09 Juin 2009

Présentation du framework JBoss Seam

par Patrice Pichereau, directeur technique
et Mikael Robert, ingenieur JEE

Introduction :

Java EE 5 :

Aujourd'hui de nombreux projets Java EE sont construits autour de spécifications communes.
On peut citer

EJB :

La technologie EJB ou Enterprise JavaBeans est une architecture de composants logiciels côté(s) serveur(s) pour la plateforme de développement JEE.
Cette architecture définie un cadre pour la création de composants distribués (déployés sur des machines distantes).

La spécification EJB permet de séparer l'abstraction de données des services de l'application. Elle incite de plus à développer des composants réutilisables et autonomes et permet une gestion affinée des transactions métiers grâce à JTA.

Intégrés à la plate-forme JEE en 2001 en version 2.0, les EJBs ont pour but de fournir une technologie de composants distribuables et transactionnels. La lourdeur d'EJB 2.0, puis 2.1 fût à l'origine d'une levée de boucliers de la part d'un certain nombre d'acteurs du monde Java.
En 2004, Rod Johnson et Juergen Holler lancent le framework Spring, qui constitue une alternative à EJB 2.1 en proposant la création de composants (bean) transactionnels et supportant l'injection de dépendance.
En 2005, EJB 3.0 voit le jour. Il n'a pas grand chose à voir avec son prédécesseur si ce n'est le nom. En effet, EJB 3.0 est une technologie très puissante, et beaucoup plus facile à mettre en œuvre que son ancêtre.

JPA via Hibernate :

Hibernate est un Framework java de mapping objet relationnel, implémentant JPA. Il permet de faire le mapping entre objets Java et objets stockés en base relationnelle. De plus il en assure la persistance.
Hibernate a été écrit sous la responsabilité de Gavin King qui fait partie de l'équipe JBoss et qui est aussi le fondateur de Seam.
Il permet un gain de productivité énorme dans le sens où il utilise la structure d'objets annotés avec JPA pour créer et modifier les tables.
Concrètement, cela signifie qu'Hibernate permet de manipuler les données d'une base de données relationnelle sous forme d'objet.

JSF :

JSF ou Java Server Faces est un framework java pour le développement de clients riches pour les applications web. Contrairement à la majorité des frameworks MVC à base d'actions, JSF est basé sur la notions de composants gérant des événements, il reprend le modèle d'autres frameworks pour l'IHM tel que Swing ou SWT ou l'état d'un composant est enregistré lors du rendu de la page pour être ensuite restauré au retour de la requête.

Malheureusement lorsqu'il voit le jour en 2004, JSF est marqué par un certain nombre de défauts qui ont considérablement ralenti son adoption (complexité à mettre en œuvre, lourdeur des fichiers de configuration XML, impossibilité d'utiliser du code HTML, pas de mécanisme de templating). Facelets apportera une solution aux deux derniers points.
Ces défauts ont considérablement ralenti l'adoption de JSF tout en favorisant l'émergence de frameworks orientés composant comme Wicket.
Aujourd'hui avec Seam les derniers défauts de JSF sont levés et l'arrivée de JSF 2 permettra d'unifier toutes les technologies utilisées pour rendre JSF pleinement exploitable pour le développement web.

Ces technologies sont très nombreuses, et on ne sait parfois pas quoi choisir pour un cas d'utilisation bien précis. Il est aussi courant de n'avoir besoin que d'une toute petite partie d'une technologie mais d'être rebuté à l'utiliser vu la complexité de configuration et d'intégration qu'elle va apporter.

Bien que très puissants, JSF et EJB 3.0 sont très éloignés. Créer une application JEE 5.0 nécessite donc beaucoup de code pour les faire cohabiter.
Seam a été conçu pour combler ce vide et faire la jointure entre ces deux technologies.
Toutefois, Seam intègre aussi un conteneur de composant permettant de se passer des EJB (ou même, d'utiliser Spring) tout en conservant un moteur transactionnel.

Unification de Java EE

Une application multi-tiers est donc typiquement fabriquée à partir d'un ensemble de composants : Services Transactionnels, services de sécurités, Persistance des données, envoi de messages Asynchrone, composant d'interfaces graphique etc ...
Parmi eux on l'a vu, on peut citer surtout EJB3, JPA (via Hibernate), JSF,EL, JMS, JAAS, iText, Groovy, JBoss Rules, Ajax .. etc
Vous connaissez tous une partie ou l'essentiel de ces technologies, au moins de nom.
Néanmoins, le problème qui revient de manière récurrente lorsqu'on entame un projet autour de ces spécifications est tout d'abord : que choisir parmi la quantité invraisemblable de technologies de Java EE ?

Ensuite il faut limiter les choix car l'intégration et la communication des différents aspects technologiques au sein du projet peut vite devenir très voir trop complexe.

L'intégration est donc l'un des problèmes les plus ardus dans le développement d'applications d'entreprises.
En effet, tout d'abord la configuration du projet intégrant ces spécifications est souvent laborieuse et longue.
Mais de projets en projets on refait encore et toujours ces même étapes.

Et bien aujourd'hui nous allons vous présenter une solution à cette problématique : Seam.

Seam est d'abord parti du principe que la spécification java EE 5 incorpore deux composants d'architecture qui sont des clefs pour créer des applications web : JSF et EJB3.
JSF est le standard de JEE5 pour la couche présentation qui fournit à la fois l'interface utilisateur, modèle de composants et de gestion événementielle coté serveur.
EJB3 est le standard de modèle de composant pour créer des composants métiers sécurisés et évolutifs , qui accèdent aux ressources de manière transactionnelle.
Mais EJB3 englobe aussi Java Persistance API qui définit un standard de modèle de persistance pour faire le mapping objet relationnel.

