **BUSINESS_PROCESS**、 **APPLICATION**、 **STATELESS** などの **ScopeType** を列挙することで可能な値を定義します。

スコープが明示的に指定されていない場合、 デフォルトはコンポーネントタイプにより異なります。 ステートレスセッション Bean の場合、 デフォルトは **STATELESS** です。 エンティティ Bean およびステートフルセッション Bean なら、 デフォルトは **CONVERSATION** です。JavaBeans のデフォルトは **EVENT** です。

@Role

```
@Role(name="roleName", scope=ScopeType.SESSION)
```

Seam コンポーネントを複数のコンテキスト変数に結合できるようにします。 **@Name** と **@Scope** のアノテーションは **デフォルトロール** を定義します。 各 **@Role** アノテーションは追加のロールを定義します。

- **name** – コンテキスト変数名です。
- **scope** – コンテキスト変数のスコープです。 スコープが明示的に指定されない場合、 デフォルトは上記のとおりコンポーネントタイプにより異なります。

@Roles

```
@Roles({ @Role(name="user", scope=ScopeType.CONVERSATION),
@Role(name="currentUser", scope=ScopeType.SESSION) })
```

複数の追加ロールを指定することができます。

@BypassInterceptors

```
@BypassInterceptors
```

特定のコンポーネントまたはコンポーネントメソッドにおける Seam インターセプタをすべて無効にします。

@JndiName

```
@JndiName("my/jndi/name")
```

Seam が EJB コンポーネントの検索に使用する JNDI 名を指定します。 JNDI 名が明示的に指定されない場合、 Seam は **org.jboss.seam.core.init.jndiPattern** で指定される JNDI パターンを使用します。

@Conversational

```
@Conversational
```

対話スコープのコンポーネントが対話用であることを指定します。 つまり長期実行の対話が起こっていない限り、 そのコンポーネントのメソッドは呼び出されません。

@PerNestedConversation

```
@PerNestedConversation
```

対話スコープのコンポーネントのスコープを、 そのコンポーネントがインスタンス化された親対話だけに制限します。 そのコンポーネントのインスタンスは独自のインスタンス内で動作するネストされた子対話からは見えません。

警告

これは推奨されるアプリケーション機能ではありません。 要求サイクルの特定部分にしかコンポーネントは見えないということになります。

@Startup

```
@Scope(APPLICATION) @Startup(depends="org.jboss.seam.bpm.jbpm")
```

アプリケーションスコープのコンポーネントが初期化時に直ちに開始されることを指定します。 JNDI、 データソースなど重要なインフラストラクチャをブートストラップする組み込みコンポーネントに使用されます。

```
@Scope(SESSION) @Startup
```

セッションスコープのコンポーネントがセッション作成時に直ちに開始されることを指定します。

- **depends** – 指定されたコンポーネントがインストールされている場合は、 そのコンポーネントを先に開始しなければならないことを指定します。

@Install

```
@Install(false)
```

コンポーネントがデフォルトでインストールされないよう指定します (このアノテーションを指定しない場合はコンポーネントがインストールされます)。

```
@Install(dependencies="org.jboss.seam.bpm.jbpm")
```

依存関係として表示されているコンポーネント群もあわせてインストールされる場合にのみ、このコンポーネントをインストールすることを指定します。

```
@Install(genericDependencies=ManagedQueueSender.class)
```

特定のクラスで実装されるコンポーネントがインストールされる場合にのみ、 コンポーネントがインストールされることを指定します。必要な依存関係の名前が不定である場合に便利です。

```
@Install(classDependencies="org.hibernate.Session")
```

指定されたクラスがクラスパス内にある場合にのみ、コンポーネントをインストールするよう指定します。

```
@Install(precedence=BUILT_IN)
```

コンポーネントの優先度を指定します。 同じ名前のコンポーネントが複数存在する場合、 より高い優先度を持つコンポーネントがインストールされます。 定義される優先度の値は次の通りです (昇順)。

- **BUILT_IN** – すべての組み込み Seam コンポーネントの優先度
- **FRAMEWORK** – Seam を拡張するフレームワークのコンポーネントを使うための優先度
- **APPLICATION** – アプリケーションコンポーネントの優先度 (デフォルトの優先度)
- **DEPLOYMENT** – 特定のデプロイメントにおいてアプリケーションコンポーネントを上書きするコンポーネントを使うための優先度
- **MOCK** – テスト時に使うモックオブジェクトに対する優先度

@Synchronized

```
@Synchronized(timeout=1000)
```

コンポーネントが複数のクライアントによって同時にアクセスされること、 Seam が要求をシリアライズすることを指定します。 要求が特定のタイムアウト期間内にコンポーネントでロックを取得できないと例外が発生します。

@ReadOnly

```
@ReadOnly
```

JavaBean コンポーネントまたはコンポーネントメソッドが呼び出しの終わりで状態の複製を必要としないことを指定します。

@AutoCreate

```
@AutoCreate
```

コンポーネントが自動的に作成されるよう指定します。クライアントが **create=true** を指定していなくても作成されます。

29.2. バイジェクション用アノテーション

次の 2 つのアノテーションはバイジェクションを制御します。 これらの属性はコンポーネントインスタンス変数またはプロパティのアクセサメソッドで発生します。

@In

```
@In
```

コンポーネントの属性が各コンポーネント呼び出しの開始時にコンテキスト変数からインジェクトされることを指定します。 コンテキスト変数が null の場合、 例外が送出されます。

```
@In(required=false)
```

コンポーネントの属性が各コンポーネント呼び出しの開始時にコンテキスト変数からインジェクトされることを指定します。 コンテキスト変数は null でも構いません。

```
@In(create=true)
```

コンポーネントの属性がコンポーネント呼び出しの開始時にコンテキスト変数からインジェクトされることを指定します。コンテキスト変数が null の場合、コンポーネントのインスタンスが Seam によって作成されます。

```
@In(value="contextVariableName")
```

アノテーションを付けられたインスタンス変数名を使用せず、 コンテキスト変数名を明示的に指定します。

```
@In(value="#{customer.addresses['shipping']}")
```

各コンポーネント呼び出しの開始時に JSF EL 式を評価することで、コンポーネントの属性がインジェクトされることを指定します。

- **value** – コンテキスト変数名を指定します。デフォルトはコンポーネントの属性名です。あるいは **#{...}** で囲まれた JSF EL 式を指定します。
- **create** – コンテキスト変数がすべてのコンテキストで未定義 (null) の場合、 Seam がコンテキスト変数と同じ名前のコンポーネントをインスタンス化するよう指定します。デフォルトは **false** です。

- **required** – コンテキスト変数がすべてのコンテキストで未定義の場合、 Seam が例外を送出するよう指定します。

@Out

```
@Out
```

Seam コンポーネントであるコンポーネントの属性が呼び出しの終わりでそのコンテキスト変数にアウトジェクトされることを指定します。 属性が null の場合、 例外が送出されます。

```
@Out(required=false)
```

Seam コンポーネントであるコンポーネント属性が呼び出しの終わりでそのコンテキスト変数にアウトジェクトされることを指定します。 属性は null でも構いません。

```
@Out(scope=ScopeType.SESSION)
```

Seam コンポーネントタイプ **ではない** コンポーネントの属性が呼び出しの終わりで特定スコープにアウトジェクトされることを指定します。

別の方法として、明示的にスコープが指定されていない場合、**@Out** 属性を持つコンポーネントのスコープ (またはコンポーネントがステートレスであれば **EVENT** スコープ) が使用されます。

```
@Out(value="contextVariableName")
```

アノテーションを付けられたインスタンス変数名を使用せず、 コンテキスト変数名を明示的に指定します。

- **value** – コンテキスト変数名を指定します。デフォルトはコンポーネントの属性名です。
- **required** – アウトジェクトするときにコンポーネントの属性が null の場合、 Seam が例外を送出するよう指定します。

これらのアノテーションは通常次のように共に利用します。

```
@In(create=true)
@Out private User currentUser;
```

Seam コンポーネントがインジェクトされる他のクラスのインスタンスのライフサイクルを管理する場合、 次のアノテーションは **マネージャコンポーネント** パターンをサポートします。 コンポーネントの getter メソッドに付与されます。

@Unwrap

```
@Unwrap
```

このアノテーションが付いている getter メソッドにより返されるオブジェクトがコンポーネントの代わりにインジェクトされることを指定します。

次のアノテーションは **ファクトリコンポーネント** パターンをサポートします。 この場合 Seam コンポーネントがコンテキスト変数の値の初期化を行います。 特に非 Faces 要求に対する応答のレンダリングに必要なあらゆる状態の初期化に便利です。 コンポーネントメソッドで指定されます。

@Factory

```
@Factory("processInstance")
public void createProcessInstance() { ... }
```

コンテキスト変数に値がない場合に、このコンポーネントのメソッドが指定コンテキスト変数の値の初期化に使用されることを指定します。このスタイルは **void** を返すメソッドと併用します。

```
@Factory("processInstance", scope=CONVERSATION)
public ProcessInstance createProcessInstance() { ... }
```

コンテキスト変数に値がない場合、指定コンテキスト変数の値の初期化にはこのメソッドで返される値を使用することを指定します。このスタイルは値を返すメソッドと併用します。明示的にスコープが指定されていない場合、**@Factory** メソッドを持つコンポーネントのスコープが使用されます (そのコンポーネントがステートレスではない場合 **EVENT** コンテキストが使用されます)。

- **value** – コンテキスト変数の名前を指定します。メソッドが getter メソッドの場合、JavaBeans のプロパティ名がデフォルトになります。
- **scope** – Seam が戻り値をバインドするスコープを指定します。値を返すファクトリメソッドの場合にのみ意味があります。
- **autoCreate** – **@In** が **create=true** を指定していない場合でも、変数が要求されたときは常にこのファクトリメソッドが自動的に呼び出されるよう指定します。

次は **Log** をインジェクトできるようにするアノテーションです。

@Logger

```
@Logger("categoryName")
```

コンポーネントフィールドに **org.jboss.seam.log.Log** のインスタンスをインジェクトするよう指定します。エンティティ Bean の場合、このフィールドは **static** として宣言されなければなりません。

- **value** – ログカテゴリの名前を指定します。デフォルトはコンポーネントのクラス名です。

最後のアノテーションでは要求パラメータ値をインジェクトできます。

@RequestParameter

```
@RequestParameter("parameterName")
```

コンポーネントの属性に要求パラメータ値をインジェクトするよう指定します。基本的なタイプの対話は自動的に行われます。

- **value** – 要求パラメータの名前を指定します。デフォルトはコンポーネントの属性名です。

29.3. コンポーネントのライフサイクルメソッド用アノテーション

これらのアノテーションにより、 コンポーネントがそのコンポーネント自体のライフサイクルイベントに対して反応することができ、コンポーネントのメソッドで発生します。各コンポーネントクラスごとにアノテーションはそれぞれ1つのみ定義できます。

@Create

```
@Create
```

コンポーネントのインスタンスが Seam によってインスタンス化されたときにメソッドが呼び出されるよう指定します。create メソッドは JavaBeans およびステートフルセッション Bean に対してしかサポートされません。

@Destroy

```
@Destroy
```

コンテキストが終了し、 そのコンテキスト変数が破棄されるときにメソッドが呼び出されることを指定します。destroy メソッドは JavaBeans およびステートフルセッション Bean に対してしかサポートされません。

Destroy メソッドはクリーンアップにのみ使用するようにしてください。 Seam は destroy メソッドから伝播する例外はすべてキャッチしてログを出力し、捨ててしまいます。

@Observer

```
@Observer("somethingChanged")
```

指定されたタイプのコンポーネント駆動イベントが発生すると、 このメソッドが呼び出されるよう指定します。

```
@Observer(value="somethingChanged",create=false)
```

指定したタイプのイベントが発生したときにそのメソッドを呼び出すこと、ただしインスタンスが存在しない場合はそのインスタンスは作成されないことを指定します。 インスタンスが存在せず、create が **false** に設定されている場合はイベントは監視されません。 create のデフォルト値は **true** です。

29.4. コンテキスト境界用アノテーション

これらのアノテーションは宣言的対話の境界を設定します。 これらは Seam コンポーネントのメソッド上、 通常はアクションリスナーメソッドに付与されます。

すべての Web 要求は対話コンテキストに関連付けられています。 ほとんどの対話は要求の完了と同時に終了します。 複数の要求にわたる対話が必要であれば、**@Begin** を付けたメソッドを呼び出すことで、その対話を **長期実行の対話** に「昇格」させなければなりません。

@Begin

```
@Begin
```

このメソッドが例外を送出することなく null 以外の結果を返す場合、長期対話の対話が開始することを指定します。

```
@Begin(join=true)
```

長期実行の対話がすでに開始されている場合、 対話コンテキストが伝播されることを指定します。

```
@Begin(nested=true)
```

長期実行の対話がすでに開始されている場合、 新たに **ネストされた** 対話コンテキストが開始することを指定します。 次の **@End** が出現したときにこのネストされた対話が終了し、 外側の対話が再開します。 外側の同じ対話内でネストされた複数の対話を同時に存在させることが可能です。

```
@Begin(pageflow="process definition name")
```

この対話のためのページフローを定義する jBPM プロセス定義の名前を指定します。

```
@Begin(flushMode=FlushModeType.MANUAL)
```

Seam 管理永続コンテキストのフラッシュモードを指定します。
flushMode=FlushModeType.MANUAL は **アトミックな対話** の使用に対応します。 この場合、 **flush ()** (通常、 対話終了時に呼び出される) への明示的な呼び出しが起きるまで、 すべての書き込み操作は対話コンテキスト内の待ち行列に入れられます。

- **join** – 長期実行の対話が既に始まっている場合にその動作を確定します。 **true** ならば、 コンテキストは伝播されます。 **false** の場合は例外が送出されます。 デフォルトは **false** です。 **nested=true** が指定されている場合はこの設定は無視されます。
- **nested** – 長期実行の対話が既に開始されている場合はネストした対話が開始されることを指定します。
- **flushMode** – この対話で作成される Seam 管理の Hibernate セッションまたは JPA 永続コンテキストのフラッシュモードを設定します。
- **pageflow** – **org.jboss.seam.bpm.jbpm.pageflowDefinitions** によってデプロイされる jBPM プロセス定義の名前です。

@End

```
@End
```

このメソッドが例外を送出することなく null 以外の結果を返す場合、長期実行の対話が終了することを指定します。

- **beforeRedirect** – デフォルトでは、 リダイレクトが発生するまでこの対話は実際には破棄されません。 **beforeRedirect=true** を設定することで、 現在の要求の最後に対話が破棄され、 リダイレクトは新しい一時的な対話コンテキストで処理されるよう指定します。
- **root** – デフォルトでは、 ネストした対話が終了すると対話のスタックを単純にポップして外側の対話を再開します。 **root=true** を設定することで、 ルートの対話が破棄され、 結果的に対話スタック全体が破棄されるよう指定します。 対話がネストしていなければ現在の対話が破棄します。

@StartTask

```
@StartTask
```

–

jBPM タスクを開始します。 このメソッドが例外を送出することなく null 以外の結果を返すと長期実行の対話を開始することを指定します。 この対話は指定の要求パラメータ中で指定される jBPM タスクと関連しています。 この対話のコンテキスト内で、 タスクインスタンスのビジネスプロセスインスタンスに対して、 ビジネスプロセスコンテキストも定義されます。

- jBPM **TaskInstance** は **taskInstance** 要求コンテキスト変数で利用可能です。 jBPM **ProcessInstance** は **processInstance** 要求コンテキスト変数で利用可能です。 これらのオブジェクトは **@In** でインジェクションが可能です。
- **taskIdParameter** – タスク ID を保持する要求パラメータの名前です。 デフォルトは **"taskId"** です。これは Seam **taskList** JSF コンポーネントによりデフォルトとしても使用されます。
- **flushMode** – この対話で作成される Seam 管理の Hibernate セッションまたは JPA 永続コンテキストのフラッシュモードを設定します。

@BeginTask

```
@BeginTask
```

完了していない jBPM タスクの処理を再開します。 このメソッドが例外を送出せず null 以外の結果を返すと長期実行の対話が開始することを指定します。 この対話は指定の要求パラメータ中で指定される jBPM タスクと関連しています。 この対話のコンテキスト内で、 タスクインスタンスのビジネスプロセスインスタンスに対して、 ビジネスプロセスコンテキストも定義されます。

- jBPM **org.jbpm.taskmgmt.exe.TaskInstance** は **taskInstance** の要求コンテキスト変数で利用可能です。 jBPM **org.jbpm.graph.exe.ProcessInstance** は、 **processInstance** の要求コンテキスト変数で利用可能です。
- **taskIdParameter** – タスクの ID を保持する要求パラメータの名前です。 デフォルトは **"taskId"** です。 これは Seam **taskList** JSF コンポーネントによりデフォルトとしても使用されます。
- **flushMode** – この対話で作成される Seam 管理の Hibernate セッションまたは JPA 永続コンテキストのフラッシュモードを設定します。

@EndTask

```
@EndTask
```

jBPM タスクを終了します。 このメソッドが null 以外の結果を返すと長期実行の対話は終了し、 現在のタスクが完了することを指定します。 jBPM 遷移を引き起こします。 アプリケーションが **transition** と呼ばれる組み込みコンポーネントの**Transition.setName()** を呼んでいない限り、 引き起こされる実際の遷移はデフォルトの遷移になります。

```
@EndTask(transition="transitionName")
```

指定された jBPM 遷移を引き起こします。

- **transition** – タスクの終了時に引き起こされる jBPM 遷移の名前です。デフォルト設定でデフォルトの遷移となっています。

- **beforeRedirect** – デフォルトでは、リダイレクトが発生するまでこの対話は実際には破棄されません。**beforeRedirect=true** を設定することで、現在の要求の最後に対話が破棄され、リダイレクトは新しい一時的な対話コンテキストで処理されるよう指定します。

@CreateProcess

```
@CreateProcess(definition="process definition name")
```

メソッドが例外を送出せずに null 以外の結果を返すとき、新しい jBPM プロセスインスタンスを作成します。**ProcessInstance** オブジェクトは **processInstance** というコンテキスト変数で使用できます。