Néanmoins les deux architectures ont un soucis, elles nécessitent des intermédiaires pour communiquer, un code qui va servir à les assembler (les coller) autant d'un coté configuration XML que code Java (les backing beans JSF).
La connexion des EJB aux pages JSF est tout à fait faisable mais sous certains aspects, trop complexe.
Seam a pris en charge cette responsabilité et permet à JSF et EJB3 de fonctionner enfin directement ensemble.
En partie grâce à cela, Seam s'est positionné lui même comme un prototype de la future spécification java EE 6.
Trois JSR (JSR = Java Specification Request) importantes ont été acceptées :
JSR 299 les Web Beans dont le but est d'unifier les managed bean de JSF avec le modèle de composant EJB.
JSR 314 JSF 2.0
JSR 303 : Bean Validations.

Cependant ne soyez pas inquiet d'une éventuelle limitation : Seam ne va pas vous rendre complètement dépendant d'EJB et de JSF.
En effet coté vue vous pourrez changer alternativement pour Wicket, Tapestry, GWT ou Flex et autres à la place de JSF.
Et coté métier Seam permet le support des JavaBeans comme composants transactionnels. De plus l'intégration du conteneur de Spring est tout à fait possible, et ce toujours dans cette volonté de simplicité d'intégration et de configuration.

Vous devez voir le but de Seam comme celui de fournir au développeur une seule source de technologies.

L'idée initial de Gaving King, créateur d'Hibernate et de Seam, était de mettre en place une fondation qui autoriserait le contexte de persistance (Session Hibernate ou EntityManager de JPA) à traverser les couches et autoriserait les EJB Stateful à répondre directement aux composants d'interface JSF.
Pour supporter cette vision, Seam encourage donc l'adoption d'une architecture Stateful pour gérer la continuité des interactions utilisateurs ou événementielles.

Le nom Seam (couture en français) a été choisi pour le projet car il apporte une fondation qui intègre JSF et EJB 3 ensembles et assemblés, et leur permet de communiquer simplement dans le même conteneur.

Dans le processus de résolution des problèmes entre JSF et EJB3, les architectes de Seam ont inclus la possibilité d'inclure les POJO comme des composant métiers, et de ne pas se limiter aux EJB.
Les composants Seam sont donc des POJOs simplement enrichis d'une annotation simple et intègrent des services implicites ou déclarés communs aux EJB tels que les transactions, les intercepteurs, la gestion des threads et de la sécurité.
Pour les composants non EJB (pojo, javabeans, springs beans), Seam se charge de traiter les annotations Java EE 5 ou leurs synonymes de l'api Seam en les redirigeants sur les services auto gérés, grâce à ses propres surcharges.

Seam est aussi parti du principe de réduire voir de supprimer la configuration XML longue, difficile à relire et récurrente aux applications JEE utilisant JSF et EJB3, et tant qu'à faire ils l'ont fait pour l'ensemble des éléments intégrés au framework.

Tout cela vous permet entre autre de reconsidérer votre choix d'utiliser les EJB si vous n'avez pas particulièrement besoin de toutes leur possibilités. En effet vous n'êtes pas obligé d'utiliser JBoss Application Server pour déployer une application Seam en dépit de ce que vous pourriez avoir entendu.
Bien que la documentation de Seam recommande l'utilisation de JBoss (n'oublions pas que Seam est soutenu par JBoss), Seam est peut être le framework respectant le plus java EE 5 et de ce fait il est compatible avec de nombreux serveurs d'applications.
- Weblogic,
- WebSphere,
- OC4J,
- Tomcat (si vous n'utilisez pas EJB)
- Glassfish

Avant de passer à une présentation plus détaillée, j'aimerais vous présenter un petit listing des améliorations apportées par Seam à la plateforme Java EE 5 :

- Eliminer les lacunes de JSF qui ont fait l'objet d'innombrables débats.
- Permettre la communication entre JSF et les composant métiers transactionnels
- Supprimer les couches non nécessaires.
- Offrir une solution pour une gestion d'état contextuelle, décourageant l'utilisation d'une architecture Stateless.
- Gérer le contexte de persistance (Session Hibernate ou EntityManager de JPA) pour prévenir les Exceptions de Lazy initialisation dans la vue et les requêtes en descendant.
- Fournir un moyen d'étendre le contexte de persistance à la durée d'un cas d'utilisation.

- Connecter les vues aux pages flows stateful.
- Apporter la gestion des processus métiers au monde des applications web
- Connecter aux POJOs les mécanismes d'authentification et d'autorisation gérés par JAAS, accessible via les EL et pouvant être étendues en utilisants des règles déclaratives.
- Fournir un conteneur d'EJB embarqué pour réaliser des tests dans un environnement non JEE.
- Fournir plus de 30 exemples de références dans la version fournie du framework.

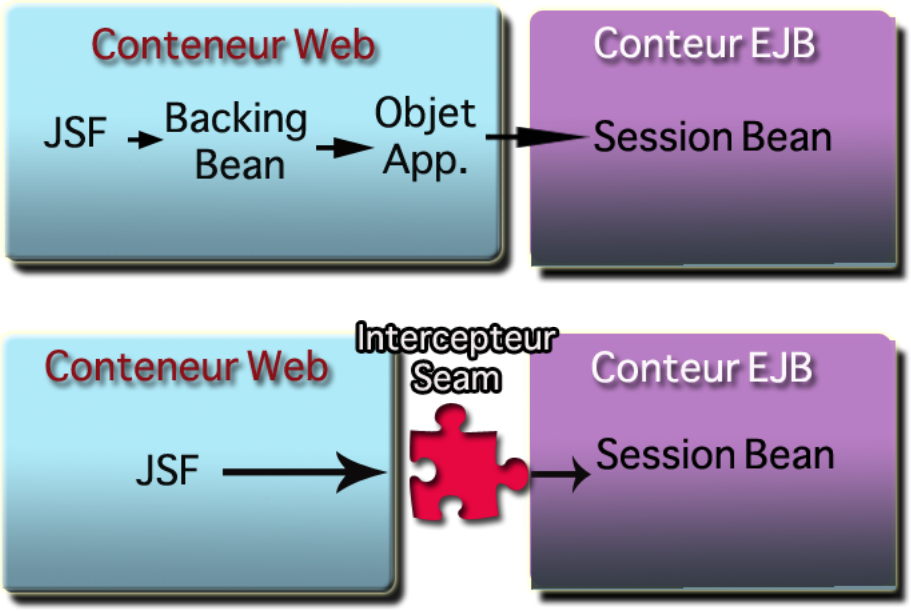
Composants Seam

Seam a apporté en plus de l'intégration simplifiée, son modèle de composant transactionnels, conçus sur un modèle très similaire à celui des EJBs.

Tout d'abord, un composant Seam est un objet de type POJO (Plain Old Java Object) ou EJB3 qui contient simplement une annotation @Name paramètrée avec son nom logique. C'est ce nom qui permettra de l'invoquer depuis une page JSF grâce à au service de nommage de Seam et à JBoss EL.

Ainsi, ce modèle de composant a été conçu pour permettre l'interaction directe avec JSF.
Dans le développement classique avec JSF on ne peut interagir directement avec les EJB, ce qui oblige à utiliser un objet interface.
On a donc au minimum deux composants pour gérer le contenu dynamique de chaque page JSF.

Seam permet alors une autre approche grâce à la notion d'intercepteurs mise en place pour cela, on peut désormais invoquer directement le composant métier, depuis la page JSF selon le modèle présenté ici.
Ces composants qu'ils soient EJB ou composants Seam, seront ensuite gérés de manière transactionnelle par le framework, ce qui permet une robustesse et une sûreté bien supérieure à ce qui se faisait avant. Cela permet aussi dans de nombreux cas de ne plus utiliser d'EJB à moins d'avoir absolument besoin de leurs particularités.
En effet malgré tous leurs avantage les EJB, et surtout leur gestion, sont très gourmands en



ressource, en tout cas plus que les composants Seam.

Accélération du développement

Nous allons maintenant vous présenter quelques exemples de facilités apportées par Seam au niveau du binding entre JSF et le code Java.

Commençons par la déclaration de composants Seam.

```
@Name("textEntryList")
@Scope(ScopeType.PAGE)
public class TextEntryList {
```

Ici on vois l'annotation @Name, c'est elle, on l'a vu, qui permet de déclarer un POJO classique comme un composant Seam, et ainsi de profiter de tous les avantages de ceux-ci.
Cette annotation va aussi permettre d'accéder directement à l'objet via le nom précisé dans l'annotation, dans la page JSF.

Fini donc, les configurations via les backing beans et les fichiers XML, tout est désormais automatisé par Seam.
Ainsi dans la page JSF nous appellerons l'objet TextEntryList de cette façon :

```
<rich:datascroller for="textEntryListTable" page="#{textEntryList.currentPage}"/>
```

Dans cet exemple on l'utilise pour récupérer la page courante et la passer à un composant de pagination.

Voyons maintenant les types d'objets que l'on peut passer à une page JSF.
Lorsqu'on est habitué à travailler avec JSF, on n'utilise pas directement de Listes d'EJB entity, mais un composant SelectItems, qui sera passé au composant JSF responsable du rendu de celui-ci.
Et bien, ici l'objet TextEntry est un EJB Entity, et la liste que vous pouvez voir est belle est bien une List de java.util.List. Seam apporte ce mécanisme pour encore plus décharger l'application d'intermédiaires.

```
@DataModel
private List<TextEntry> textEntries;

 * The current text entry.
@Out(required = false)
@DataModelSelection
private TextEntry textEntry;
```

Ainsi vous pourrez directement appeler le contenu d'une entité en état attaché, par une EL dans la JSF.
Imaginez le temps gagné ... :

```
<h:outputText value="#{entry.content}" escape="false" />
```

entry désigne bien ici une Entité JPA.

Les annotations maintenant, ici **@DataModel** permet de définir l'attribut annoté comme un Datamodel JSF,
L'annotation en question renvoi a JSF un attribut de type liste a la page JSF sous la forme d'une instance de javax.faces.model.DataModel.
Ce qui permet d'utiliser la liste directement dans une page JSF.

Mais ceci va aussi nous permettre d'utiliser, sur un autre attribut l'annotation **@DataModelSelection** qui sera elle automatiquement bindée par Seam à la ligne sélectionnée du DataModel affiché. Ainsi on peut directement travailler sur l'élément sélectionné par exemple dans un tableau de données, par l'utilisateur.

```
@Factory("textEntries")
public void initTextEntries() {
```

L'annotation **@Factory**, maintenant demande à Seam dans cet exemple de créer une instance de TextEntryList puis d'invoquer la méthode initTextEntires pour initialiser textEntries, que l'on pourra invoquer directement dans la JSF, sans même préciser le nom de l'objet l'encapsulant.
Certains y verront un lien avec un design pattern bien connu.

```
<rich:dataTable id="textEntryListTable" value="#{textEntries}" var="entry" style="border-style: none;" rows="5">
  <rich:column style="border-bottom: 0px; border-right: 0px;">
   <rich:panel>
    <f:facet name="header"> <h:outputText value="#{entry.getFormatedTitle()}" /> </f:facet>
    <h:outputText value="#{entry.content}" escape="false" />
```

Une autre annotation, Unwrap, relativement similaire, demande à Seam de fournir en type de retour, celui de la méthode annotée : List, au lieu du composant lorsqu'on appellera l'objet par son nom (**@Name**) dans une JSF. Ainsi si vous avez un composant nommé «composant» et que vous avez encapsulé dans celui-ci une méthode annotée @Unwrap, l'appel a «composant» renverra le type de retour de la méthode annotée.