- **definition** – **org.jboss.seam.bpm.jbpm.processDefinitions** によってデプロイされる jBPM プロセス定義の名前です。

@ResumeProcess

```
@ResumeProcess(processIdParameter="processId")
```

メソッドが例外を送出せずに null 以外の結果を返すとき、既存の jBPM プロセスインスタンスのスコープに再度入ります。**ProcessInstance** オブジェクトは **processInstance** というコンテキスト変数で使用できます。

- **processIdParameter** – プロセス ID を保持する要求パラメータの名前です。デフォルトは **"processId"** です。

@Transition

```
@Transition("cancel")
```

メソッドが null 以外の結果を返すときは常に現在の jBPM プロセスインスタンス内で遷移にシグナルを送るように、メソッドをマークします。

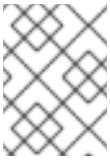
29.5. J2EE 環境で SEAM JAVABEAN コンポーネントを使用するためのアノテーション

Seam は特定のアクションリスナーの結果に対して JTA トランザクションのロールバックを強制するアノテーションを提供します。

@Transactional

```
@Transactional
```

JavaBean コンポーネントにセッション Bean コンポーネントのデフォルト動作と同じようなトランザクション動作を持たせることを指定します。例えば、メソッド呼び出しはトランザクション内で起こるべきであり、メソッドが呼び出されたときにトランザクションが存在しない場合は、トランザクションがそのメソッドのためだけに開始されます。このアノテーションはクラスレベルでもメソッドレベルでも適用可能です。



注記

EJB3 コンポーネントではこのアノテーションではなく、**@TransactionAttribute** を代わりに使用してください。

@ApplicationException

```
@ApplicationException
```

アプリケーション例外でありクライアントに直接報告すべきであることを示す例外に適用されます (ラップされていないということです)。 Java EE 5 より前の環境で使用する場合は **javax.ejb.ApplicationException** とまったく同様に動作します。

注記

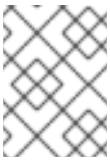
EJB3 コンポーネントではこのアノテーションではなく、**@javax.ejb.ApplicationException** を代わりに使用してください。

- **rollback** – デフォルトでは **false** です。 **true** の場合この例外はトランザクションを rollback only に設定します。
- **end** – デフォルトでは **false** です。 **true** の場合この例外は現在の長期実行の対話を終了します。

@Interceptors

```
@Interceptors({DVDInterceptor, CDInterceptor})
```

クラスまたはメソッドのインターセプタの順序一覧を宣言します。 Java EE 5 より前の環境で使用する場合は **javax.interceptors.Interceptors** とまったく同様に動作します。これはメタアノテーションとしての使用のみに限定してください。



注記

EJB3 コンポーネントではこのアノテーションではなく、**@javax.interceptor.Interceptors** を代わりに使用してください。

これらのアノテーションは主に JavaBean Seam コンポーネントに使用されます。 EJB3 コンポーネントを使う場合は、 標準 Java EE5 アノテーションを使用してください。

29.6. 例外用のアノテーション

これらのアノテーションにより Seam コンポーネントから伝播している例外の処理方法を指定することができます。

@Redirect

```
@Redirect(viewId="error.jsp")
```

このアノテーションが付いている例外によりブラウザが指定されたビュー ID にリダイレクトされることを指定します。

- **viewId** – リダイレクト先となる JSF ビュー ID を指定します。ここで EL を利用することができます。
- **message** – 表示するメッセージです。 デフォルトはその例外メッセージです。
- **end** – 長期実行の対話が終了するよう指定します。 デフォルトは **false** です。

@HttpError

```
@HttpError(errorCode=404)
```

このアノテーションが付いている例外により HTTP エラーが送信されます。

- **errorCode** – HTTP エラーコードです。 デフォルトは **500** です。
- **message** – HTTP エラーで送信されるメッセージです。 デフォルトはその例外メッセージです。
- **end** – 長期実行の対話が終了するよう指定します。 デフォルトは **false** です。

29.7. SEAM REMOTING 用のアノテーション

Seam Remoting は、以下のアノテーションを付けたセッション Bean のローカルインタフェースが必要です。

@WebRemote

```
@WebRemote(exclude="path.to.exclude")
```

クライアント側の JavaScript からアノテーション付きメソッドの呼び出しが可能であることを示します。 **exclude** プロパティはオプションで、 これを使用すると結果のオブジェクトグラフからオブジェクトを除外することができます (詳細は 24章 *リモーティング*の章を参照してください)。

29.8. SEAM インターセプタ用のアノテーション

以下のアノテーションは、Seam インターセプタクラスで使われます。

EJB インターセプタ定義に必要なアノテーションに関する詳細は EJB3 仕様のドキュメントを参照してください。

@Interceptor

```
@Interceptor(stateless=true)
```

このインターセプタはステートレスであることを指定するので、 Seam は複製処理を最適化できます。

```
@Interceptor(type=CLIENT)
```

このインターセプタは EJB コンテナより先に呼ばれる「クライアント側」インターセプタであることを指定します。

```
@Interceptor(around={SomeInterceptor.class, OtherInterceptor.class})
```

このインターセプタは特定のインターセプタよりスタック内でより高い位置に配置されることを指定します。

```
@Interceptor(within={SomeInterceptor.class, OtherInterceptor.class})
```

このインターセプタは特定のインターセプタよりスタック内でより深い位置に配置されることを指定します。

29.9. 非同期用のアノテーション

次のアノテーションは非同期メソッドの宣言に使用されます。 たとえば

```
@Asynchronous public void scheduleAlert(Alert alert,
                                        @Expiration Date date) {
  ...
}
```

```
@Asynchronous public Timer scheduleAlerts(Alert alert,
                                          @Expiration Date date,
                                          @IntervalDuration long
interval) {
  ...
}
```

@Asynchronous

```
@Asynchronous
```

メソッド呼び出しが非同期で処理されることを指定します。

@Duration

```
@Duration
```

非同期呼び出しが処理されるまでの期間に関連するその呼び出しのパラメータを指定します (または反復呼び出しの場合は初めての処理が行われるまで) 。

@Expiration

```
@Expiration
```

非同期呼び出しが処理される (または反復呼び出しの場合は初めての処理が行われる) 日付と時刻に関連するその呼び出しのパラメータを指定します。

@IntervalDuration

```
@IntervalDuration
```

```
@IntervalDuration
```

非同期のメソッド呼び出しが反復することを指定します。 関連付けられたパラメータはその反復間隔の長さを定義します。

29.10. JSF と使用するアノテーション

以下のアノテーションで JSF をより簡単に使えるようになります。

@Converter

Seam コンポーネントが JSF コンバータとして動作できるようにします。アノテーションを付けられたクラスは Seam コンポーネントでなければいけません。また **javax.faces.convert.Converter** を実装しなければなりません。

- **id** – JSF コンバータの ID です。 デフォルトはコンポーネント名です。
- **forClass** – 指定されると、 このコンポーネントをある型のデフォルトコンバータとして登録します。

@Validator

Seam コンポーネントが JSF バリデータとして動作できるようにします。アノテーションを付けられたクラスは Seam コンポーネントでなければいけません。また **javax.faces.validator.Validator** を実装しなければなりません。

- **id** – JSF バリデータの ID です。 デフォルトはコンポーネント名です。

29.10.1. dataTable と使用するアノテーション

以下のアノテーションはステートフルセッション Bean を使ったクリック可能リストの実装を容易にします。 これらのアノテーションは属性に付与されます。

@DataModel

```
@DataModel("variableName")
```

List、 **Map**、 **Set** または **Object[]** 型のプロパティを JSF **DataModel** として所有しているコンポーネントのスコープ (所有しているコンポーネントが **STATELESS** の場合は **EVENT** スコープ) へアウトジェクトします 。 **Map** の場合、 **DataModel** の各行は **Map.Entry** です。

- **value** – 対話コンテキスト変数の名前です。 デフォルトは属性の名前です。
- **scope** – **scope=ScopeType.PAGE** が明示的に指定されると、 **DataModel** は**PAGE** コンテキストに保持されるようになります。

@DataModelSelection

```
@DataModelSelection
```

JSF **DataModel** から選択された値をインジェクトします (これは基礎となるコレクションのエレメントまたはマップ値です)。 コンポーネントにひとつしか **@DataModel** 属性が定義されていなければ、 その **DataModel** から選択された値がインジェクトされます。これ以外は、 各**@DataModel**

のコンポーネント名を各 **@DataModelSelection** の value 属性に指定しなければなりません。

関連付けられた **@DataModel** に **PAGE** スコープが指定されると、 DataModel Selection がインジェクトされるのに加えて関連付けられた DataModel もインジェクトされます。このとき、 **@DataModel** のアノテーションが付いたプロパティが getter メソッドである場合、 プロパティの setter メソッドも含まれている Seam Component を含む Business API の一部でなければなりません。

- **value** – 対話コンテキスト変数の名前です。 コンポーネントに 1 つの**@DataModel** しかない場合は不要です。

@DataModelSelectionIndex

```
@DataModelSelectionIndex
```

JSF **DataModel** の選択インデックスをコンポーネントの属性として公開します (これは基礎となるコレクションの行番号またはマップキーです)。 1 コンポーネントにひとつしか **@DataModel** 属性が定義されていなければ、 その **DataModel** から選択された値がインジェクトされます。 これ以外は、 各 **@DataModel** のコンポーネント名を各 **@DataModelSelectionIndex** の value 属性に指定する必要があります。

- **value** – 対話コンテキスト変数の名前です。 コンポーネントに 1 つの**@DataModel** しかない場合は不要です。

29.11. データバインディング用のメタアノテーション

これらのメタアノテーションは、リスト以外のデータ構造に対して **@DataModel** や **@DataModelSelection** と同様の機能の実装を可能にします。

@DataBinderClass

```
@DataBinderClass(DataModelBinder.class)
```

アノテーションがデータバインディングのアノテーションであることを指定します。

@DataSelectorClass

```
@DataSelectorClass(DataModelSelector.class)
```

アノテーションがデータ選択のアノテーションであることを指定します。

29.12. パッケージング用のアノテーション

このアノテーションは、 一緒にパッケージングするコンポーネントセットに関する情報を宣言するメカニズムを提供します。 どの Java パッケージに対しても適用できます。

@Namespace

```
@Namespace(value="http://jboss.com/products/seam/example/seampay")
```

現在のパッケージにあるコンポーネントが特定の名前空間に関連付けられることを指定します。 宣言された名前空間は **components.xml** ファイル内で XML 名前空間として使用することでアプリケーションの設定を簡略化することができます。

```
@Namespace(value="http://jboss.com/products/seam/core",
                   prefix="org.jboss.seam.core")
```

名前空間を特定のパッケージに関連付けるよう指定します。さらに、あるコンポーネントの名前のプレフィックスが XML ファイルで指定されたコンポーネント名に適用されることを指定します。 たとえば、 この名前空間に関連付けられる **init** という XML 要素は実際には **org.jboss.seam.core.init** というコンポーネントを参照するように解釈されます。

29.13. SERVLET コンテナと統合するためのアノテーション

これらのアノテーションを使用して Seam コンポーネントを Servlet コンテナに統合することができます。

@Filter

javax.servlet.Filter を実装している Seam コンポーネントにアノテーションを付与するために使用する場合、 Seam のマスターフィルタで実行されるサーブレットフィルタとしてそのコンポーネントを指定します。

- ```
  @Filter(around={"seamComponent", "otherSeamComponent"})
  ```

  このフィルタは特定のフィルタよりスタック内でより高い位置に配置されることを指定します。

- ```
  @Filter(within={"seamComponent", "otherSeamComponent"})
  ```

  このフィルタは特定のフィルタよりスタック内でより深い位置に配置されることを指定します。

第30章 組み込み SEAM コンポーネント

本章では Seam の組み込みコンポーネントおよびその設定プロパティについて説明していきます。 組み込みコンポーネントは **components.xml** ファイルに記載されていなくても自動的に作成されます。 ただし、デフォルトのプロパティを上書き、または特定タイプのコンポーネントを複数指定する必要がある場合は **components.xml** を使用します。

@Name を使って適切な組み込みコンポーネントにちなみ独自のクラスに名前を付けるだけでいずれの組み込みコンポーネントも独自の実装に置き換えることができます。

30.1. コンテキストインジェクションのコンポーネント

最初の組み込みコンポーネントセットは各種のコンテキスト依存オブジェクトのインジェクションをサポートしています。 たとえば、 次のコンポーネントインスタンスの変数により Seam セッションのコンテキストオブジェクトがインジェクトされます。

```
@In private Context sessionContext;
```

org.jboss.seam.core.contexts

Seam Context オブジェクトへのアクセスを提供するコンポーネントです。 たとえば、 **org.jboss.seam.core.contexts.sessionContext['user']** など。

org.jboss.seam.faces.facesContext

FacesContext コンテキストオブジェクトのマネージャコンポーネントです (正確には Seam コンテキストではありません)。

これらコンポーネントはすべて常にインストールされます。

30.2. JMS 関連のコンポーネント

次のコンポーネントセットにより JSF を補完します。

org.jboss.seam.faces.dateConverter

タイプ **java.util.Date** のプロパティ用のデフォルト JSF コンバータを提供します。

このコンバータは自動的に JSF に登録されるため、 開発者は入力フィールドやページパラメータに DateTimeConverter を指定する必要がありません。 デフォルトではタイプは日付 (時間や日時とは対照的に) と仮定して、 ユーザーの **Locale** に調整された短い入力スタイルを使用します。 **Locale.US** の場合、 入力パターンは **mm/dd/yy** です。 ただし、 Y2K に準拠するため、 年は 2 桁から 4 桁に変更されます (**mm/dd/yyyy**)。

コンポーネントを再設定することで入力パターンをグローバルに上書きすることができます。 このクラスに関する JavaServer Faces ドキュメントでサンプルを参照してください。

org.jboss.seam.faces.facesMessages

Faces がブラウザリダイレクト全体に成功のメッセージを伝播できるようにします。

- **add(FacesMessage facesMessage)** – Faces メッセージを追加します。 現在の対話内で発生する次のレスポンス出力フェーズで表示されます。

- **add(String messageTemplate)** – Faces メッセージを追加します。EL 式を含むことができる特定のメッセージテンプレートからレンダリングされます。
- **add(Severity severity, String messageTemplate)** – Faces メッセージを追加します。EL 式を含むことができる特定のメッセージテンプレートからレンダリングされます。
- **addFromResourceBundle(String key)** – Faces メッセージを追加します。EL 式を含むことができる Seam リソースバンドルで定義されたメッセージテンプレートからレンダリングされます。
- **addFromResourceBundle(Severity severity, String key)** – Faces メッセージを追加します。EL 式を含むことができる Seam リソースバンドルで定義されたメッセージテンプレートからレンダリングされます。
- **clear()** – 全メッセージを消去します。

org.jboss.seam.faces.redirect

パラメータ付きでリダイレクトを行う場合に便利な API です。 特にブックマーク可能な検索結果画面などに役立ちます。

- **redirect.viewId** – リダイレクト先となる JSF ビュー ID です。
- **redirect.conversationPropagationEnabled** – 対話をリダイレクト全体に伝播させるかどうか決定します。
- **redirect.parameters** – 値への要求パラメータ名のマップで、リダイレクト要求に渡されます。
- **execute()** – リダイレクトを直ちに実行します。
- **captureCurrentRequest()** – 現在の GET 要求 (対話コンテキスト内) の要求パラメータとビュー ID を格納します。 **execute()** の呼び出しで後ほど使用されます。

org.jboss.seam.faces.httpError

HTTP エラーを送信する場合に便利な API です。

org.jboss.seam.ui.renderStampStore

レンダリングスタンプのコレクションを管理するコンポーネントです。 レンダリングスタンプはレンダリングされたフォームがサブミットされたかどうかを示します。 特に JSF のクライアント側の状態保存のメソッドと併用すると便利です。 フォームのステータス (ポストされたまたはされていない) はクライアントではなくサーバーによって制御されているためです。

クライアント側の状態保存はセッションからこのチェックのバインディングを外すためによく使用されます。 これを行うにはそのアプリケーション (アプリケーションが実行中は有効) またはデータベース (サーバーが再起動されても有効) 内にレンダリングスタンプを保存できる実装が必要となります。

- **maxSize** – ストアに保持可能なスタンプの最大数です。 デフォルトは **100** です。

JSF コンポーネントはクラスパスに **javax.faces.context.FacesContext** クラスが使用可能な場合にインストールされます。

30.3. ユーティリティコンポーネント

次のコンポーネントはさまざまなアプリケーションに幅広く便利な各種機能を提供します。

org.jboss.seam.core.events

@Observer のメソッドまたは **components.xml** 内のメソッドバインディングで監視できるイベントを引き起こす API です。

- **raiseEvent(String type)** – 特定タイプのイベントを発生させてすべての監視者に配信します。
- **raiseAsynchronousEvent(String type)** – EJB3 タイマーサービスで非同期に処理されるイベントを発生させます。
- **raiseTimedEvent(String type, ...)** – EJB3 タイマーサービスで非同期に処理されるイベントをスケジュールします。
- **addListener(String type, String methodBinding)** – 特定イベントタイプの監視者を追加します。

org.jboss.seam.core.interpolator

文字列内に JFS EL 式の値を挿入するための API です。

- **interpolate(String template)** – **#{...}** 形式の JSF EL 式のテンプレートをスキャンしてその評価値と置換します。

org.jboss.seam.core.expressions

値とメソッドのバインティングを作成するための API です。

- **createValueBinding(String expression)** – 値バインディングのオブジェクトを作成します。
- **createMethodBinding(String expression)** – メソッドバインディングのオブジェクトを作成します。

org.jboss.seam.core.pojoCache

JBoss Cache **PojoCache** インスタンスのマネージャコンポーネントです。

- **pojoCache.cfgResourceName** – 設定ファイル名です。 デフォルトでは **treecache.xml** に設定されます。

これらコンポーネントはすべて常にインストールされます。

30.4. 国際化とテーマのコンポーネント

次のコンポーネントにより簡単に Seam を使用した国際化ユーザーインターフェースを構築できます。

org.jboss.seam.core.locale

Seam ロケールです。

org.jboss.seam.international.timezone

Seam のタイムゾーンです。 タイムゾーンはセッションスコープです。

org.jboss.seam.core.resourceBundle

Seam リソースバンドルです。 リソースバンドルはステートレスです。 Seam リソースバンドルは Java リソースバンドル一覧内にあるキーに深さ優先検索を行います。

org.jboss.seam.core.resourceLoader

リソースローダーはアプリケーションリソースおよびリソースバンドルへのアクセスを提供します。

- **resourceLoader.bundleNames** – Seam リソースバンドルを使用する場合に検索する Java リソースバンドルの名前です。 デフォルトは **messages** です。

org.jboss.seam.international.localeSelector

設定時間またはランタイム時のユーザーのいずれかでロケール選択をサポートします。

- **select()** – 指定されたロケールを選択します。
- **localeSelector.locale** – 実際の **java.util.Locale** です。
- **localeSelector.localeString** – ロケールの文字列表現です。
- **localeSelector.language** – 指定されたロケールの言語です。
- **localeSelector.country** – 指定されたロケールの国です。
- **localeSelector.variant** – 指定されたロケールのバリアントです。
- **localeSelector.supportedLocales** – **jsf-config.xml** に記載のサポートロケールを表している **SelectItem** 一覧です。
- **localeSelector.cookieEnabled** – ロケール選択がクッキーで永続化されることを指定します。

org.jboss.seam.international.timezoneSelector

設定時間またはランタイム時のユーザーのいずれかでタイムゾーン選択をサポートします。

- **select()** – 指定されたロケールを選択します。
- **timezoneSelector.timezone** – 実際の **java.util.TimeZone** です。
- **timezoneSelector.timeZoneId** – タイムゾーンの文字列表現です。
- **timezoneSelector.cookieEnabled** – タイムゾーン選択がクッキーによって永続化されることを指定します。

org.jboss.seam.international.messages

Seam リソースバンドル内で定義されるメッセージテンプレートからレンダリングした国際化メッセージを含んでいるマップです。

org.jboss.seam.theme.themeSelector

設定時間またはランタイム時のユーザーのいずれかでテーマ選択をサポートします。

- **select()** – 指定されたテーマを選択します。
- **theme.availableThemes** – 定義されたテーマの一覧です。
- **themeSelector.theme** – 選択されたテーマです。
- **themeSelector.themes** – 定義されたテーマを表している **SelectItem** の一覧です。
- **themeSelector.cookieEnabled** – テーマ選択がクッキーで永続化されることを指定します。

org.jboss.seam.theme.theme

テーマエントリを含んでいるマップです。

これらコンポーネントはすべて常にインストールされます。

30.5. 対話を制御するためのコンポーネント

次のコンポーネントグループを使うとアプリケーションまたはユーザーインターフェースにより対話の制御を行うことができます。

org.jboss.seam.core.conversation

アプリケーション内から現在の Seam 対話の属性を制御するための API です。

- **getId()** – 現在の対話 ID を返します。
- **isNested()** – 現在の対話がネストされている対話かどうかを指定します。
- **isLongRunning()** – 現在の対話が長期実行の対話であるかどうかを指定します。
- **getId()** – 現在の対話 ID を返します。
- **getParentId()** – 親対話の対話 ID を返します。
- **getRootId()** – ルート対話の対話 ID を返します。
- **setTimeout(int timeout)** – 現在の対話のタイムアウトを設定します。
- **setViewId(String outcome)** – 対話スイッチャー、 対話リストまたはブレッドクラムから現在の対話に切り替えて戻した場合に使用する ビュー ID を設定します。
- **setDescription(String description)** – 対話スイッチャー、 対話リストまたはブレッドクラムで表示される現在の対話の詳細を設定します。
- **redirect()** – この対話用に明確に定義された最後のビュー ID にリダイレクトします。 ログイン試行後に使うと便利です。
- **leave()** – 実際には対話を終了せずにこの対話のスコープを終了します。
- **begin()** – 長期実行の対話を開始します (**@Begin** と同等)。
- **beginPageflow(String pageflowName)** – ページフローを付けて長期実行の対話を開始します (**@Begin(pageflow="...")** と同等)。

- **end()** – 長期実行の対話を終了します (**@End** と同等)。
- **pop()** – 対話スタックをポップして親対話に戻ります。
- **root()** – 対話スタックのルート対話に戻ります。
- **changeFlushMode(FlushModeType flushMode)** – 対話のフラッシュモードを変更します。

org.jboss.seam.core.conversationList

対話一覧のマネージャコンポーネントです。

org.jboss.seam.core.conversationStack

対話スタックのマネージャコンポーネントです (ブレッドクラム)。

org.jboss.seam.faces.switcher

対話スイッチャーです。

これらコンポーネントはすべて常にインストールされます。

30.6. JBPM 関連のコンポーネント

以下は jBPM と併用するコンポーネントです。

org.jboss.seam.pageflow.pageflow

Seam ページフロー制御用の API です。

- **isInProcess()** – 現在プロセス中のページフローがある場合には **true** を返します。
- **getProcessInstance()** – 現在のページフローの jBPM **ProcessInstance** を返します。
- **begin(String pageflowName)** – 現在の対話のコンテキスト内でページフローを開始します。
- **reposition(String nodeName)** – 現在のページフローを特定ノードに再配置します。

org.jboss.seam.bpm.actor

現在のセッションに関連付けられた jBPM actor の属性をアプリケーション内から制御する API です。

- **setId(String actorId)** – 現在のユーザーの jBPM actor ID を設定します。
- **getGroupActorIds()** – 現在のユーザーグループ群用の jBPM actor ID が追加される **Set** を返します。

org.jboss.seam.bpm.transition

現在のタスクの jBPM 遷移をアプリケーション内から制御する API です。

- **setName(String transitionName)** – 現在のタスクが **@EndTask** で終了される場合に使用する jBPM 遷移名を設定します。

org.jboss.seam.bpm.businessProcess

対話とビジネスプロセス間の関連性をプログラム制御するための API です。

- **businessProcess.taskId** – 現在の対話に関連付けられたタスクの ID です。
- **businessProcess.processId** – 現在の対話に関連付けられたプロセスの ID です。
- **businessProcess.hasCurrentTask()** – タスクインスタンスを現在の対話に関連付けるかどうかを指定します。
- **businessProcess.hasCurrentProcess()** – プロセスインスタンスを現在の対話に関連付けるかどうかを指定します。
- **createProcess(String name)** – 名前付きプロセスの定義のインスタンスを作成し現在の対話に関連付けます。
- **startTask()** – 現在の対話に関連付けられたタスクを開始します。
- **endTask(String transitionName)** – 現在の対話に関連付けられたタスクを終了します。
- **resumeTask(Long id)** – 特定の ID を持つタスクを現在の対話に関連付けます。
- **resumeProcess(Long id)** – 特定の ID を持つプロセスを現在の対話に関連付けます。
- **transition(String transitionName)** – 遷移を引き起こします。

org.jboss.seam.bpm.taskInstance

jBPM **TaskInstance** のマネージャコンポーネントです。

org.jboss.seam.bpm.processInstance

jBPM **ProcessInstance** のマネージャコンポーネントです。

org.jboss.seam.bpm.jbpmContext

イベントスコープ **JbpmContext** のマネージャコンポーネントです。

org.jboss.seam.bpm.taskInstanceList

jBPM タスクリストのマネージャコンポーネントです。

org.jboss.seam.bpm.pooledTaskInstanceList

jBPM プールされたタスクリストのマネージャコンポーネントです。

org.jboss.seam.bpm.taskInstanceListForType

jBPM タスクリスト集のマネージャコンポーネントです。

org.jboss.seam.bpm.pooledTask

プールされたタスク割り当てのアクションハンドラです。

org.jboss.seam.bpm.processInstanceFinder

プロセスインスタンスのタスクリストのマネージャコンポーネントです。

org.jboss.seam.bpm.processInstanceList

プロセスインスタンスのタスクリストです。

org.jboss.seam.bpm.jbpm コンポーネントがインストールされると常にこれらの全コンポーネントがインストールされます。

30.7. セキュリティ関連のコンポーネント

これらのコンポーネントはウェブ層のセキュリティに関連しています。

org.jboss.seam.web.userPrincipal

現在のユーザー **Principal** のマネージャコンポーネントです。

org.jboss.seam.web.isUserInRole

現在の principal に使用可能なロールに応じて JSF ページがコントロールのレンダリングを選択できるようにします。例えば **<h:commandButton value="edit" rendered="#{isUserInRole['admin']}"/>** です。

30.8. JMS 関連のコンポーネント

これらのコンポーネントは管理の **TopicPublisher** および **QueueSender** との併用を目的としています (下記参照)。

org.jboss.seam.jms.queueSession

JMS **QueueSession** のマネージャコンポーネントです。

org.jboss.seam.jms.topicSession

JMS **TopicSession** のマネージャコンポーネントです。

30.9. メール関連のコンポーネント

これらのコンポーネントは Seam の Email サポートと併用します。

org.jboss.seam.mail.mailSession

JavaMail **Session** のマネージャコンポーネントです。 セッションを JNDI コンテキスト内で検索させるか (**sessionJndiName** プロパティを設定)、 設定オプションから作成することができます。後者の場合、 **host** は必須です。

- **org.jboss.seam.mail.mailSession.host** – 使用する SMTP サーバーのホスト名です。
- **org.jboss.seam.mail.mailSession.port** – 使用する SMTP サーバーのポートです。
- **org.jboss.seam.mail.mailSession.username** – SMTP サーバーの接続に使用するユーザー名です。
- **org.jboss.seam.mail.mailSession.password** – SMTP サーバーの接続に使用するパスワードです。

- **org.jboss.seam.mail.mailSession.debug** – JavaMail のデバッグ機能を有効にします (かなり冗長です)。
- **org.jboss.seam.mail.mailSession.ssl** – SMTP への SSL 接続を有効にします (デフォルトはポート 465)。

  org.jboss.seam.mail.mailSession.tls – メールセッションで TLS サポートを有効にします。デフォルトは **true** です。
- **org.jboss.seam.mail.mailSession.sessionJndiName** – JNDI にバインドされる javax.mail.Session と同じ名前です。これが与えられると他のプロパティはすべて無視されます。

30.10. 基盤となるコンポーネント

これらのコンポーネントは非常に重要なプラットフォームの基盤を提供します。 デフォルトではインストールされないコンポーネントは、**components.xml** 内のそのコンポーネントで**install="true"** を設定するとインストールすることができます。

org.jboss.seam.core.init

このコンポーネントは Seam の初期化設定を含んでいます。 常にインストールされます。

- **org.jboss.seam.core.init.jndiPattern** – セッション Bean の検索に使用される JNDI パターンです。
- **org.jboss.seam.core.init.debug** – Seam デバッグモードを有効にします。 実稼働では **false** に設定してください。 デバッグが有効になっている状態でシステムに負荷がかかるとエラーが表示される場合があります。
- **org.jboss.seam.core.init.clientSideConversations** – **true** に設定すると Seam は対話のコンテキスト変数を **HttpSession** 内ではなくクライアント内に保存します。

org.jboss.seam.core.manager

Seam ページおよび対話コンテキスト管理用の内部コンポーネントです。 常にインストールされます。

- **org.jboss.seam.core.manager.conversationTimeout** – ミリ秒単位で対話コンテキストのタイムアウトを設定します。
- **org.jboss.seam.core.manager.concurrentRequestTimeout** – 長期実行の対話コンテキストでロックの取得を試行しているスレッドの最大待機時間です。
- **org.jboss.seam.core.manager.conversationIdParameter** – 対話 ID の伝播に使用する要求パラメータです。 デフォルトは **conversationId** です。
- **org.jboss.seam.core.manager.conversationIsLongRunningParameter** – 対話が長期実行であることを伝播するために使用する要求パラメータです。 デフォルトは **conversationIsLongRunning** です。
- **org.jboss.seam.core.manager.defaultFlushMode** – すべての Seam 管理永続コンテキストにデフォルトで設定されるフラッシュモードを設定します。 デフォルトでは **AUTO** に設定されます。

org.jboss.seam.navigation.pages

Seam ワークスペース管理用の内部コンポーネントです。 常にインストールされます。

- **org.jboss.seam.navigation.pages.noConversationViewId** – 対話エントリがサーバー側で見つからない場合にグローバルなリダイレクト先となるビュー ID を指定します。
- **org.jboss.seam.navigation.pages.loginViewId** – 未承認ユーザーが保護されたビューへのアクセスを試行する場合にグローバルなリダイレクト先となるビュー ID を指定します。
- **org.jboss.seam.navigation.pages.httpPort** – HTTP スキームが要求された場合にグローバルに使用するポートを指定します。
- **org.jboss.seam.navigation.pages.httpsPort** – HTTPS スキームが要求された場合にグローバルに使用するポートを指定します。
- **org.jboss.seam.navigation.pages.resources** – **pages.xml** スタイルのリソースを検索するリソース一覧です。 デフォルトは **WEB-INF/pages.xml** です。

org.jboss.seam.bpm.jbpm

このコンポーネントは **JbpmConfiguration** をブートストラップします。 クラス **org.jboss.seam.bpm.Jbpm** としてインストールします。

- **org.jboss.seam.bpm.jbpm.processDefinitions** – ビジネスプロセスの編成に使用する jPDL ファイルのリソース名一覧を指定します。
- **org.jboss.seam.bpm.jbpm.pageflowDefinitions** – 対話ページフローの編成に使用する jPDL ファイルのリソース名一覧を指定します。

org.jboss.seam.core.conversationEntries

要求間のアクティブな長期実行の対話を記録するセッションスコープの内部コンポーネントです。

org.jboss.seam.faces.facesPage

ページに関連付けられた対話コンテキストを記録するページスコープの内部コンポーネントです。

org.jboss.seam.persistence.persistenceContexts

現在の対話に使用された永続コンテキストを記録する内部コンポーネントです。

org.jboss.seam.jms.queueConnection

JMS **QueueConnection** を管理します。これは管理 **QueueSender** がインストールされると必ずインストールされます。

- **org.jboss.seam.jms.queueConnection.queueConnectionFactoryJndiName** – JMS **QueueConnectionFactory** の JNDI 名を指定します。 デフォルトは **UIL2ConnectionFactory** です。

org.jboss.seam.jms.topicConnection

JMS **TopicConnection** を管理します。これは管理 **TopicPublisher** がインストールされると必ずインストールされます。

- **`org.jboss.seam.jms.topicConnection.topicConnectionFactoryJndiName`** – JMS **`TopicConnectionFactory`** の JNDI 名を指定します。 デフォルトは **`UIL2ConnectionFactory`** です。

`org.jboss.seam.persistence.persistenceProvider`

JPA プロバイダの標準化されていない機能に対する抽象化レイヤです。

`org.jboss.seam.core.validators`

Hibernate Validator **`ClassValidator`** のインスタンスをキャッシュします。

`org.jboss.seam.faces.validation`

検証が失敗または成功のどちらかをアプリケーションが判断できます。

`org.jboss.seam.debug.introspector`

Seam Debug Page のサポートです。

`org.jboss.seam.debug.contexts`

Seam Debug Page のサポートです。

`org.jboss.seam.exception.exceptions`

例外処理用の内部コンポーネントです。

`org.jboss.seam.transaction.transaction`

トランザクションを制御し JTA 互換のインターフェースの背後にあるトランザクション管理の基礎となる実装を抽象化するための API です。

`org.jboss.seam.faces.safeActions`

アクション式がビュー内に存在するかどうかを確認することで、着信 URL 内のアクション式の安全性を判断します。

30.11. その他のコンポーネント

追加コンポーネントや未分類のコンポーネントです。

`org.jboss.seam.async.dispatcher`

非同期メソッドのステートレスセッション Bean をディスパッチします。

`org.jboss.seam.core.image`

イメージの操作および問い合わせに使用されます。

`org.jboss.seam.core.pojoCache`

PojoCache インスタンスのマネージャコンポーネントです。

`org.jboss.seam.core.uiComponent`

コンポーネント ID によりキー付けされた UIComponents のマップを管理します。

30.12. 特殊なコンポーネント

特定の Seam コンポーネントのクラスは、Seam 設定内で指定した名前で複数回インストールすることができます。 例えば、 **components.xml** 内の次の行では、Seam コンポーネントを 2 つインストールして、設定します。

```
<component name="bookingDatabase"
           class="org.jboss.seam.persistence.ManagedPersistenceContext">
  <property name="persistenceUnitJndiName">
    java:/comp/emf/bookingPersistence
  </property>
</component>

<component name="userDatabase"
           class="org.jboss.seam.persistence.ManagedPersistenceContext">
  <property name="persistenceUnitJndiName">
    java:/comp/emf/userPersistence
  </property>
</component>
```

Seam コンポーネント名は **bookingDatabase** と **userDatabase** です。

<entityManager>, org.jboss.seam.persistence.ManagedPersistenceContext

拡張永続コンテキストを持つ対話スコープで管理の **EntityManager** のマネージャコンポーネントです。

- **<entityManager>**.entityManagerFactory – **EntityManagerFactory** のインスタンスに評価を行う値バインディング式です。

  <entityManager>.persistenceUnitJndiName – エンティティマネージャファクトリの JNDI 名です。 デフォルトではこれは **java:/<managedPersistenceContext>** です。

<entityManagerFactory>, org.jboss.seam.persistence.EntityManagerFactory

JPA **EntityManagerFactory** を管理します。 EJB3 サポートの環境の外部で JPA を使用する場合に最適となります。

- **entityManagerFactory.persistenceUnitName** – 永続ユニット名です。

設定プロパティの詳細は API JavaDoc をご覧ください。

<session>, org.jboss.seam.persistence.ManagedSession

対話スコープで管理の Hibernate **Session** のマネージャコンポーネントです。

- **<session>**.sessionFactory – **SessionFactory** のインスタンスに評価を行う値バインディング式です。

  <session>.sessionFactoryJndiName – セッションファクトリの JNDI 名です。 デフォルトは **java:/<managedSession>** です。