Entity Query et Entity Home

Dans de nombreuses applications sont présents des types de données sur lesquelles on ne réalise que des opérations de type CRUD.
De ce fait, afin d'éviter les répétitions de code inutiles, Seam a mis en place deux types de composants très simple à mettre en oeuvre : EntityHome et EntityQuery.

EntityQuery

```
28  @Name("tagList")
29  public class TagList extends EntityQuery<Tag> {
30
31      /**
32       *
33       */
34      private static final long serialVersionUID = 1458243491
35
36      /**
37       * Constructor.
38       */
39      public TagList() {
40          setEjbql("select tag from Tag tag");
41      }
42  }
```

EntityQuery est un composant Seam qui s'utilise via l'héritage.
Ce composant vous permet de gérer pour des CRUD simple, des listes de données, simplement en surchargeant la méthode setEJBQL de votre objet en héritant.

EntityHome

EntityHome est un composant Seam basé sur le pattern Mediator que vous pouvez utilisez par l'héritage pour gérer une instance d'entité. Celui ci se chargera du lien avec le cache du gestionnaire de persistance. Son rôle est aussi de coordonner les opérations de type CRUD sur l'instance avec le gestionnaire de persistance.
Seam fournit deux type d'implémentations de Home : Un pour JPA : EntityHome, et un autre pour Hibernate : HibernateEntityHome.
Un composant Home est donc en fait le gardien d'une instance d'Entité. C'est le changement d'Id de l'entité géré, qui va déclencher la mise à jour de l'instance d'entité gérée par le composant. Par défaut ces composants sont stockés en contexte conversation.
Voici un exemple de composant EntityHome ici pour gérer une Entité Tag, comme vous pouvez le voir le code à écrire pour avoir un mécanisme CRUD sur une entité est très court.

```
32  @Name("tagHome")
33  public class TagHome extends EntityHome<Tag> {
34      /**
35       *
36       */
37      private static final long serialVersionUID = -14490956848919765L;
38      /**
39       * The tag id.
40       */
41      @RequestParameter
42      private Long tagId;
43
44      /**
45       * Return the id of the current managed tag.
46       * @return tagId the tag id.
47       */
48      @Override
49      public Object getId() {
50          if (tagId == null) {
51              return super.getId();
52          } else {
53              return tagId;
54          }
55      }
56
57      /**
58       * Initialization method.
59       */
60      @Override @Begin(join = true)
61      public void create() {
62          super.create();
63      }
64
65  }
```

Ces objets générés présentent de nombreux avantages pour un CRUD simple.
Toutefois ils sont déconseillés dès que la gestion de l'entité sort du contexte de CRUD.

L'inversion de contrôle absolue : le mécanisme de Bijection

Le pattern IOC (Inversion of control) est utilisé par Seam dans sa gestion de l'injection.
Pas seulement pour câbler les composants entre eux, mais aussi pour produire des variables de contextes et permettre à vos composants de communiquer entre eux via des notifications événementielles.
L'interaction entre composants, on l'a vu n'est pas un mécanisme évident en JEE. Seam a pour cela mis en place un mécanisme de gestion bijectif des dépendances.

Un composant peut alors être injecté automatiquement dans un autre composant (injection), et de la même façon un composant peut créer et projeter un autre composant dans un contexte qu'il définit (outjection). Il deviendra ainsi accessible par tous les composants qui connaissent son contexte. Ce mécanisme est très facile à mettre en œuvre, il requiert juste une très bonne compréhension des contextes, et l'utilisation d'annotations : @In et @Out.
Seam élimine donc toute la partie XML très verbeuse qui était nécessaire pour gérer les accès entre composants.

Je ne suis pas entré dans les détails tout à l'heure mais vous avez pu voir dans un des extraits de code précédent l'annotation @Out :

```
@DataModel
private List<TextEntry> textEntries;

 * The current text entry.
@Out(required = false)
@DataModelSelection
private TextEntry textEntry;
```

Et bien cette annotation va permettre d'outjecter dans le contexte courant, l'objet textEntry pour les surcharges Seam de JSF. Dans l'exemple de tout à l'heure il s'agissait d'exposer à la JSF l'attribut correspondant à la ligne sélectionnée par l'utilisateur.

Mais cela est surtout très utile quand vous voulez pouvoir via un autre composant, en reinjecter un.

```
@In
private EntityManager entityManager;
```

Par exemple l'EntityManager d'Hibernate ici, est outjecté par Seam lors de son initialisation, ce qui nous permet de l'injecter très facilement pour l'utiliser. Et cela fonctionne pour n'importe quel objet outjecté, tant que vous respecter le nom de celui-ci. En effet Seam utilise le nom donné à l'attribut pour rechercher l'objet dans le contexte.