<sessionFactory>, org.jboss.seam.persistence.HibernateSessionFactory

Hibernate **SessionFactory** を管理します。

- **<sessionFactory>.cfgResourceName** – 設定ファイルへのパスを指定します。デフォルトは **hibernate.cfg.xml** です。

設定プロパティの詳細は API JavaDoc をご覧ください。

<managedQueueSender>, org.jboss.seam.jms.ManagedQueueSender

イベントスコープで管理の JMS **QueueSender** のマネージャコンポーネントです。

- **<managedQueueSender>**.queueJndiName – JMS キューの JNDI 名です。

<managedTopicPublisher>, org.jboss.seam.jms.ManagedTopicPublisher

イベントスコープで管理の JMS **TopicPublisher** のマネージャコンポーネントです。

- **<managedTopicPublisher>**.topicJndiName – JMS トピックの JMDI 名です。

<managedWorkingMemory>, org.jboss.seam.drools.ManagedWorkingMemory

対話スコープで管理の Drools **WorkingMemory** のマネージャコンポーネントです。

- **<managedWorkingMemory>**.ruleBase – **RuleBase** のインスタンスに評価を行う値式です。

<ruleBase>, org.jboss.seam.drools.RuleBase

アプリケーションスコープの Drools **RuleBase** のマネージャコンポーネントです。 新しいルールの動的なインストールには対応しないため、 実稼働使用には適さない点に注意してください。

- **<ruleBase>**.ruleFiles – Drools のルール郡を含むファイルの一覧です。

  <ruleBase>.dslFile – Drools DSL 定義です。

第31章 SEAM JSF コントロール

Seam には組み込みコントロールや他のサードパーティライブラリのコントロールの機能を補完する JavaServer Faces (JSF) コントロールが多く含まれています。 Seam と併用する場合は、JBoss RichFaces および Apache MyFaces Trinidad タグライブラリの使用を推奨します。 Tomahawk タグライブラリの使用はお薦めできません。

31.1. タグ

これらのタグを使用するには、 以下のように使用するページで **s** 名前空間を定義します (Facelets のみ)。

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:s="http://jboss.com/products/seam/taglib">
```

ユーザーインターフェースのサンプルではいくつかのタグの使用例を示しています。

31.1.1. ナビゲーションコントロール

31.1.1.1. <s:button>

説明

対話の伝搬を制御することでアクションの呼び出しをサポートするボタンです。 **このボタンではフォームはサブミットしません。**

Attributes

- **value** – ボタンのラベルです。
- **action** – アクションリスナーを指定するメソッドバインディングです。
- **view** – リンク先となる JSF ビュー ID を指定します。
- **fragment** – リンク先となるフラグメント識別子を指定します。
- **disabled** – リンクを無効にするかどうか指定します。
- **propagation** – 対話の伝播方式を指定します、 **begin**、 **join**、 **nest**、 **none**、 **end** があります。
- **pageflow** – 開始するページフロー定義を指定します (**propagation="begin"** または **propagation="join"** が使用される場合のみ有効)。

使用方法

```
<s:button id="cancel" value="Cancel" action="#{hotelBooking.cancel}"/>
```

<s:link /> には、 **view** と **action** の両方が指定可能です。 この場合、指定されたビューへのリダイレクトが発生した時点でアクションが呼び出されます。

<s:button />ではアクションリスナー (デフォルトの JSF アクションリスナーも含む) の使用はサポートされていません。

31.1.1.2. <s:conversationId>

説明

対話 ID を JSF リンクまたはボタンに追加します。例えば以下のとおりです。

<h:commandLink />, <s:button />.

Attributes

なし

31.1.1.3. <s:taskId>

説明

タスクが **#{task}** で利用可能な場合、 出力リンク (または同様の JSF コントロール) にタスク ID を付加します。

Attributes

なし

31.1.1.4. <s:link>

説明

対話の伝搬を制御することでアクションの呼び出しをサポートするリンクです。 **このボタンではフォームはサブミットしません。**

<s:link />ではアクションリスナー (デフォルトの JSF アクションリスナーも含む) の使用はサポートされていません。

Attributes

- **value** – リンクラベルを指定します。
- **action** – アクションリスナーを指定するメソッドバインディングです。
- **view** – リンク先となる JSF ビュー ID を指定します。
- **fragment** – リンク先となるフラグメント識別子を指定します。
- **disabled** – リンクを無効にするかどうか指定します。
- **propagation** – 対話の伝播方式を指定します、 **begin**、 **join**、 **nest**、 **none**、 **end** があります。
- **pageflow** – 開始するページフロー定義を指定します (**propagation="begin"** または **propagation="join"** が使用される場合のみ有効)。

使用方法

```
<s:link id="register" view="/register.xhtml" value="Register New User"/>
```

<s:link /> には、 **view** と **action** の両方が指定可能です。 この場合、指定されたビューへのリダイレクトが発生した時点でアクションが呼び出されます。

31.1.1.5. <s:conversationPropagation>

説明

コマンドリンクやボタン (または同様の JSF コントロール) に対し対話の伝搬をカスタマイズします。 **Facelets のみです。**

Attributes

- **type** – 対話の伝播方式を指定します、 **begin**、 **join**、 **nest**、 **none end** があります。
- **pageflow** – 開始するページフロー定義を指定します (**propagation="begin"** または **propagation="join"** が使用される場合のみ有効)。

使用方法

```
<h:commandButton value="Apply" action="#{personHome.update}">
  <s:conversationPropagation type="join" />
</h:commandButton>
```

31.1.1.6. <s:defaultAction>

説明

enter キーでフォームをサブミットしたときに実行するデフォルトのアクションを指定します。

現時点では、 **<h:commandButton />**、 **<a:commandButton />**、 **<tr:commandButton />** など、 ボタンの内側にのみネストが可能です。

アクションソースには ID を指定する必要があり、 1 つのフォームに指定できるのはデフォルトの 1 アクションのみとなります。

Attributes

なし

使用方法

```
<h:commandButton id="foo" value="Foo" action="#{manager.foo}">
  <s:defaultAction />
</h:commandButton>
```

31.1.2. コンバータとバリデータ

31.1.2.1. <s:convertDateTime>

説明

Seam タイムゾーン内でデータ変換または時刻変換を行います。

Attributes

なし

使用方法

```
<h:outputText value="#{item.orderDate}">
  <s:convertDateTime type="both" dateStyle="full"/>
</h:outputText>
```

31.1.2.2. <s:convertEntity>

説明

エンティティコンバータを現在のコンポーネントに割り当てます。 ラジオボタンコントロールおよびドロップダウンコントロールに役立ちます。

コンバータは単純なエンティティ、複合エンティティなどすべての管理エンティティとも動作します。フォームのサブミット時にコンバータが JSF コントロールで宣言された項目を発見できないと、検証エラーが発生します。

Attributes

なし

設定

<s:convertEntity /> は **Seam 管理のトランザクション** (「Seam 管理トランザクション」 参照) とともに使う必要があります。

管理永続コンテキスト は **entityManager** という名前でなければなりません。 この名前が付いていない場合は **components.xml** で変更します。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">
<ui:jpa-entity-loader entity-manager="#{em}" />
```

管理 Hibernate セッション を使用している場合は、 それを **components.xml** でも設定する必要があります。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">
<ui:hibernate-entity-loader />
```

管理 Hibernate セッション は **session** という名前でなければなりません。 この名前が付いていない場合は **components.xml** で変更します。

```
<components xmlns="http://jboss.com/products/seam/components"
            xmlns:ui="http://jboss.com/products/seam/ui">
<ui:hibernate-entity-loader session="#{hibernateSession}" />
```

このエンティティコンバータで複数のエンティティマネージャを使用する場合は、 **components.xml** でそれぞれのエンティティマネージャに対してこのエンティティコンバータのコピーを作成します。 エンティティコンバータはエンティティローダーに委譲して次のような永続化操作を行います。

```
<components xmlns="http://jboss.com/products/seam/components"
```

```
            xmlns:ui="http://jboss.com/products/seam/ui">
<ui:entity-converter name="standardEntityConverter"
    entity-loader="#{standardEntityLoader}" />
<ui:jpa-entity-loader name="standardEntityLoader"
    entity-manager="#{standardEntityManager}" />
<ui:entity-converter name="restrictedEntityConverter"
    entity-loader="#{restrictedEntityLoader}" />
<ui:jpa-entity-loader name="restrictedEntityLoader"
    entity-manager="#{restrictedEntityManager}" />
```

```
<h:selectOneMenu value="#{person.continent}">
  <s:selectItems value="#{continents.resultList}"
     var="continent" label="#{continent.name}" />
  <f:converter converterId="standardEntityConverter" />
</h:selectOneMenu>
```

使用方法

```
<h:selectOneMenu value="#{person.continent}" required="true">
  <s:selectItems value="#{continents.resultList}" var="continent"
     label="#{continent.name}" noSelectionLabel="Please Select..."/>
  <s:convertEntity />
</h:selectOneMenu>
```

31.1.2.3. <s:convertEnum>

説明

enum コンバータを現在のコンポーネントに割り当てます。主にラジオボタンコントロールおよびドロップダウンコントロールに役立ちます。

Attributes

なし

使用方法

```
<h:selectOneMenu value="#{person.honorific}">
  <s:selectItems value="#{honorifics}" var="honorific"
     label="#{honorific.label}" noSelectionLabel="Please select" />
  <s:convertEnum />
</h:selectOneMenu>
```

31.1.2.4. <s:convertAtomicBoolean>

説明

java.util.concurrent.atomic.AtomicBoolean 用の **javax.faces.convert.Converter** です。

Attributes

なし

使用方法

```
<h:outputText value="#{item.valid}">
  <s:convertAtomicBoolean />
</h:outputText>
```

31.1.2.5. <s:convertAtomicInteger>

説明

java.util.concurrent.atomic.AtomicInteger 用の **javax.faces.convert.Converter** です。

Attributes

なし

使用方法

```
<h:outputText value="#{item.id}">
  <s:convertAtomicInteger />
</h:outputText>
```

31.1.2.6. <s:convertAtomicLong>

説明

java.util.concurrent.atomic.AtomicLong 用の **javax.faces.convert.Converter** です。

Attributes

なし

使用方法

```
<h:outputText value="#{item.id}">
  <s:convertAtomicLong />
</h:outputText>
```

31.1.2.7. <s:validateEquality>

説明

入力コントロールの親の値が、参照されたコントロールの値と同等かどうかを確認します。

Attributes

- **for** – 検証の対象となるコントロールの ID です。
- **message** – 失敗時に表示されるメッセージです。
- **required** – False は値がフィールドで入力されたかのチェックを無効にします。
- **messageId** – 失敗時に表示するメッセージ ID です。

- **operator** – 値の比較時に使用する演算子です。 有効な演算子は次のとおりです。
  - **equal** – **value.equals(forValue)** を検証します。
  - **not_equal** – **!value.equals(forValue)** を検証します。
  - **greater** – ((Comparable)value).compareTo(forValue) > 0 を検証します。
  - **greater_or_equal** – ((Comparable)value).compareTo(forValue) >= 0 を検証します。
  - **less** – ((Comparable)value).compareTo(forValue) < 0 を検証します。
  - **less_or_equal** – ((Comparable)value).compareTo(forValue) <= 0 を検証します。

使用方法

```
<h:inputText id="name" value="#{bean.name}"/>
<h:inputText id="nameVerification" >
  <s:validateEquality for="name" />
</h:inputText>
```

31.1.2.8. <s:validate>

説明

非視覚的なコントロールです。 Hibernate Validator を使用してバウンドプロパティに対して JSF 入力フィールドを確認します。

Attributes

なし

使用方法

```
<h:inputText id="userName" required="true" value="#{customer.userName}">
  <s:validate />
</h:inputText>
<h:message for="userName" styleClass="error" />
```

31.1.2.9. <s:validateAll>

説明

非視覚的なコントロールです。 Hibernate Validator を使ってそのバウンドプロパティに対しすべての子 JSF 入力フィールドを確認します。

Attributes

なし

使用方法

```
<s:validateAll>
  <div class="entry">
    <h:outputLabel for="username">Username:</h:outputLabel>
```

```
    <h:inputText id="username" value="#{user.username}" required="true"/>
    <h:message for="username" styleClass="error" />
  </div>
  <div class="entry">
    <h:outputLabel for="password">Password:</h:outputLabel>
    <h:inputSecret id="password" value="#{user.password}"
     required="true"/>
    <h:message for="password" styleClass="error" />
  </div>
  <div class="entry">
    <h:outputLabel for="verify">Verify Password:</h:outputLabel>
    <h:inputSecret id="verify" value="#{register.verify}"
     required="true"/>
    <h:message for="verify" styleClass="error" />
  </div>
</s:validateAll>
```

31.1.3. フォーマット

31.1.3.1. <s:decorate>

説明

検証に失敗した場合または **required="true"** に設定されている場合、 JSF 入力フィールドを「装飾」します。

Attributes

- **template** – コンポーネントの装飾に使用される Facelet テンプレートです。
- **enclose** – **true** にすると、 入力フィールドの装飾に使用されたテンプレートが「element」属性で指定されたエレメントで囲まれます (デフォルトでは **div** エレメントです)。
- **element** – 入力フィールドを装飾するテンプレートを囲むエレメントです。 デフォルトではテンプレートは **div** エレメントで囲まれます。

#{invalid} と **#{required}** が **s:decorate** の内部で利用可能です。 装飾された入力コンポーネントが **required** に設定されると **#{required}** は **true** と評価されます。 また、 検証エラーが発生すると、**#{invalid}** は **true** と評価されます。

使用方法

```
<s:decorate template="edit.xhtml">
  <ui:define name="label">Country:</ui:define>
  <h:inputText value="#{location.country}" required="true"/>
</s:decorate>
```

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:s="http://jboss.com/products/seam/taglib">
  <div>
    <s:label styleClass="#{invalid?'error':''}">
```

```
      <ui:insert name="label"/>
      <s:span styleClass="required" rendered="#{required}">*</s:span>
    </s:label>
    <span class="#{invalid?'error':''}">
      <s:validateAll>
        <ui:insert/>
      </s:validateAll>
    </span>
    <s:message styleClass="error"/>
  </div>
</ui:composition>
```

31.1.3.2. <s:div>

説明

HTML ** ** をレンダリングします。

Attributes

なし

使用方法

```
<s:div rendered="#{selectedMember == null}">
  Sorry, but this member does not exist.
</s:div>
```

31.1.3.3. <s:span>

説明

HTML **** をレンダリングします。

Attributes

- **title** – span のタイトルです。

使用方法

```
<s:span styleClass="required" rendered="#{required}" title="Small
tooltip">
  *
</s:span>
```

31.1.3.4. <s:fragment>

説明

レンダリングされないコンポーネントで、その子のレンダリングを有効化 / 無効化するのに役立ちます。

Attributes

なし

使用方法

```
<s:fragment rendered="#{auction.highBidder ne null}">
  Current bid:
</s:fragment>
```

31.1.3.5. <s:label>

説明

JSF 入力フィールドをラベルで「装飾」します。 ラベルは HTML **<label>** タグの内側に配置され、 最も近い JSF 入力コンポーネントと関連付けられます。 **<s:decorate>** と併用されることがよくあります。

Attributes

- **style** – コントロールのスタイルです。
- **styleClass** – コントロールのスタイルクラスです。

使用方法

```
<s:label styleClass="label"> Country: </s:label>
<h:inputText value="#{location.country}" required="true"/>
```

31.1.3.6. <s:message>

説明

検証のエラーメッセージで JSF 入力フィールドを「装飾」します。

Attributes

なし

使用方法

```
<f:facet name="afterInvalidField">
  <s:span>
    Error:
    <s:message/>
  </s:span>
</f:facet>
```

31.1.4. Seam Text

31.1.4.1. <s:validateFormattedText>

説明

サブミットされた値が有効な Seam Text であることを確認します。

Attributes

なし

31.1.4.2. <s:formattedText>

説明

Seam Text を出力します。**Seam Text** はブログや **wiki**、 リッチテキストを使うその他のアプリケーションにとって便利なリッチテキストマークアップです。使用方法の詳細は **Seam Text** の章を参照してください。

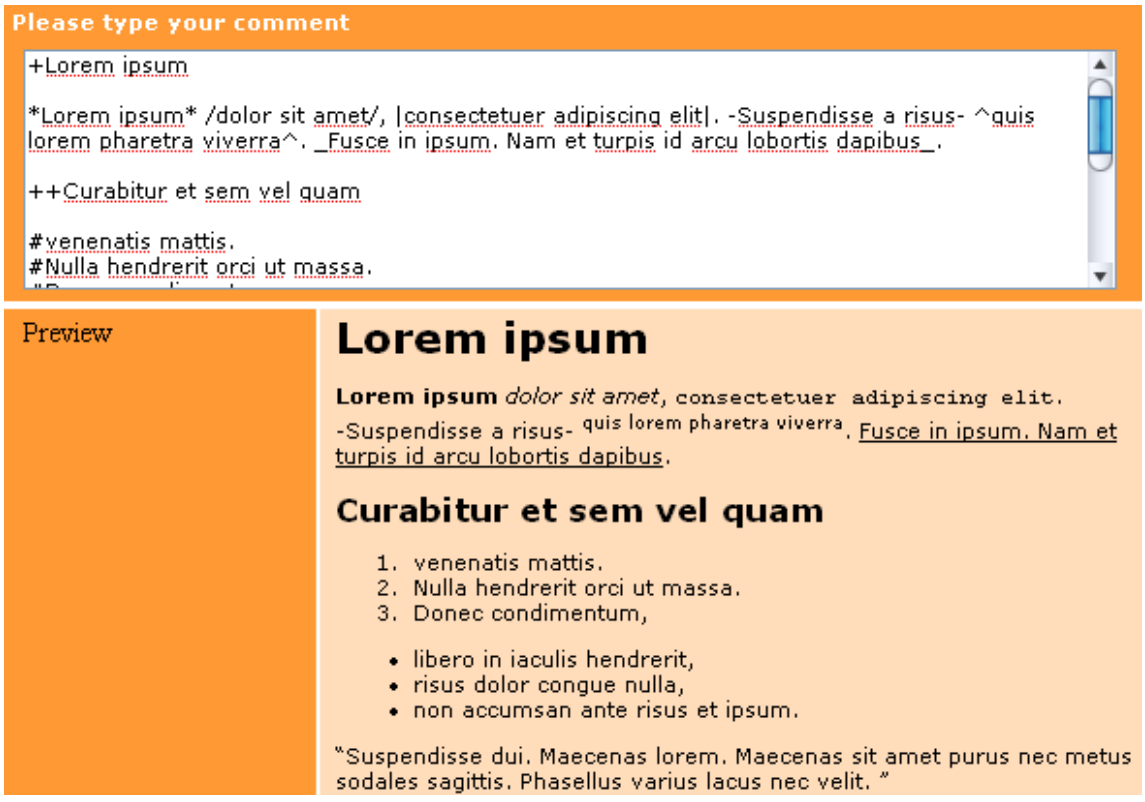
Attributes

- **value** – レンダリングするリッチテキストマークアップを指定する **EL** 式です。

使用方法

```
<s:formattedText value="#{blog.text}"/>
```

例



31.1.5. フォームのサポート

31.1.5.1. <s:token>

説明

クロスサイトリクエストフォージェリ **(XSRF)** 攻撃に対し **JSF** フォーム送信の安全性を確保するためランダムなトークンを生成して非表示のフォームフィールドに挿入します。 このコンポーネントを含むフォームを送信するためにブラウザのクッキーを有効にしておく必要があります。

Attributes

- **requireSession** – トークンをセッションにバインドさせるためセッション ID をフォーム署名に含ませるかどうかを示します。 デフォルト値は **false** ですが、 Facelets が「build before restore」モードの場合にしか使用されないはずです (「build before restore」は JSF 2.0 ではデフォルトのモードです)。

- **enableCookieNotice** – クッキーがブラウザで有効であることを確認するために JavaScript チェックがページに挿入されるべきであると示します。 クッキーが有効でないとフォームの送信は機能しないという通知をユーザーに表示します。 デフォルト値は **false** です。

- **allowMultiplePosts** – 同じフォームを同じ署名を使って複数送信が可能かどうかを示します (ビューは変更なし)。 フォームがそれ自体または UIToken コンポーネントをレンダリングしないで AJAX 呼び出しを行っている場合に必要とされることが多くあります。 UIToken コンポーネントが処理されるであろう AJAX の呼び出し時に UIToken コンポーネントを再度レンダリングした方がよいでしょう。 デフォルト値は **false** です。

使用方法

```
<h:form>
  <s:token enableCookieNotice="true" requireSession="false"/>
  ...
</h:form>
```

31.1.5.2. <s:enumItem>

説明

列挙値から **SelectItem** を作成します。

Attributes

- **enumValue** – 列挙値の文字列表現です。

- **label** – **SelectItem** をレンダリングするときに使用するラベルです。

使用方法

```
<h:selectOneRadio id="radioList"
                  layout="lineDirection"
 value="#{newPayment.paymentFrequency}">
   <s:convertEnum />
   <s:enumItem enumValue="ONCE"         label="Only Once" />
   <s:enumItem enumValue="EVERY_MINUTE" label="Every Minute" />
   <s:enumItem enumValue="HOURLY"       label="Every Hour" />
   <s:enumItem enumValue="DAILY"        label="Every Day" />
   <s:enumItem enumValue="WEEKLY"       label="Every Week" />
</h:selectOneRadio>
```

31.1.5.3. <s:selectItems>

説明

List、 Set、 DataModel または Array から **List<SelectItem>** を作成します。

Attributes

- **value** – **List<SelectItem>** に格納されるデータを指定する **EL** 式です。
- **var**– 反復中に現在のオブジェクトを保持するローカル変数の名前を定義します。
- **label** – **SelectItem** をレンダリングするときに使用するラベルです。**var** 変数の参照が可能です。
- **itemValue** – この選択肢を選ぶとサーバに戻される値を指定します。 これはオプション属性です。 含ませると **var** が使用されるデフォルトのオブジェクトになります。**var** 変数の参照が可能です。
- **disabled** – **true** の場合、 **SelectItem** が無効としてレンダリングされます。**var** 変数の参照が可能です。
- **noSelectionLabel** – 一覧の冒頭に記載する (オプションの) ラベルを指定します。 この値を選択し、 また **required="true"** も指定すると検証エラーとなります。
- **hideNoSelectionLabel** – true の場合、 値が選択されていると **noSelectionLabel** は非表示になります。

使用方法

```
<h:selectOneMenu value="#{person.age}" converter="ageConverter">
  <s:selectItems value="#{ages}" var="age" label="#{age}" />
</h:selectOneMenu>
```

31.1.5.4. <s:fileUpload>

説明

ファイルアップロードコントロールをレンダリングします。 このコントロールはエンコーディングタイプを **multipart/form-data** にしてフォーム内で使用しなければなりません。

```
<h:form enctype="multipart/form-data">
```

マルチパート要求の場合、 Seam Multipart サーブレットフィルタも **web.xml** 内で設定しなければなりません。

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

設定

components.xml では、 次のようなマルチパート要求用の設定オプションが設定できます。

- **createTempFiles** – **true** の場合、アップロードされたファイルはメモリではなく一時ファイルに保存されます。
- **maxRequestSize** – ファイルアップロード要求のバイト単位の最大サイズです。

例:

```
<component class="org.jboss.seam.web.MultipartFilter">
  <property name="createTempFiles">true</property>
  <property name="maxRequestSize">1000000</property>
</component>
```

Attributes

- **data** – バイナリのファイルデータを受け取る値バインディングを指定します。 受け取るフィールドは **byte[]** または **InputStream** として宣言されている必要があります。
- **contentType** – ファイルのコンテンツタイプを受け取る値バインディングを指定するオプションの属性です。
- **fileName** – ファイル名を受け取る値バインディングを指定するオプションの属性です。
- **fileSize** – ファイルサイズを受け取る値バインディングを指定するオプションの属性です。
- **accept** – 受け入れ可能なコンテンツタイプをコンマで区切った一覧です。 たとえば、**"images/png,images/jpg"**、 **"images/*"** などです。 記載されているタイプはブラウザによってサポートされない場合があります。
- **style** – コントロールのスタイルです。
- **styleClass** – コントロールのスタイルクラスです。

使用方法

```
<s:fileUpload id="picture"
              data="#{register.picture}" accept="image/png"
              contentType="#{register.pictureContentType}" />
```

31.1.6. その他

31.1.6.1. <s:cache>

説明

JBoss Cache を使用してレンダリングされたページフラグメントをキャッシュします。 **<s:cache>** は実際には組み込みの **pojoCache** コンポーネントで管理される JBoss Cache のインスタンスを使用するので注意してください。

Attributes

- **key** – レンダリングされたコンテンツをキャッシュするキーです。 値式がよく使用されます。例えば、 ドキュメントを表示するページフラグメントをキャッシュする場合、**key="Document-#{document.id}"**のように使います。

- **enabled** – キャッシュを使うべきかどうか決定する値式です。
- **region** – 使用する JBoss Cache のノードを指定します。 ノード毎に異なる有効期限ポリシーを持つことができます。

使用方法

```
<s:cache key="entry-#{blogEntry.id}" region="pageFragments">
  <div class="blogEntry">
    <h3>#{blogEntry.title}</h3>
    <div>
      <s:formattedText value="#{blogEntry.body}"/>
    </div>
    <p>
      [Posted on
      <h:outputText value="#{blogEntry.date}">
        <f:convertDateTime timezone="#{blog.timeZone}"
         locale="#{blog.locale}" type="both"/>
      </h:outputText>]
    </p>
  </div>
</s:cache>
```

31.1.6.2. <s:resource>

説明

ファイルダウンロードプロバイダとして動作するタグです。 JSF ページでは単独でなければなりません。 このコントロールを使用する場合は以下のように **web.xml** を設定する必要があります。

設定

```
<servlet>
  <servlet-name>Document Store Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.document.DocumentStoreServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Document Store Servlet</servlet-name>
  <url-pattern>/seam/docstore/*</url-pattern>
</servlet-mapping>
```

Attributes

- **data** – ダウンロードすべきデータを指定します。 java.util.File、 InputStream、 バイトアレイなどになります。
- **fileName** – 処理するファイルのファイル名です。
- **contentType** – ダウンロードするファイルのコンテンツタイプです。
- **disposition** – 使用するディスポジションです。 デフォルトのディスポジションは **inline** です。

使用方法

以下のようにタグを使用します。

```
<s:resource xmlns="http://www.w3.org/1999/xhtml"
   xmlns:s="http://jboss.com/products/seam/taglib"
   data="#{resources.data}" contentType="#{resources.contentType}"
   fileName="#{resources.fileName}" />
```

上記の **resources** という名前の Bean はバッキング Bean であり、 要求パラメータの場合、 特定のファイルを処理します。 **s:download** を参照してください。

31.1.6.3. <s:download>

説明

<s:resource> への RESTful リンクをビルドします。 ネストされた **f:param** はその URL を構成します。

- **src** – ファイルを処理するリソースファイルです。

Attributes

```
<s:download src="/resources.xhtml">
  <f:param name="fileId" value="#{someBean.downloadableFileId}"/>
</s:download>
```

http://localhost/resources.seam?fileId=1 と同じようなフォームのリンクを生成します。

31.1.6.4. <s:graphicImage>

説明

Seam コンポーネント内で画像を作成できるようにする拡張 **<h:graphicImage>** です。 さらに画像の変換も可能です。

<h:graphicImage> の属性はすべてサポートされる他、 以下もサポートされます。

Attributes

- **value** – 表示する画像を指定します。 パスを表す **String** (クラスパスからロードされます)、 **byte[]**、 **java.io.File**、 **java.io.InputStream**、 **java.net.URL** が指定可能です。 現在サポートされている画像フォーマットは **image/bmp**、**image/png**、 **image/jpeg** と **image/gif** です。

- **fileName** – 画像のファイル名を指定します。 この名前は一意にしてください。 指定しないと画像に一意のファイル名が生成されます。

変換

画像に変換を適用するには、 適用する変換を指定するタグをネストさせます。 Seam は現在、 次のような変換タグをサポートしています。

<s:transformImageSize>

- **width** – 画像の新しい幅を指定します。
- **height** – 画像の新しい高さを指定します。
- **maintainRatio** – **width** か **height** のいずれかを指定した状態で **true** にすると、 画像はリサイズされて **height**:**width** の縦横比を維持します。
- **factor** – 与えられた係数で画像を拡大縮小します。

<s:transformImageBlur>

- **radius** – 与えられた半径でコンボリューションブラーを実行します。

<s:transformImageType>

- **contentType** – 画像のタイプを **image/jpeg** または **image/png** に変更します。

独自の変換を作成することもできます。 **org.jboss.seam.ui.graphicImage.ImageTransform** を実装する **UIComponent** を作成します。 **applyTransform()** メソッド内で **image.getBufferedImage()** を使って元の画像を取得し、 **image.setBufferedImage()** で変換した画像を設定します。 変換はビューに指定された順序で適用されます。

使用方法

```
<s:graphicImage rendered="#{auction.image ne null}"
   value="#{auction.image.data}">
  <s:transformImageSize width="200" maintainRatio="true"/>
</s:graphicImage>
```

31.1.6.5. <s:remote>

説明

Seam Remoting を使う場合に必要な Javascript のスタブを生成します。

Attributes

- **include** – コンポーネント名 (または完全修飾クラス名)をコンマで区切った一覧です。 Seam Remoting の Javascript スタブを生成します。 詳しくは 24章 *リモーティング* を参照してください。

使用方法

```
<s:remote include="customerAction,accountAction,com.acme.MyBean"/>
```

31.2. アノテーション

Seam は Seam コンポーネントを JSF コンバータやバリデータとして使えるようにするアノテーションも提供します。

@Converter

```
@Name("itemConverter")
```

```
@BypassInterceptors
@Converter
public class ItemConverter implements Converter {
  @Transactional
  public Object getAsObject(FacesContext context, UIComponent cmp,
                            String value) {
    EntityManager entityManager =
      (EntityManager) Component.getInstance("entityManager");
    entityManager.joinTransaction(); // Do the conversion
  }
  public String getAsString(FacesContext context, UIComponent cmp,
                            Object value) {
    // Do the conversion
  }
}
```

```
<h:inputText value="#{shop.item}" converter="itemConverter" />
```

Seam コンポーネントを JSF コンバータとして登録します。 上記では、 値をそのオブジェクト表現に変換するときにコンバータが JTA トランザクション内の JPA EntityManager にアクセスします。

@Validator

```
@Name("itemValidator")
@BypassInterceptors
@org.jboss.seam.annotations.faces.Validator
public class ItemValidator implements javax.faces.validator.Validator {
  public void validate(FacesContext context, UIComponent cmp,
                       Object value) throws ValidatorException {
    ItemController ItemController =
      (ItemController) Component.getInstance("itemController");
    boolean valid = itemController.validate(value);
    if (!valid) {
      throw ValidatorException("Invalid value " + value);
    }
  }
}
```

```
<h:inputText value="#{shop.item}" validator="itemValidator" />
```

Seam コンポーネントを JSF バリデータとして登録します。 上記では、 バリデータは別の Seam コンポーネントをインジェクトし、 インジェクトされたコンポーネントが値の検証に使用されます。

第32章 JBOSS EL

Seam は JBoss EL を使用し、標準の統合式言語 (Unified Expression Language : Unified EL) を拡張します。これにより EL 式のパワーや表現力の機能拡張が実現します。

32.1. パラメータ化された式

標準の EL によりメソッドをユーザー定義のパラメータで使用することはできませんが、JBoss EL を使用するとこの制約を解消できます。例えば以下のとおりです。

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel)}"
                 value="Book Hotel"/>
```

```
@Name("hotelBooking")
public class HotelBooking {
  public String bookHotel(Hotel hotel) {
    // Book the hotel
  }
}
```

32.1.1. 使い方

Java からのメソッド呼び出しのように、パラメータは括弧で囲まれコンマで区切られます。

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel, user)}"
                 value="Book Hotel"/>
```

パラメータ **hotel** と **user** は値式として評価され、コンポーネントの**bookHotel()** メソッドに渡されます。

以下のように、パラメータにはいずれの値式も使用できます。

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel.id,
                           user.username)}"
                 value="Book Hotel"/>
```

ページがレンダリングされるとパラメータの名前 **hotel.id** および **user.username** が保存され、ページがサブミットされるときに値式として評価されます。 パラメータとしてオブジェクトを渡すことはできません。

ページがレンダリングされる場合だけでなくサブミットされる場合にもパラメータは使用可能である必要があります。ページがサブミットされるときに引数が解決できないと、アクションメソッドは **null** 引数を付けて呼び出されます。

一重引用符を用いてリテラル文字列を渡すこともできます。

```
<h:commandLink action="#{printer.println('Hello world!')}"
               value="Hello"/>
```

Unified EL は値式にも対応し、バッキング Bean にフィールドをバインドするために使用されます。値式は JavaBean の命名規則を使用し getter と setter の組み合わせが必要です。 JSF は値の取得 (get) のみが必要な場合にも値式を必要とすることがよくあります (**rendered** 属性など)。しかし、多くのオ

ブジェクトは適切な名前が付いたプロパティアクセサを持たず、パラメータを必要としません。

JBoss EL は、メソッド構文を使って値が取得されるようにすることでこの制約を取り除きます。例えば以下のとおりです。

```
<h:outputText value="#{person.name}"
              rendered="#{person.name.length() > 5}" />
```

同様にして 1 つのコレクションのサイズにアクセスが可能です。

```
#{searchResults.size()}
```

一般的には **#{obj.property}** 形式の式は **#{obj.getProperty()}** 式と同一となります。

パラメータを使うこともできます。 次の例ではリテラル文字列の引数を持つ **productsByColorMethod** を呼び出しています。

```
#{controller.productsByColor('blue')}
```

32.1.2. 制約とヒント

JBoss EL には制約がいくつかあります。

- **JSP 2.1 との非互換性** – JBoss EL は現在 JSP 2.1 との併用はできません。 コンパイラがパラメータ付きの式を拒否するためです。 JSF 1.2 でこの拡張機能を使用したい場合は Facelets を使用する必要があります。 この拡張機能は JSP 2.0 では正常に動作します。
- **反復コンポーネント内での使用** – **<c:forEach />** や **<ui:repeat />** といったコンポーネントはリストまたはアレイに対して反復を行い、 一覧内の各アイテムをネストされるコンポーネントに公開します。これは **<h:commandButton />** または **<h:commandLink />** を使った行を選択している場合に効果的です。

  ```
  @Factory("items")
  public List<Item> getItems() {
    return entityManager.createQuery("select ...").getResultList();
  }
  ```

  ```
  <h:dataTable value="#{items}" var="item">
    <h:column>
      <h:commandLink value="Select #{item.name}"
         action="#{itemSelector.select(item})" />
    </h:column>
  </h:dataTable>
  ```

  ただし、 **<s:link />** や **<s:button />** を使用したい場合はアイテムを **DataModel** として公開して **<dataTable />** (または **<rich:dataTable />** のようなコンポーネントセットと同等) を使用しなければなりません。 **<s:link />** または **<s:button />** のいずれもフォームをサブミットしないためブックマーク可能なリンクを生成しません。 アクションメソッドが呼び出された場合にそのアイテムを再度作成するためには追加のパラメータが必要です。 **DataModel** で支えられるデータテーブルが使用される場合のみこのパラメータは追加可能です。
- **Java コードから MethodExpression を呼び出す** – 通常、 **MethodExpression** が作成され

るとパラメータタイプが JSF によって渡されます。ただし、メソッドバインディングでは JSF は渡すパラメータがないとみなします。この拡張機能では、 式の評価が終了するまでパラメータタイプを知ることはできません。これにより以下の 2 つの結果が生じます。

- Java コードで **MethodExpression** を呼び出すとき、渡すパラメータが無視される場合があります。 式で定義されたパラメータが優先されます。
- 通常、**methodExpression.getMethodInfo().getParamTypes()** はいつでも安全に呼び出すことができます。パラメータを持つ式に関しては、まず **MethodExpression** を呼び出してから、 **getParamTypes()** を呼び出すようにしてください。

上記のようなケースは非常に稀であり、 Java コードで **MethodExpression** を手作業で呼び出したい場合にのみ適用されます。

32.2. プロジェクション

JBoss EL は限られたプロジェクション構文に対応します。 プロジェクション式はサブ式を複数値 (リスト、セットなど) の式全体にマッピングします。 以下が例です。