Simplification de JSF

- La validation est directement intégrée à JSF.
Habituellement, on doit valider au niveau de l'interface et au niveau des EJBs ici la validation se fait automatiquement des deux cotés

```
@NotNull(message="Credit card number is required")
@Length(min=16, max=16, message="Credit card number must 16 digits long")
@Pattern(regex="\\d*", message="Credit card number must be numeric")
private String creditCard;
```

Grâce à s:validate, la validation est intégrée à la phase de validation du cycle JSF.

```
<div class="input">
    <h:inputText id="creditCard" value="#{booking.creditCard}">
        <s:validate />
    </h:inputText>
    <s:message id="message" styleClass="error errors"/>
</div>
```

- On l'a vu précédemment, les backings beans sont éliminés, et donc la couche métier peut être invoquée directement depuis la page.

Jboss EL

Seam utilise JBoss EL, qui fournit une extension à l'implémentation standard de « Unified Expression Language (EL) ». JBoss EL apporte certaines améliorations pour simplifier et augmenter la puissance des expressions EL.

Expressions paramétrées

Les expressions standard EL ne permettent pas d'utiliser une méthode définie avec des paramètres.
Jboss EL supprime cette restriction.
Par exemple :

```
<h:commandButton action="#{hotelBooking.bookHotel(user, hotel)}" value="Book Hotel"/>
```

```
@Name("hotelBooking")
public class HotelBooking {

    public String bookHotel(User user, Hotel hotel) {
        // Book the hotel
    }
}
```

Binding de méthodes

Unified EL permet de lier la valeur d'un backing bean à un composant JSF. Les « value expressions » utilisent la convention de nomage JavaBean, mais uniquement pour les accesseurs (un getter et un setter). Souvent, la page JSF n'a besoin que de retrouver une valeur (par exemple, pour les attributs « `rendered` »). Les objets n'ont pas forcément d'accesseur pour ce paramètre.
JBoss EL supprime cette contrainte en autorisant l'utilisation de méthodes.
Par exemple :

```
<h:outputText value="#{person.name}" rendered="#{person.name.length() > 5}" />
```

Il est aussi possible de passer un paramètre :

```
#{controller.productsByColor('blue')}
```

Projection

Jboss EL supporte la sytaxe projection de manière limitée. Une projection définit une sous expression d'une expression multiple (List, Set, etc...)
Par exemple :
#{company.departments} peut retourner une liste de départements pour une société : si nous ne nous intéressons qu'au nom du département, il va falloir itérer sur la liste pour obtenir chaque valeur.
Jboss EL permet d'obtenir cette liste grâce à une expression de projection :
`#{company.departments.{d|d.name}}`

Il est aussi possible d'obtenir le nombre de départements grâce à l'expression suivante :
`#{company.departments.{d|d.size()}}`

Ou encore, d'obtenir le nom de tous les employés, quel que soit leur département :
`#{company.departments.{d|d.employees.{emp|emp.lastName}}}`

Ajax

Avec Ajax4JSF intégré à Richfaces, ou Icefaces et l'intégration de ces deux librairies au framework, l'ajax est largement facilité y compris en liaison avec des objets Seam, des EJB Stateful, ou des Pojo classiques.

Les annotations plutôt que le XML

Un des moyens utilisées par Seam pour simplifier la configuration, a été la suppression du code XML inutile.
Bien que pensé pour être flexible, la configuration XML est externe et devient rapidement laborieuse à maintenir et déconnectée de la logique de l'application.
De plus la compréhension de la majorité des code JEE demande la relecture de fichiers xml très verbeux, et peu lisibles. Ce qui rend entre autre la revue de code très longue.
Seam a mis en place tout un ensemble d'annotations placées dans le code de l'objet, qui permettent de visualiser directement la configuration.
Par exemple :
@Transactional, @Asynchronous, @Install ,@Startup @Name, les annotations d'Injection/outjection @In / @Out, @Observer, @Converter, @Validator etc ...
De plus une bonne partie des annotations Seam sont paramètrables avec des EL, ce qui permet une configuration par annotation dynamique.

Seam Gen

1. Génération

Débuter avec un Framework est souvent difficile et très long. Il faut mettre en place l'environnement, les librairies, déployer ...
Et on se retrouve parfois avec de nombreuses galères durant cette étape, qu'on ne sait pas vraiment résoudre parce qu'on ne connaît pas encore le fonctionnement du framework.
Un des objectifs de Seam est de tout faire pour que le développeur ne se concentre que sur le code métier qu'il a développer, qu'il puisse s'y atteler le plus vite possible, et non qu'il perdre du temps sur l'intégration et le code purement technique, afin de gagner en productivité.

Pour cela, vous pourrez utiliser un outil très pratique : Seam-Gen.
Il existe aussi un plugin pour eclipse, intégré dans l'ensemble de plugins JBossTools qui permet de réaliser les mêmes actions que l'utilitaire en ligne de commande.

Il s'agit d'un outil qui va vous permettre de créer une structure simpliste de projet CRUD, prêt à être déployé, un peu à la manière des scaffolding de certains autres framework ou langages.

L'utilitaire va alors vous poser une série de question comme le nom du projet, le type de base de données et son adresse, le type de projet (ear avec ejb, war sans), richfaces ou icefaces comme librairie de composants JSF etc ...
Une fois la série de question terminée, Seam gen vous génère un projet prêt au développement et à être déployé pour les premiers tests.
En plus de cela les projet générés par Seam gen, gèrent le déploiement à chaud via un script ant de tout le contenu : Pojo, JSF, configuration. Toutefois les EJBs requièrent toujours un redémarrage du serveur.

En plus de cela, cet outil va vous permettre de générer :

- Des «Actions» à savoir, un composant Seam avec une méthode du nom voulu, et la page JSF avec un bouton pour appeler la méthode, vous n'avez plus qu'à remplir le corps de la méthode.
- Des Entités, à savoir un EJB Entité avec des attributs de base (Id, Version, Nom), un composant EntityQuery, et un composant EntityHome pour la gérer.
- Des conversations : EJB session Stateful en contexte conversation, et l'interface liée.
- Des formulaires : Créer l'interface et l'EJB Stateful pour gérer le formulaire ainsi que les tests unitaires (pour testNG) qui simulent une requête et une réponse JSF.