```
#{company.departments}
```

この式は部署の一覧を返します。 部署名の一覧のみが必要な場合はその値を取得するために一覧全体を反復する必要があります。 JBoss EL ではプロジェクション式を使って実行できます。

```
#{company.departments.{d|d.name}}
```

サブ式は中括弧で囲みます。 この例では各部署ごとに **d.name** 式が評価され、 **d** を部署のオブジェクトへのエイリアスとして使用します。 この式の結果は文字列値の一覧となります。

あらゆる長さの会社の部署名を使用すると仮定した場合、式にはどの有効な式も使用できるため、次の記述も有効となります。

```
#{company.departments.{d|d.size()}}
```

プロジェクションはネストさせることが可能です。 次の式は各部署内のそれぞれの従業員の姓を返します。

```
#{company.departments.{d|d.employees.{emp|emp.lastName}}}
```

ただしプロジェクションのネストは若干の注意が必要です。 次の式は全部署の全従業員一覧を返すように見えます。

```
#{company.departments.{d|d.employees}}
```

しかし実際には、各部署ごとの従業員一覧を含む一覧を返します。 値を結合させるにはもう少し長い式を使う必要があります。

```
#{company.departments.{d|d.employees.{e|e}}}
```

この構文は Facelets や JSP では解析不能なため、XHTML または JSP ファイルでは使用できません。 JBoss EL の今後のバージョンではプロジェクション構文により容易に対応できるようになる予定です。

第33章 クラスター化と EJB 非活性化

Web クラスタ化と EJB 非活性化は Seam では 1 つの共通のソリューションを共有するため一緒に説明していくことにします。 本章ではプログラミングモデルおよびクラスタ化と EJB 非活性化を使うことによってそれが受ける影響について見ていきます。 Seam によるコンポーネントおよびエンティティのインスタンスの非活性化、非活性化とクラスタ化との関連、 非活性化を有効にする方法について記載しています。 また、 Seam アプリケーションをクラスタにデプロイして HTTP セッションの複製が正しく動作していることを確認する方法についても記載しています。

まず、 クラスタ化に関する基本的な情報から見ていくことにします。次に JBoss AS クラスタへの Seam アプリケーションのデプロイ例を示します。

33.1. クラスタ化

Web クラスタリングとしても知られるクラスタ化では、 クライアントにはアプリケーションのユニフォームビューを表示しながらこのアプリケーションを 2 台以上の複数の並列サーバー **(ノード)** で実行させることができます。 1 台または複数のサーバーに障害が発生した場合、 アプリケーションは生き残っているノードからアクセスすることができるよう負荷をサーバーに分散します。 ノードを追加するだけでパフォーマンスや可用性が高まるためスケーラブルなエンタープライズアプリケーションには極めて重要となります。 ただし、 障害を起こしたサーバーに保持された状態はどうなるのかという疑問が起こってきます。

今までのところ、 Seam は追加のスコープを含ませステートフルの (スコープされた) コンポーネントのライフサイクルを統制することで状態管理を行うことは理解されていることと思います。 しかし Seam の状態管理はインスタンスの作成、 保存、 破棄のみではありません。 Seam では JavaBean コンポーネントに対する変更の追跡や要求時に戦略的なポイントでその変更を保存するため、 要求がクラスタ内の 2 次ノードに移行した場合に変更を復元することができます。 ステートフル EJB コンポーネントの監視や複製は EJB サーバーによって処理されます。 Seam 状態管理ではステートフル JavaBean に同様の機能を提供します。

JavaBean コンポーネントの監視に加え、 Seam では管理エンティティインスタンス (つまり、 JPA および Hiernate のエンティティ) が複製時に付随しないようにします。 ロードすべきエンティティの記録を保持し、 2 次ノードで自動的にそれらをロードします。 **この機能を利用するには Seam 管理永続コンテキストを使用する必要があります。** 詳細は本章の後半で説明します。

次にクラスタ化用にプログラムを行う方法について見ていくことにします。

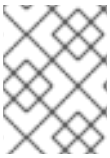
33.1.1. クラスタ化用のプログラミング

クラスタ環境で使用するセッションまたは対話スコープの可変の JavaBean コンポーネントは、すべて Seam API より **`org.jboss.seam.core.Mutable`** インターフェースを実装しなければなりません。 コントラクトの一部としてコンポーネントは **`dirtyflag`** イベントを管理する必要があり、 これが保存しなければならないフォームにユーザーが変更を加えたかどうかを示します。 このイベントは **`clearDirty()`** メソッドにより報告およびリセットが行われ、 このメソッドはコンポーネントの複製が必要かどうかを確定するため呼び出されます。 これによりオブジェクトに対するあらゆる変更に対しセッション属性の追加や削除に Servlet API を使用する必要がなくなります。

また、 すべてのセッションおよび対話スコープの JavaBean コンポーネントが **シリアライズ可能**であることを必ず確認してください。 **transient** のマークが付けられているか **`@PrePassivate`** メソッドで null に設定されていない限り、 ステートフルコンポーネントのフィールドはすべてシリアライズ可能でなければなりません (EJB または JavaBean)。 transient または nullified のフィールドの値は **`@PostActivate`** メソッドで復元することができます。

リスト作成に **`List.subList`** を使用する場合、 問題が発生する可能性があります。 作成されたリストがシリアライズできないためです。 オブジェクトを自動的に作成するいくつかのメソッドにも同様の

状況が発生することがあります。 **java.io.NotSerializableException** が発生した場合は、 この例外にブレークポイントを配置しデバッグモードでアプリケーションサーバーを稼働して問題となるメソッドを探します。



注記

クラスタ化はホットデプロイしたコンポーネントとは動作しません。 また、 開発環境以外ではホットデプロイが可能なコンポーネントの使用は避けてください。

33.1.2. Seam アプリケーションをセッション複製で JBoss AS クラスタへデプロイ

次の手順は **seam-gen** アプリケーションおよび Seam Booking サンプルに対して検証が行われました。

本項ではマスターとスレーブのサーバーの IP アドレスはそれぞれ **192.168.1.2** と **192.168.1.3** であると仮定します。 両方のノードが要求に応答しセッションと通信していることを確認しやすいよう意図的に **mod_jk** ロードバランサーは使用しませんでした。

次のログメッセージは WAR アプリケーション **vehicles.war** のデプロイメントおよびそれに該当するデータソース **vehiclesDatasource** から生成されました。 Booking サンプルはこのプロセスに完全に対応しています。クラスタへのデプロイ方法に関する説明は **examples/booking/readme.txt** ファイルでご覧頂けます。

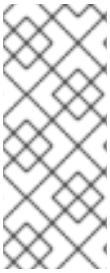
以降の手順はファームデプロイメントの方法を用いていますが、通常通りにアプリケーションをデプロイしてから起動順序に応じて 2 つのサーバー同士でマスターとスレーブの関係を決めさせることもできます。

チュートリアル内のタイムスタンプは混同を避けるためすべてゼロに置換しています。

注記

使用するノードが Red Hat Enterprise Linux または Fedora を稼働する別々のマシンに存在する場合、互いを自動的に認識し合わないことがあります。 JBoss AS のクラスタ化は jGroups によって提供される UDP (ユーザーデータグラムプロトコル) マルチキャスティングに依存します。 Red Hat Enterprise Linux および Fedora に同梱される SELinux はデフォルトでこれらのパケットをブロックします。 パケットを許可するには **iptables** のルールを変更してください (root 権限)。 IP アドレスが **192.168.1.x** とした場合、 コマンドは次のようになります。

```
  /sbin/iptables -I RH-Firewall-1-INPUT 5 -p udp -d 224.0.0.0/4
-j ACCEPT
  /sbin/iptables -I RH-Firewall-1-INPUT 9 -p udp -s
192.168.1.0/24 -j ACCEPT
  /sbin/iptables -I RH-Firewall-1-INPUT 10 -p tcp -s
192.168.1.0/24 -j ACCEPT
  /etc/init.d/iptables save
```



注記

ステートフルセッション Bean と HTTP Session 複製でアプリケーションを JBoss AS クラスタにデプロイする場合、 ステートフルセッション Bean のクラスに **@Clustered** (JBoss EJB 3.0 アノテーション API より) アノテーションを付与するか **jboss.xml** 記述子で **clustered** の印を付ける必要があります。 詳細については Booking サンプルを参照してください。

33.1.3. チュートリアル

1. JBoss AS のインスタンスを 2 つ作成します (**zip** を 2 度展開するだけです)。

   HSQLDB を使用しない場合は JDBC ドライバを両方のインスタンスの **server/all/lib/** にデプロイします。

2. **WEB-INF/web.xml** に最初の子エレメントとして **<distributable/>** を追加します。

3. **org.jboss.seam.core.init** にある **distributable** プロパティを **true** に設定して **ManagedEntityInterceptor** を有効にします (つまり、 **WEB-INF/components.xml** に **<core:init distributable="true">** と設定します)。

4. 使用可能な IP アドレスが 2 つあることを確認します (同じインターフェースに 2 台のコンピュータ、 2 枚のネットワークカード、 または 2 つの IP アドレスがバインドされているということです)。2 つの IP アドレスは **192.168.1.2** と **192.168.1.3** と仮定します。

5. 最初の IP でマスターとなる JBoss AS のインスタンスを起動します。

   ```
   ./bin/run.sh -c all -b 192.168.1.2
   ```

   ログはクラスタメンバーが 1 つ、 他のメンバーはゼロであると報告するはずです。

6. スレーブとなる JBoss AS インスタンスの **server/all/farm** ディレクトリは空であることを確認します。

7. スレーブの JBoss AS インスタンスを 2 番目の IP で起動します。

   ```
   ./bin/run.sh -c all -b 192.168.1.3
   ```

   ログはクラスタメンバーが 2 つ、 他のメンバーが 1 つであると報告するはずです。 また、 マスターインスタンスから取得されている状態を表示します。

8. **-ds.xml** をマスターインスタンスの**server/all/farm** にデプロイします。

   このデプロイメントの確認がマスターインスタンスのログに表示されるはずです。 スレーブインスタンスに対するデプロイメント確認の該当メッセージが表示されるはずです。

9. アプリケーションを server/all/farm ディレクトリにデプロイします。 通常のアプリケーション起動メッセージが終了すると、マスターインスタンスのログにデプロイメントの確認が表示されるはずです。 スレーブインスタンスのログはデプロイメントを確認する該当メッセージを表示します (デプロイされたアーカイブが転送されるのに最長 3 分間待つ必要がある場合があります)。

これでアプリケーションは HTTP Session 複製によりクラスタ内で実行しています。次のステップはクラスタ化が正しく動作していることを確認する作業です。

33.1.4. JBoss AS クラスタで稼働しているアプリケーションの配信可能なサービスの検証

これでアプリケーションが 2 つの異なる JBoss AS サーバーで正しく起動するようになりました。 しかし、2 つのインスタンスが正しく HTTP Session を交換していることを確認することが重要です。 これによりマスターインスタンスが停止した場合にもアプリケーションはスレーブインスタンスで動作を続行できます。

まず、 マスターインスタンスでアプリケーションに行き最初の HTTP Session を開始します。 同じインスタンスで JBoss AS JMX Console を開き次の管理 Bean に移動します。

- **Category:** jboss.cache
- **Entry:** config=standard-session-cache,service=Cache
- **Method:** printDetails()

printDetails() メソッドを呼び出します。 これによりアクティブな HTTP Session のツリーが表示されます。 ブラウザで使用しているセッションがツリーにあるセッションのいずれかに該当していることを確認します。

次にスレーブインスタンスに切り替え、JMX Console で同じメソッドを呼び出します。 このアプリケーションのコンテキストパスにまったく同一のツリーが表示されるはずです。

両方のツリーがまったく同一であれば両方のサーバーのセッションは同一であること判断できます。 次に、 データが正しくシリアライズおよびデシリアライズを行うかをテストする必要があります。

マスターインスタンスの URL よりサインインします。 次にサーブレットパスのすぐ後ろに **;jsessionid=XXXX** を置き IP アドレスを変更して 2 番目のインスタンスの URL を構成します (セッションが別のインスタンスに引き継がれたのがわかるはずです)。

ここでマスターインスタンスを終了して、スレーブインスタンスからアプリケーションを継続して使用できるか確認してみます。 そのあと、 **server/all/farm** ディレクトリからデプロイされたアプリケーションを削除してインスタンスを再起動します。

URL の IP をマスターインスタンスの IP に戻し、 新しい URL に移動します。 オリジナルのセッション ID がまだ使用されているのがわかるはずです。

セッションまたは対話スコープの Seam コンポーネントを作成して適切なライフサイクルメソッドを実装することで、オブジェクトの非活性化と活性化を監視することができます。**HttpSessionActivationListener** インターフェースのメソッドが使用できます (EJB 以外の全コンポーネントで自動的に登録されます)。

```
public void sessionWillPassivate(HttpSessionEvent e);
public void sessionDidActivate(HttpSessionEvent e);
```

代わりに、2 つの public void のメソッドに **@PrePassivate** と **@PostActivate** のマークをそれぞれ付けることもできます。 不活性化は各要求の最後に発生しますが、活性化はノードが呼び出されたときに発生するのを忘れないようにしてください。

複製を透過的にするために、 Seam は変更されたオブジェクトを自動的に追跡します。 このため、必要となるのはセッションまたは対話スコープのコンポーネントにある **dirtyflag** の管理のみで、 あとは Seam が JPA エンティティのインスタンスを処理してくれます。

33.2. EJB 非活性化と MANAGEDENTITYINTERCEPTOR

ManagedEntityInterceptor (MEI) は Seam ではオプションのインターセプタです。 有効にすると対話スコープのコンポーネントに適用されます。 MEI を有効にするには **org.jboss.seam.init.core** コンポーネントで **distributable** を **true** に設定します。 また、次のコンポーネント宣言を **components.xml** ファイルに追加または更新することができます。

```
<core:init distributable="true"/>
```

これは HTTP Session の複製を有効にしませんが、Seam による EJB コンポーネントまたは HTTP Session 内のコンポーネントいずれかの非活性化の処理を可能にします。

MEI によって、拡張永続コンテキストを 1 つ以上持つ対話の全寿命に渡り、 永続コンテキストによってロードされたエンティティインスタンスは全て管理されているようにします。つまり、 これらのエンティティインスタンスは非活性化イベントによって時期尚早に切り離されることはありません。 これにより拡張永続コンテキストの整合性を維持するため、その整合性が保証されます。

整合性が脅かされる状況として、 拡張永続コンテキストをホストするステートフルセッション Bean の不活性化と HTTP Session の非活性化があります。

33.2.1. 非活性化と永続性の衝突

永続コンテキスト は永続マネージャがデータベースからロードしたエンティティインスタンス (オブジェクト) の保存に使用されます。 永続コンテキスト内には一意のデータベースの記録に対し 1 つのオブジェクトしか存在しません。 記録が永続コンテキストにロードされる場合、 アプリケーションはそのデータベースに対する呼び出しを回避することができるため、よく **1 次レベルキャッシュ** と呼ばれます。

永続コンテキスト内のオブジェクトは変更可能であり、 変更後は **ダーティ** とみなされます。 変更は永続マネージャによって追跡され、 必要に応じてデータベースに移行されます。 このため永続コンテキストはデータベースに対して保留となっている変更を管理します。

データベース指向のアプリケーションは直ちにデータベースに転送しなければならないトランザクション情報をキャプチャします。 この情報は必ずしも 1 画面でキャプチャできるとは限らず、 ユーザーは保留にしている変更に応じるか拒否するかの選択をしなければならない場合があります。

こうしたトランザクション側のことはユーザーにとっては必ずしもわかりやすい必要はありませんでした。拡張永続コンテキストによりトランザクションに関するユーザーの理解は広がります。アプリケーションが必要とする限り変更を保持してから、組み込み永続マネージャ機能で保留している変更をデータベースにプッシュすることができます (`EntityManager#flush()`)。

永続マネージャは **オブジェクト参照** でエンティティインスタンスにリンクされます。 エンティティインスタンスはシリアライズ可能ですが永続マネージャはシリアライズできません。 シリアライズはステートフルセッション Bean または HTTP Session のいずれかが非活性化される場合に可能となります。 アプリケーションの動作を継続させるためには、 永続マネージャとそのエンティティインスタンス間の関係を管理する必要があります。 MEI によってこれに対応します。

33.2.2. ケース 1: EJB 非活性化の存続

当初、 対話はステートフルセッション Bean 用に設計されていました。 EJB3 仕様がステートフルセッション Bean を拡張永続コンテキストのホストとして定義するためです。 Seam は **Seam 管理の永続コンテキスト** を導入し、この仕様の制約のいくつかに対処します。いずれのコンテキストもステートフルセッション Bean で使用可能です。

ステートフルセッション Bean をアクティブにしておくためには、 クライアントにステートフルセッション Bean への参照を持たせておく必要があります。 Seam の対話コンテキストはこの参照用に最適の場所です。 つまり、 ステートフルセッション Bean は対話コンテキストの間はアクティブのままとなります。 さらに、 `@PersistenceContext(EXTENDED)` アノテーションを付与されたステートフルセッション Bean にインジェクトされる `EntityManager` は、ステートフルセッション Bean にバインドされ Bean の寿命を通してアクティブであり続けます。 `@In` アノテーションでインジェクトされる `EntityManager` は Seam によって管理され対話コンテキストに直接保存されるため、**対話** の間はアクティブであり続け、 ステートフルセッション Bean とは無関係になります。

また、 Java EE コンテナもステートフルセッション Bean の非活性化を行えますがこの方法は問題となる場合があります。この処理をコンテナに任せず、 ステートフルセッション Bean のそれぞれの呼び出

し後に Seam にそのエンティティインスタンスの参照をステートフルセッション Bean から現在の対話にそして HTTP Session に転送させます。 これによりステートフルセッション Bean の関連フィールドを無効化にします。次に Seam はこれらの参照を後続する呼び出しの冒頭で復元します。 Seam はすでに永続マネージャを対話に保存しているため、 ステートフルセッション Bean の非活性化や活性化はアプリケーションに悪影響は与えません。

重要

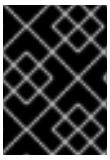
アプリケーションが拡張永続コンテキストへの参照を持つステートフルセッション Bean を使用し、 その Bean の非活性化が可能である場合は、 単一インスタンスまたはクラスタいずれを使用するかに関係なく MEI を **使用する必要があります**。

ステートフルセッション Bean で非活性化を無効にすることができます。 JBoss Wiki にある Ejb3DisableSfsbPassivation ページで詳細をご覧ください。

33.2.3. ケース 2: HTTP セッション複製の存続

ステートフルセッション Bean を非活性化するときに HTTP Session が使用されますが、セッションの複製を有効にしたクラスタ環境では HTTP Session も非活性化にできます。 永続マネージャはシリアライズ化できないため、HTTP Session の非活性化では通常永続マネージャを破棄することになります。

エンティティインスタンスが対話に初めて配置されるときに Seam はそのインスタンスをラッパーに埋め込みます。これにはシリアライズ後に永続マネージャとインスタンスを再度関連付けを行うための情報が含まれています。 アプリケーションがノードを変更すると (サーバーに障害が発生した場合など)、Seam の MEI は永続コンテキストを再構築します。 永続コンテキストはデータベースから再構築されるため、 インスタンスに対して保留となっていた変更は失われます。 ただし、 Seam の楽観的ロックにより、 ファイルが変更している場合に 1 インスタンスの複数のバージョンが発生すると最新の変更のみが受け付けられるようになっています。



重要

アプリケーションが HTTP Session 複製でクラスタにデプロイされる場合は **MEI を使用しなければなりません**。

第34章 パフォーマンス調整

本章では Seam アプリケーションで最適なパフォーマンスを得るためのヒントを説明しています。

34.1. インターセプタの迂回

repetitive valueJavaServer Faces (JSF) データテーブルにある反復値バインディングや **ui:repeat** などの反復コントロールの場合、 参照される Seam コンポーネントの呼び出しが行われるたびに完全なインターセプタスタックが呼び出されます。特にコンポーネントが何度もアクセスされると、これにより大幅にパフォーマンスが低下することがあります。 呼び出される Seam コンポーネントのインターセプタスタックを無効にすることでパフォーマンスを向上させることができます。 コンポーネントクラスに **@BypassInterceptors** アノテーションを付与します。

警告

インターセプタを無効にする前に、 **@BypassInterceptors** の付いたコンポーネントはすべてバイジェクション、 アノテーション付きセキュリティ制限、 同期などの機能は使用できない点に注意してください。 ただし、 通常は **@In** でコンポーネントをインジェクトする代わりに **Component.getInstance()** を使用するなどこれらの機能を補う手段があります。

次のコード一覧はその無効化されたインターセプタを持つ Seam コンポーネントを表しています。

```
@Name("foo")
@Scope(EVENT)
@BypassInterceptors
public class Foo {
  public String getRowActions() {
    // Role-based security check performed inline instead of using
    // @Restrict or other security annotation
    Identity.instance().checkRole("user");

    // Inline code to lookup component instead of using @In
    Bar bar = (Bar) Component.getInstance("bar");

    String actions;
    // some code here that does something
    return actions;
  }
}
```

第35章 SEAM アプリケーションのテスト

Seam アプリケーションのほとんどは少なくとも 2 種類の自動テストが必要です。 個々の Seam コンポーネントを独立してテストする **ユニットテスト** と、 アプリケーションのすべての Java 層 (ページの表示を除きすべて) を実行するスクリプト化された **統合テスト** です。

どちらのテストも簡単に作成できます。

35.1. SEAM コンポーネントのユニットテスト

Seam コンポーネントはすべて POJO (純粋な旧式 Java オブジェクト) であり、 ユニットテストを簡略化します。 また Seam はコンポーネント間でのやり取りやコンテキスト依存オブジェクトへのアクセスにバイジェクションを多用しているので、 通常のランタイム環境でなくてもとても簡単に Seam コンポーネントをテストすることができます。

次のような顧客アカウントのステートメントを作成する Seam コンポーネントを考えてみましょう。

```
@Stateless
@Scope(EVENT)
@Name("statementOfAccount")
public class StatementOfAccount {

  @In(create=true) EntityManager entityManager

  private double statementTotal;

  @In
  private Customer customer;

  @Create
  public void create() {
    List<Invoice> invoices = entityManager
      .createQuery("select invoice from Invoice invoice where " +
                   "invoice.customer = :customer")
      .setParameter("customer", customer)
      .getResultList();
    statementTotal = calculateTotal(invoices);
  }

  public double calculateTotal(List<Invoice> invoices) {
    double total = 0.0;
      for (Invoice invoice: invoices) {
        double += invoice.getTotal();
      }
      return total;
     }
   // getter and setter for statementTotal
}
```

以下のように、 コンポーネントのビジネスロジックをテストする **calculateTotal** メソッドをテストすることができます。

```
public class StatementOfAccountTest {
  @Test
  public testCalculateTotal {
```

```
    List<Invoice> invoices =
      generateTestInvoices(); // A test data generator
    double statementTotal =
      new StatementOfAccount().calculateTotal(invoices);
    assert statementTotal = 123.45;
  }
}
```

データの取り出しや永続化、 また Seam が提供する機能をテストしているわけではない点に注意してください。 ここでは POJO のロジックのみをテストしてます。 Seam コンポーネントは通常はコンテナのインフラストラクチャには直接依存しないため、 ほとんどのユニットテストは簡単です。

アプリケーション全体をテストする場合は、 この後の項をお読みください。

35.2. SEAM コンポーネントの統合テスト

統合テストはもう少し複雑です。 コンテナのインフラストラクチャは除外することはできませんが、 自動テストを実行するためにわざわざアプリケーションサーバーへアプリケーションをデプロイしたいとも思わないでしょう。 そこで、 最低限必要なコンテナのインフラストラクチャをテスト環境に再現し、 性能を大きく損なうことなくアプリケーション全体を実行可能にする必要があります。

Seam により JBoss Enbedded のコンテナを使ってコンポーネントをテストすることができます。 詳細は「設定」の章を参照してください。 最小限のコンテナ内でアプリケーションを完全に実行するテストの記述が可能です。

```
public class RegisterTest extends SeamTest {

  @Test
  public void testRegisterComponent() throws Exception {

    new ComponentTest() {

      protected void testComponents() throws Exception {
        setValue("#{user.username}", "1ovthafew");
        setValue("#{user.name}", "Gavin King");
        setValue("#{user.password}", "secret");
        assert invokeMethod("#{register.register}").equals("success");
        assert getValue("#{user.username}").equals("1ovthafew");
        assert getValue("#{user.name}").equals("Gavin King");
        assert getValue("#{user.password}").equals("secret");
      }

    }.run();

  }

  ...

}
```

35.2.1. モックを使用した統合テスト

統合テスト環境では使用できないリソースを必要とする Seam コンポーネントの置き換えが必要な場合があります。 たとえば、 支払処理システムのファサードに次の Seam コンポーネントを使用するとします。

```
@Name("paymentProcessor")
public class PaymentProcessor {
  public boolean processPayment(Payment payment) { .... }
}
```

統合テストの場合、 次のようなモックコンポーネントを作成します。

```
@Name("paymentProcessor")
@Install(precedence=MOCK)
public class MockPaymentProcessor extends PaymentProcessor {
  public boolean processPayment(Payment payment) {
    return true;
  }
}
```

MOCK の優先度はアプリケーションコンポーネントのデフォルト優先度より高くなるため、 モック実装がクラスパスにあるときは必ず Seam はそれをインストールします。 実稼働環境にデプロイする場合はモック実装は存在しないので、 実際のコンポーネントがインストールされます。

35.3. SEAM アプリケーションのユーザーインタラクション統合テスト

さらに難しいことはユーザーインタラクションを模倣し、 適切にアサーションを配置することです。 テストフレームワークの中には Web ブラウザでユーザーのインタラクションを再生することでアプリケーション全体をテストできるものがあります。このようなフレームワークは便利ですが、 開発段階での使用には適していません。

SeamTest を使用して擬似 JSF 環境で **スクリプト化した**テストを記述することができます。 スクリプト化したテストは、 ビューと Seam コンポーネント間のやりとりを再現するため、 テスト中は JSF 実装の役割を演じることになります。 この方法ならビュー以外のあらゆるものすべてがテスト可能です。

上記でユニットテストしたコンポーネントの JSP ビューを考えてみましょう。

```
<html>
  <head>
    <title>Register New User</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <table border="0">
          <tr>
            <td>Username</td>
            <td><h:inputText value="#{user.username}"/></td>
          </tr>
          <tr>
            <td>Real Name</td>
            <td><h:inputText value="#{user.name}"/></td>
          </tr>
          <tr>
            <td>Password</td>
            <td><h:inputSecret value="#{user.password}"/></td>
          </tr>
        </table>
        <h:messages/>
```

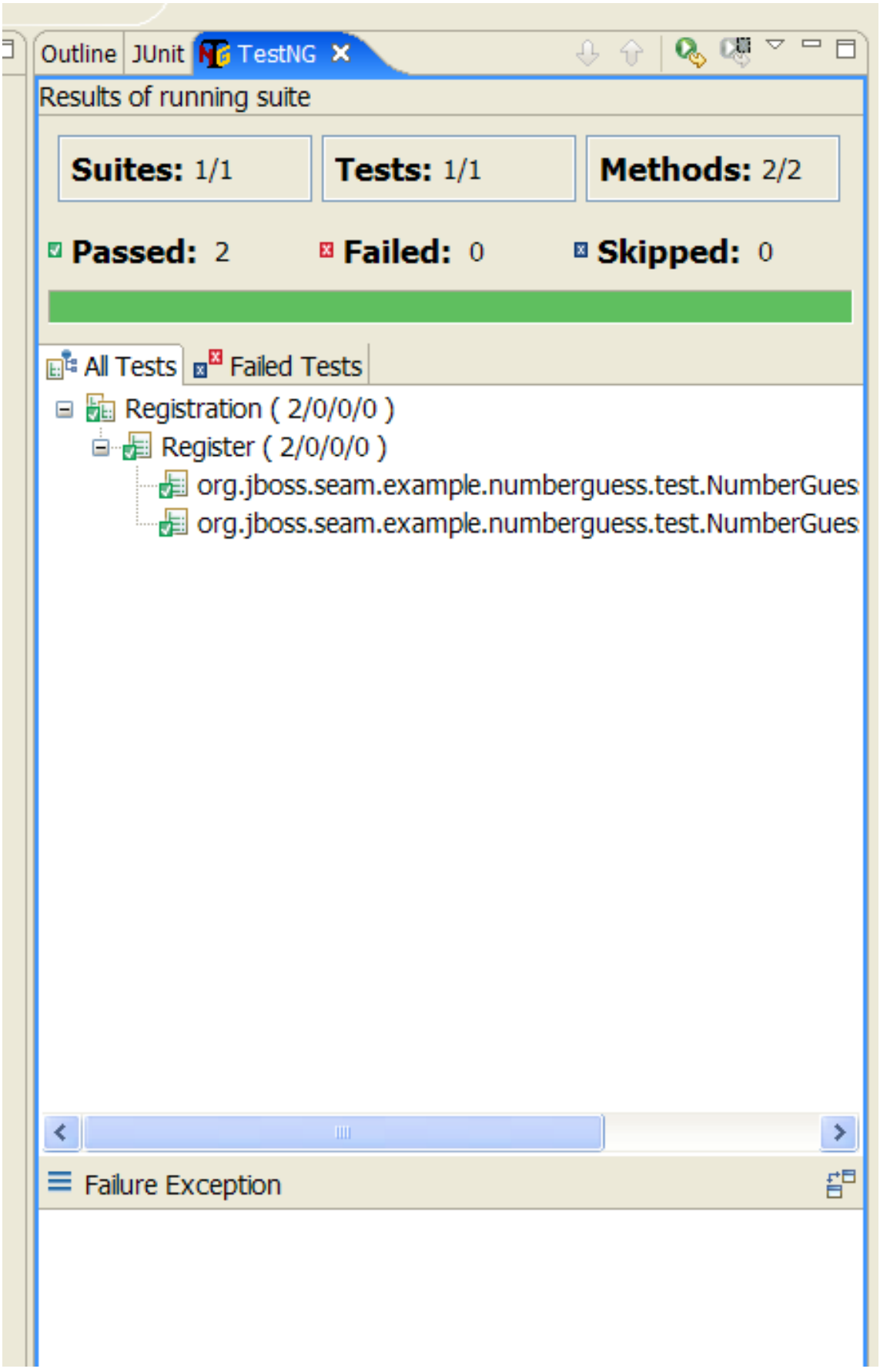
```
        <h:commandButton type="submit" value="Register"
                         action="#{register.register}"/>
      </h:form>
    </f:view>
  </body>
</html>
```

このアプリケーションの登録機能 (ユーザーが Register ボタンをクリックしたときの動作) をテストしたいとします。 TestNG 自動テストで JSF 要求のライフサイクルを再現してみます。

```
public class RegisterTest extends SeamTesFt {

  @Test
  public void testRegister() throws Exception {

    new FacesRequest() {

      @Override
      protected void processValidations() throws Exception {
        validateValue("#{user.username}", "1ovthafew");
        validateValue("#{user.name}", "Gavin King");
        validateValue("#{user.password}", "secret");
        assert !isValidationFailure();
      }

      @Override
      protected void updateModelValues() throws Exception {
        setValue("#{user.username}", "1ovthafew");
        setValue("#{user.name}", "Gavin King");
        setValue("#{user.password}", "secret");
      }

      @Override
      protected void invokeApplication() {
        assert invokeMethod("#{register.register}").equals("success");
      }

      @Override
      protected void renderResponse() {
        assert getValue("#{user.username}").equals("1ovthafew");
        assert getValue("#{user.name}").equals("Gavin King");
        assert getValue("#{user.password}").equals("secret");
      }
    }.run();
  }
  ...
}
```

ここで **SeamTest** を拡張し、コンポーネントに Seam 環境を提供し、テストスクリプトは **SeamTest.FacesRequest** を拡張する匿名クラスとして記述されています。 これにより模倣された JSF 要求ライフサイクルを提供しています (GET 要求をテストする **SeamTest.NonFacesRequest** もあります)。コードには各種の JSF フェーズ用に名前が付けられたメソッドが含まれ、 コンポーネントに対して JSF が行うであろう呼び出しを模倣します。 さらに、さまざまなアサーションもあります。

Seam のサンプルアプリケーションにはさらに複雑なケースのデモを行うことができる統合テストが用意されています。 これらのテストは Ant または Eclipse 用の TestNG プラグインを使用して実行します。



35.3.1. 設定

seam-gen でプロジェクトを作成した場合は、 すぐにテストを書き始めることができます。そうでない場合は、Ant、Maven、 Eclipse などのビルドツールでまずテスト環境を設定する必要があります。

少なくとも 次の依存が必要となります。

表35.1

グループ ID	アーティファクト ID	Seam での場所
org.jboss.seam.embedded	**hibernate-all**	**lib/test/hibernate-all.jar**
org.jboss.seam.embedded	**jboss-embedded-all**	**lib/test/jboss-embedded-all.jar**
org.jboss.seam.embedded	**thirdparty-all**	**lib/test/thirdparty-all.jar**
org.jboss.seam.embedded	**jboss-embedded-api**	**lib/jboss-embedded-api.jar**
org.jboss.seam	**jboss-seam**	**lib/jboss-seam.jar**
org.jboss.el	**jboss-el**	**lib/jboss-el.jar**
javax.faces	**jsf-api**	**lib/jsf-api.jar**
javax.el	**el-api**	**lib/el-api.jar**
javax.activation	**javax.activation**	**lib/activation.jar**

Embedded JBoss が起動しなくなるため、 **lib/** にあるコンパイル時の JBoss AS 依存 (**jboss-system.jar** など) をクラスパスに置かないでください。必要に応じて Drools や jBPM などの依存を追加してください。

bootstrap/ ディレクトリには Embedded JBoss の設定が含まれているため、それをクラスパスに含める必要があります。

テストフレームワーク用の **jar** ファイル、 プロジェクト、 テストの他、 JPA および Seam の設定ファイル郡もクラスパスに含めてください。 Seam は Embedded JBoss に指示して **seam.properties** を持つリソース (JAR やディレクトリ) はすべてそのルートにデプロイさせます。ビルドしたプロジェクトを含むディレクトリ構造がデプロイ可能なアーカイブのそれに似ていない場合は、各リソースに **seam.properties** を含ませる必要があります。

デフォルトでは、生成されたプロジェクトは **java:/DefaultDS** (Embedded JBoss で HSQL データソースにビルドされたもの) をテストに使用します。 別のデータソースを使用する場合は、 **foo-ds.xml** を **bootstrap/deploy** ディレクトリに置いてください。

35.3.2. 別のテストフレームワークでの SeamTest の使用

Seam では特に設定することなく TestNG に対応しますが、 JUnit など他のテストフレームワークを使用することもできます。 これを行うには次を行う **AbstractSeamTest** の実装を与える必要があります。

- それぞれのテストメソッドの前に **super.begin()** を呼び出します。
- それぞれのテストメソッドの後に **super.end()** を呼び出します。
- 統合テスト環境を設定する **super.setupClass()** を呼び出します。 いずれのテストメソッドよりも先にこれを呼び出してください。
- 統合テスト環境を消去する **super.cleanupClass()** を呼び出します。
- 統合テストの開始時に **super.startSeam()** を呼び出して Seam を起動します。
- 統合テストの終了時に **super.stopSeam()** を呼び出して Seam を正しくシャットダウンします。

35.3.3. モックデータを利用した統合テスト

各テストの前にデータベースにデータを挿入または消去する必要がある場合は、Seam 統合を DBUnit で使用することができます。 これを行うには **SeamTest** ではなく DBUnitSeamTest を拡張します。

DBUnit 用のデータセットを与える必要があります。

DBUnit は flat と XML の 2 種類のデータセットファイル形式に対応します。 Seam の DBUnitSeamTest は flat 形式が使用されるものとみなしますので、使用するデータセットもこの形式であることを確認してください。