Et pour le plugin eclipse :
- Des pageflows avec un éditeur graphique.
- Des composants Seam

2. Reverse Engineering avec Seam gen

Un autre apport important de cet outil, est le reverse engineering sur une base de donnée.
En effet Seam gen est capable d'analyser un schéma relationnel et de générer les classes des entités JPA depuis une base de donnée existante, ainsi qu'une interface CRUD complète avec une interface basée sur des templates Facelets et des composants JSF et richfaces/iceface.
Cette fonctionnalité peut être très utile si on souhaite refaire totalement une application existante, et faire gagner beaucoup d'heure d'écriture de classes JPA.

Events et Observer : Gestion des événements asynchrone avec Seam.

Parmi les nombreux apports de Seam il y en a un qui est particulièrement utile : la gestion des événements (Events). Et celle-ci devient vraiment intéressante par sa simplicité d'utilisation.
Tout d'abord quel est l'intérêt ? Admettons que vous avez un composant qui réalise une tâche interne à votre application (ajout de données, modification, visualisation d'une page … tout et n'importe quoi en fait).
Or suite à cette action vous avez un ou plusieurs process à lancer après, par exemple envoi d'un mail, enregistrement d'un statut en base etc … bref votre première action, doit être suivie d'autres.
Dans un développement classique vous allez enchaîner les appels de méthodes pour pouvoir réaliser votre process complet, ou bien mettre en oeuvre JMS qui est relativement gourmand en ressources.

Or si ce process change, cela deviens vite laborieux de modifier votre enchevêtrement de méthodes, de tests ou votre envoi de message JMS.

Et bien c'est là qu'est l'intérêt des événements : vous allez pouvoir pour une action donnée déclencher un événement, une sorte de message qui sera reçu par un ou plusieurs composants de l'applications, complètement indépendants de votre émetteur d'événements.
Toute fois on peut se demander l'intérêt de la chose par rapport à JMS : disons pour aller vite, vu qu'il ne s'agit là que d'une présentation non exhaustive de Seam, qu'il s'agit ici vraiment d'une gestion événementielle, et non d'un envoi de message. Ensuite, il est beaucoup plus simple à mettre en oeuvre que JMS.

Ce modèle présente un avantage majeur : vous développez vraiment des composants indépendants : vous devez juste spécifier quels événements ils doivent écouter (à la manière de Threads avec les signaux dans des langages plus bas niveau que Java).
Cela vous évite donc les modèles classiques d'objets interdépendants, ou la gestion des dépendances devient laborieuse dans une grosse application.
Tout ce que vous avez à faire dans le composant émetteur est de déclencher un événement sans vous préoccuper de quel ou combien de composants seront appelés après, ou de quel appel sera réalisé après.
Ensuite à vous de mettre en place les objets "Observateurs' de l'événement en question.

Voyons un exemple très simple :
Mettons en place un exemple simple : Ici un objet héritant d'EntityHome<TextEntry>, ou TextEntry est un EJB Entity déjà défini.

```
@Name("textEntryHome")
public class TextEntryHome extends EntityHome {
    @RequestParameter
    private Long textEntryId;
    /** ... Code tronqué pour l'exemple */
    }
    @Override
    public String persist() {
        Events.instance().raiseEvent("postAdded", this.getInstance().getId());
        return super.persist();
    }
}
```

La ligne intéressante est donc celle ci :
`Events.instance().raiseEvent("postAdded", this.getInstance().getId());`
Simplement, lorsqu'une Entité de type TextEntry est persistée, on déclenche l'événement "postAdded" avec comme paramètre l'Id de l'entity.
Vous pouvez passer ce que vous voulez en paramètre et le récupérer plus tard lors du traitement de l'événement par le composant observateur.

Vous pouvez aussi déclencher un événement sur l'appel d'une méthode grâce à l'annotation @RaiseEvent("nomEvent") sur la méthode, au lieu de faire un appel dans le corps de la méthode. Cela vous permet de complètement sortir l'événement du code métier.

Enfin vous pouvez aussi décider d'envoyer l'événement de manière Asynchrone en utilisant raiseAsynchronousEvent au lieu de raiseEvent. Si vous déclenchez un événement de manière asynchrone celui-ci sera géré par le service timer du conteneur d'EJB. Imaginez alors les possibilités d'un tel mécanisme.
Voyons maintenant comment rattraper cet événement, et bien il s'agit d'un simplement d'un Pojo, avec une méthode annotée @Observer:

```
@Observer("postAdded")
@Asynchronous
public void processEvent(Long id) {
    TextEntry textEntry = entityManager.find(TextEntry.class, id);
    if (textEntry != null) {
        logger.info("Text entry with title : #0 has been added ",
                                                textEntry.getTitle());
    }
}
```

Voilà, comme vous pouvez le voir ce qu'on fait ici est simple : l'événement peut être récupéré simplement grâce à l'annotation @Observer, avec en paramètre le nom de l'événement.
Cet exemple illustre bien le fonctionnement du mécanisme et sa facilité de mise en oeuvre.
Les paramètres de la méthode doivent avoir le même ordre que ceux passés à l'événement pour pouvoir les récupérer.
Seule une annotation suffit donc pour récupérer l'événement et ses paramètres, et vous pouvez mettre autant d'Observers que vous le souhaitez en écoute du même événement.