```
<dataset>

  <ARTIST
    id="1"
    dtype="Band"
    name="Pink Floyd" />

  <DISC
    id="1"
    name="Dark Side of the Moon"
    artist_id="1" />

</dataset>
```

テストクラス内で **prepareDBUnitOperations()** を無効にしてデータセットを以下のように設定します。

```
protected void prepareDBUnitOperations() {
  beforeTestOperations.add(
    new DataSetOperation("my/datasets/BaseData.xml")
  );
}
```

コンストラクタ引数として他にオペレーションが指定されていない場合は、 **DataSetOperation** は **DatabaseOperation.CLEAN_INSERT** をデフォルトで設定します。 前述の例では **BaseData.xml** に定義された全テーブルを消去し、 **BaseData.xml** に宣言されているすべての行を挿入してから各

@Testメソッドが呼び出されます。

テストメソッドの実行後さらにデータ消去が必要な場合は **afterTestOperations** の一覧にオペレーションを追加します。

TestNG のテストパラメータ **datasourceJndiName** を設定してデータソースに関する情報を DBUnit に伝える必要があります。

```
<parameter name="datasourceJndiName" value="java:/seamdiscsDatasource"/>
```

DBUnitSeamTest は MySQL および HSQL のいずれにも対応します。 使用するデータベースを伝える必要があります。 これをしないとデフォルトでは HSQL に設定されます。

```
 <parameter name="database" value="MYSQL" />
```

また、バイナリデータをテストデータセットに挿入することもできます (これは **Window では未検証**ですので注意してください)。クラスパス上にあるリソースの場所を DBUnitSeamTest に伝えます。

```
<parameter name="binaryDir" value="images/" />
```

HSQL を使用するためバイナリのインポートがない場合は、これらのパラメータはいずれも設定する必要はありません。 ただし、 テスト設定で **datasourceJndiName** を指定しない限り、 テスト実行の前に **setDatabaseJndiName()** を呼び出す必要があります。HSQL、MySQL のいずれも使用していない場合は、いくつかのメソッドを無効にする必要があります。詳細は **DBUnitSeamTest** の Javadoc を参照してください。

35.3.4. Seam メールの統合テスト

警告

この機能はまだ開発中です。

Seam Mail の統合テストはとても簡単です。

```
public class MailTest extends SeamTest {

  @Test
  public void testSimpleMessage() throws Exception {

    new FacesRequest() {

      @Override
      protected void updateModelValues() throws Exception {
        setValue("#{person.firstname}", "Pete");
        setValue("#{person.lastname}", "Muir");
        setValue("#{person.address}", "test@example.com");
      }
```

```
      @Override
      protected void invokeApplication() throws Exception {
        MimeMessage renderedMessage =
          getRenderedMailMessage("/simple.xhtml");
        assert renderedMessage.getAllRecipients().length == 1;
        InternetAddress to =
          (InternetAddress) renderedMessage.getAllRecipients()[0];
        assert to.getAddress().equals("test@example.com");
      }

    }.run();
  }
}
```

いつも通り新しい **FacesRequest** を作成します。**invokeApplication** フック内では、**getRenderedMailMessage(viewId);** を使ってメッセージを表示し、表示すべきメッセージの **viewId** を渡します。メソッドはテストを実行できる表示されたメッセージを返します。また、 標準 JSF のどのライフサイクルメソッドでも使用できます。

標準 JSF コンポーネントのレンダリングには対応していないため、 メールメッセージのコンテンツをテストすることは容易ではありません。

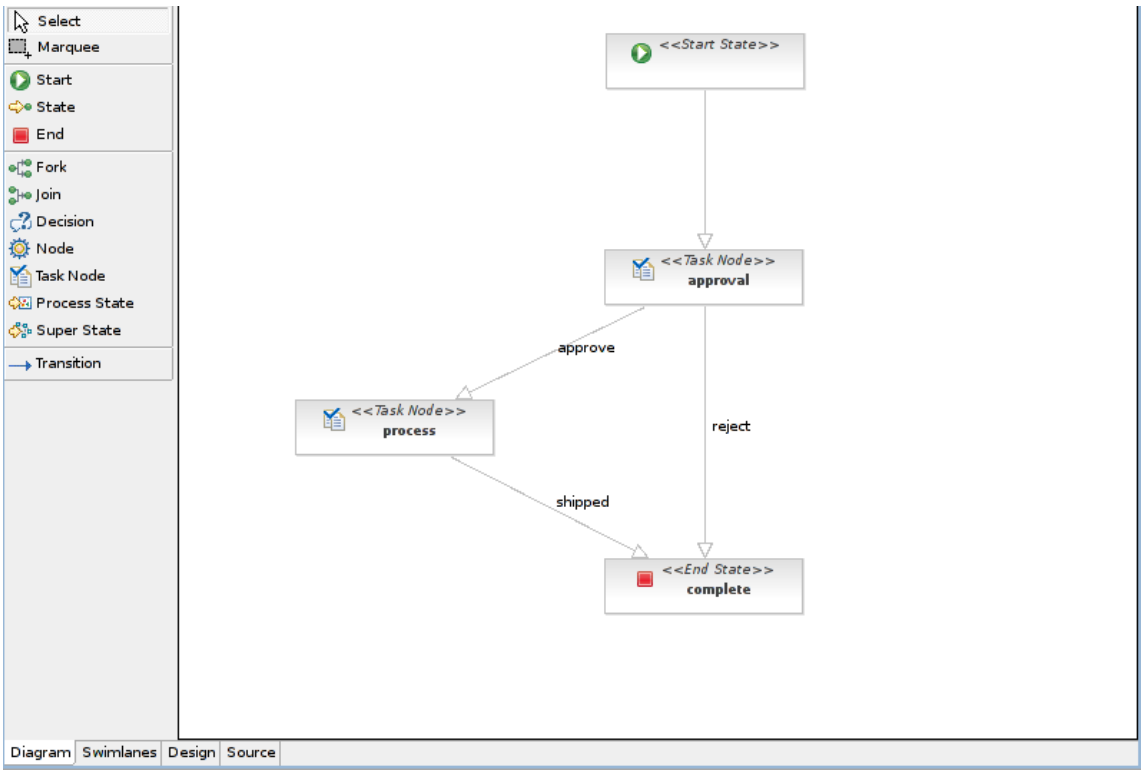
第36章 SEAM ツール

36.1. JBPM デザイナとビューア

jBPM デザイナーとビューアは JBoss Eclipse IDE に含まれています。 ビジネスプロセスやページフローの設計および表示を魅力的にすることが可能です。 詳細は jBPM ドキュメント を参照してください。

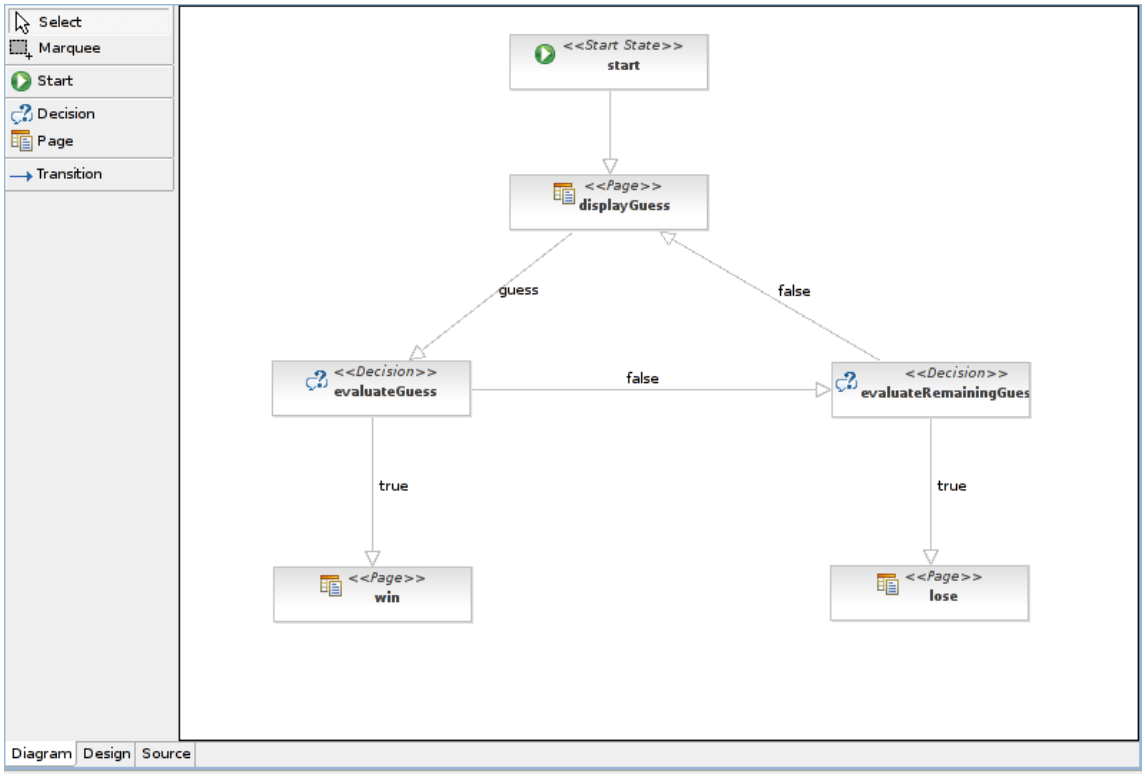
36.1.1. ビジネスプロセスデザイナ

このツールによってグラフィカルに独自のビジネスプロセスを設計することができます。



36.1.2. ページフロービューア

このツールによってグラフィカルなページフローの表示を構築できるため、 複雑なデザインを簡単に共有したり比較したりすることができます。


Select
Marquee
Start
Decision
Page
Transition
<<Start State>>
start
<<Page>>
displayGuess
guess
false
<<Decision>>
evaluateGuess
false
<<Decision>>
evaluateRemainingGues
true
true
<<Page>>
win
<<Page>>
lose
Diagram
Design
Source

第37章 依存性

37.1. JAVA DEVELOPMENT KIT (JDK) の依存性

Seam は JDK™ (Java Development Kit) 1.4 とは動作せず、 またアノテーションや他の機能に対応させるには JDK 6 以降のバージョンが必要です。 Seam では他\tの JDK を使用した徹底的な検証が行われており、 Seam 固有の既知の問題はありません。

37.1.1. Sun の JDK 6 に関する注意点

JDK 6 の旧バージョンが同梱された JAXB のバージョンは Seam とは互換性がなかったため上書きが必要でした。 JAXB 2.1 (JDK 6 Update 4 でリリース) へのアップグレードによりこの問題は解決しました。 ビルド、 テスト、 実行を行うときはこのバージョンまたはこれ以降のバージョンを必ず使用してください。

Seam はそのユニットおよび統合テストで Embedded JBoss を使用します。JDK 6 で Embedded JBoss を使用する場合は、次の JVM 引数を設定する必要があります。 **-*Dsun.lang.ClassLoader.allowArraySyntax=true***

Seam のテストスィートを実行するときにはこれは Seam の内部的なビルドシステムによりデフォルトで設定されますが、 Embedded JBoss を使用する場合には手作業で値を設定する必要があります。

37.2. プロジェクトの依存性

本項では Seam のコンパイルとランタイムの依存性の両方について記載しています。 **EAR** タイプの依存性は、 ご使用のアプリケーションの **EAR** ファイルの **/lib** ディレクトリにそのライブラリを含ませます。 **WAR** タイプの依存性には、 ご使用のアプリケーションの**WAR** ファイルの **/WEB-INF/lib** ディレクトリにそのライブラリを含ませます。 それぞれの依存性のスコープは **all**、 **runtime**、 **provided** のいずれかです (JBoss AS 4.2 または 5.0 により)。

最新バージョン情報および完全な依存性情報は本書には含まれていません。この情報は **/build** に格納されている Maven POM より生成する **/dependency-report.txt** に記載されています。 **ant dependencyReport** を実行するとこのファイルを生成できます。

37.2.1. Core

表37.1

名前	スコープ	タイプ	注記
jboss-seam.jar	all	**EAR**	コアの Seam ライブラリで常に必要です。
jboss-seam-debug.jar	runtime	**WAR**	開発時 Seam のデバッグ機能を有効にする場合に含めます。
jboss-seam-ioc.jar	runtime	**WAR**	Spring と Seam を併用する場合に必要です。
jboss-seam-pdf.jar	runtime	**WAR**	Seam の PDF 機能を使用する場合に必要です。

名前	スコープ	タイプ	注記
jboss-seam-excel.jar	runtime	**WAR**	Seam の Microsoft® Excel® 機能を使用する場合に必要です。
jboss-seam-remoting.jar	runtime	**WAR**	Seam Remoting を使用する場合に必要です。
jboss-seam-ui.jar	runtime	**WAR**	Seam JavaServer Faces (JSF) のコントロールを使用する場合に必要です。
jsf-api.jar	provided		JSF API です。
jsf-impl.jar	provided		JSF リファレンス実装です。
jsf-facelets.jar	runtime	**WAR**	Facelets です。
urlrewritefilter.jar	runtime	**WAR**	URL Rewrite ライブラリです。
quartz.jar	runtime	**EAR**	Seam の非同期機能で Quartz を使用する場合に必要です。

37.2.2. RichFaces

表37.2 RichFaces の依存性

名前	スコープ	タイプ	注記
richfaces-api.jar	all	**EAR**	RichFaces を使用する場合に必要です。 ツリーの作成などアプリケーションからの使用を可能にする API クラスを提供します。
richfaces-impl.jar	runtime	**WAR**	RichFaces を使用する場合に必要です。
richfaces-ui.jar	runtime	**WAR**	RichFaces を使用する場合に必要です。 全 UI コンポーネントを提供します。

37.2.3. Seam Mail

表37.3 Seam Mail の依存性

名前	スコープ	タイプ	注記
activation.jar	runtime	**EAR**	添付のサポートに必要です。
mail.jar	runtime	**EAR**	メール送信サポートに必要です。
mail-ra.jar	compile only		メール受信サポートに必要です。
jboss-seam-mail.jar	runtime	**WAR**	Seam Mail です。

37.2.4. Seam PDF

表37.4 Seam PDF の依存性

名前	タイプ	スコープ	注記
itext.jar	runtime	**WAR**	PDF ライブラリです。
jfreechart.jar	runtime	**WAR**	チャートライブラリです。
jcommon.jar	runtime	**WAR**	JFreeChart で必要です。
jboss-seam-pdf.jar	runtime	**WAR**	Seam PDF のコアライブラリです。

37.2.5. Seam Microsoft®Excel®

表37.5 Seam Microsoft®Excel® の依存性

名前	タイプ	スコープ	注記
jxl.jar	runtime	**WAR**	JExcelAPI ライブラリです。
jboss-seam-excel.jar	runtime	**WAR**	Seam Microsoft® Excel® のコアライブラリです。

37.2.6. JBoss Rules

JBoss Rules (Drools) のライブラリは Seam の **drools/lib** ディレクトリにあります。

表37.6 JBoss Rules の依存性

名前	スコープ	タイプ	注記
antlr-runtime.jar	runtime	**EAR**	ANTLR ランタイムライブラリです。

名前	スコープ	タイプ	注記
core.jar	runtime	EAR	Eclipse JDT です。
drools-api.jar	runtime	EAR	
drools-compiler.jar	runtime	EAR	
drools-core.jar	runtime	EAR	
drools-decisiontables.jar	runtime	EAR	
drools-templates.jar	runtime	EAR	
janino.jar	runtime	EAR	
mvel2.jar	runtime	EAR	

37.2.7. JBPM

表37.7 JBPM の依存性

名前	スコープ	タイプ	注記
jbpm-jpdl.jar	runtime	EAR	

37.2.8. GWT

これらのライブラリは Seam アプリケーションで Google Web Toolkit (GWT) を使用する場合に必要です。

表37.8 GWT の依存性

名前	スコープ	タイプ	注記
gwt-servlet.jar	runtime	WAR	GWT Servlet ライブラリです。

37.2.9. Spring

これらのライブラリは Seam アプリケーションで Spring Framework を使用する場合に必要です。

表37.9 Spring Framework の依存性

名前	スコープ	タイプ	注記
`spring.jar`	runtime	**EAR**	Spring Framework ライブラリです。

37.2.10. Groovy

これらのライブラリは Seam アプリケーションで Groovy を使用する場合に必要です。

表37.10 Groovy の依存性

名前	スコープ	タイプ	注記
`groovy-all.jar`	runtime	**EAR**	Groovy のライブラリです。

付録A 改訂履歴

改訂 5.1.2-2.400 **2013-10-31** **Rüdiger Landmann**
Rebuild with publican 4.0.0

改訂 5.1.2-2 **2012-07-18** **Anthony Towns**
Rebuild for Publican 3.0

改訂 5.1.2-100 **Thu 8 December 2011** **Russell Dickenson**
JBoss Enterprise Application Platform 5.1.2 GA の変更が含まれます。このガイドの内容の変更については、『リリースノート 5.1.2』 を参照してください。