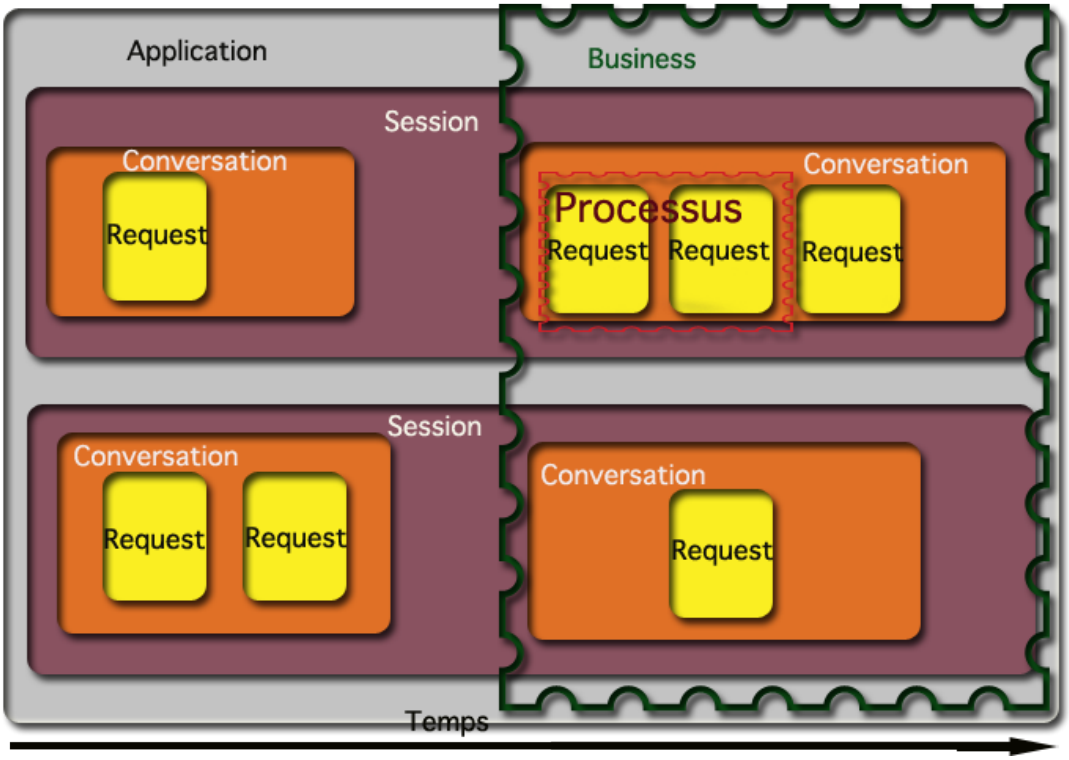
Enfin ici la méthode a été annotée @Asynchronous pour rendre le processus de traitement complètement asynchrone. Juste encore un exemple d'annotation Seam simplifiant la vie du développeur. Le framework regorge de petits éléments comme celui-ci qui deviennent vite indispensables.

Si vous maîtrisez le concept, et commencez à le mettre en oeuvre avec des batchs Quartz par exemple, vous pouvez arrivez à une gestion vraiment très fine et optimisée de vos processus tout en développant des objets qui deviennent réellement des composants indépendants.
Attention cependant à ne pas abuser des événements, ce mécanisme n'est pas fait pour être utilisé à tout bout de champ dans l'application, il est bien entendu plus lourd d'un simple appel de méthode : ciblez les cas qui nécessitent vraiment un traitement semblable.

Gestion améliorée des contextes applicatifs

La notion de contexte a elle aussi été améliorée dans Seam, en effet par défaut dans un environnement web, on ne possède que 4 contextes pour déclarer les objets. Un contexte peut être vu comme un conteneur avec une durée de vie prédéfinie, ce qui permet de gérer le cycle de vie des objets. Seam rajoute alors trois contextes : Conversation, Process, et Business.

- **Stateless** qui est le contexte par défaut des composants EJB Stateless,
- **Request** (J2EE correspond à la requête HTTP) ou Event qui est le contexte par défaut des composants du type POJO ou JavaBean,
- **Page** (J2EE), le composant est accessible sur la durée de vie d'une page JSF,
- **Conversation** (Seam) qui est le contexte par défaut des composants du type EJB Stateful,
- **Session** (idem J2EE), le composant est accessible depuis la session HTTP,
- **Process** (Seam), contexte utilisé pour la gestion des processus (par exemple JBPM)
- **Buisiness** (Seam), contexte utilisé pour la gestion d'un long processus métier.
- **Application** (J2EE).



Les Contextes de Seam

Pour spécifier le contexte d'un objet dans Seam, il suffit de l'annoter : @Scope(ScopeType.CONVERSATION).

Les cycles de vie d'un contexte web (requête/session/application) et ceux des EJBs étant différents, mixer JSF et EJB 3.0 peut se révéler complexe.
Par exemple, l'accès à une relation lazy d'un EJB entité fourni par un EJB session produira une LazyLoadingException si cet EJB n'est plus managé. Il s'agit d'un problème récurrent de l'utilisation de solutions ORM telles que Hibernate ou JPA.
Seam permet de résoudre ce problème, grâce à une gestion élargie des contextes applicatifs.
Traditionnellement, les applications JEE gèrent leur état en plaçant des attributs de requête ou de session. Cette approche de la gestion étatique est la source de nombreux bugs et fuites de mémoire lorsque les applications ne parviennent pas à nettoyer les attributs de session, ou lorsque les données de session associée à différents processus entrent en collision dans une application multi-fenêtres.
Seam ajoute deux contextes aux contextes web standards : conversation, qui permet de stocker les objets d'un processus utilisateur, et business, qui permet la gestion de processus métiers.
La gestion des conversations de Seam permet de résoudre facilement ces problèmes : en effet, le contexte de persistance est placé en contexte conversation : les entités resteront ainsi managées, ce qui évite les problèmes de LazyLoading.
La gestion de la mémoire est elle aussi plus fine : à la fin de la conversation (déclarative ou sur timeout), tous les objets sont détruits.
Une conversation différente étant initialisée pour chaque processus, il n'y a plus de problèmes de collision de données lorsque l'utilisateur utilise plusieurs fenêtres sur l'application (par exemple, qu'il tente de réserver deux hôtels parallèlement).

Les conversations

Toute personne commençant à manipuler le framework Seam va se poser la même question, quelle est vraiment l'utilité de ce contexte ?

Actuellement, quelque soit le langage, (PHP, .NET, ou Java) pour gérer l'état conversationnel dans une application Web il n'y a qu'une seule solution : stocker les informations dans la session HTTP, ce qui entraîne de nombreux problèmes de purges des informations et de ré initialisation.

Seam introduit alors la notion de conversation longue, définie soit de manière implicite avec des EJB de type Stateful, soit de manière explicite en déclarant le contexte de la conversation. Le développeur peut ainsi choisir à quel moment commence la conversation et à quel moment elle se termine.

Le premier exemple cité dans la documentation de Seam pour convaincre à son utilité est très parlant pour comprendre l'intérêt, par exemple, de la conversation :
Prenez le cas d'un site web de réservation d'hôtel, si l'utilisateur a ouvert deux onglets sur le site et parcours les informations de deux hôtels différents, les informations visualisées par l'utilisateur sont stockées dans la session.
Maintenant admettons qu'il trouve un hôtel à sa convenance, il va effectuer la réservation.
S'il retourne sur son autre onglet, il risque probablement d'avoir des valeurs tout à fait fausses affichés à l'écran comme sur la majorité des sites web bloqués aux contextes de bases de stockages des objets (Request qui ne va maintenir aucune donnée, page qui va perdre les données dès que l'utilisateur change de page, et session qui sert un peu de décharge à tout ce qui doit être conservé plus longtemps que la page).

Seam résout donc ce type de problème grâce à la conversation, on peut désormais programmer un composant à état conservé en ayant la certitude de n'avoir aucun problème d'accès concurrent.

Les « nested conversation » permettent de créer une « sous conversation » : la conversation principale est mise en pause le temps que la nested conversation se termine. Cette dernière a accès à toutes les données présentes dans la conversation mère.
Une fois la nested conversation terminée, la conversation principale reprend là ou elle s'était arrêtée.

Ce mécanisme peut-être utilisé, par exemple, dans une application de réservation d'hôtels, pour choisir le type de chaque chambre durant un processus de réservation : il suffi de lancer une nested conversation affichant la page de choix des chambres. Une fois le type de chambre choisi, la réservation poursuivra son cours.

Pageflow

Seam intègre aussi jBPM pour permettre la gestion et de workflow, celle-ci a été simplifiée.
Un pageflow est en fait le déroulement programmé d'une séquence ordonnée de pages.
Grâce à la gestion fine apportée par les pageflow il est possible de contrôler les problèmes du bouton retour du navigateur. Ces problèmes de bouton retour sont récurrents dans le développement web, mais grâce à ce mécanisme il est possible de prévenir une partie de ceux-ci.

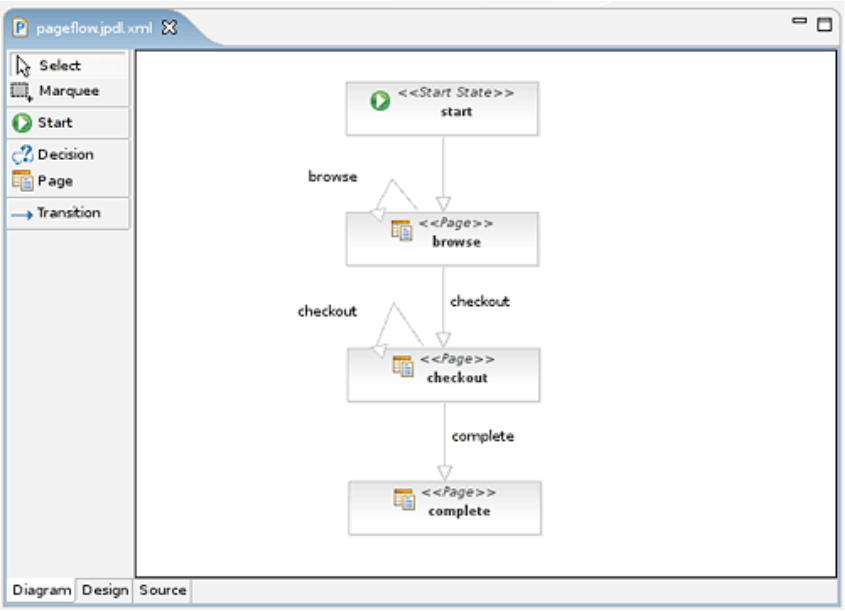
Démarrage d'un pageflow se fait simplement, comme pour une conversation, mais en précisant le pageflow qui sera utilisé.

```
@Begin(join=true, processDefinition="shopping")
public String doSearch() {
    …
}
```

Le pageflow est défini dans un fichier XML :

```
<pageflow-definition […] name="shopping">

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

   <page name="browse" view-id="/browse.xhtml" redirect="true">
      <transition name="browse" to="browse"/>
      <transition name="checkout" to="checkout"/>
   </page>

   <page name="checkout" view-id="/checkout.xhtml" redirect="true">
      <transition name="checkout" to="checkout"/>
      <transition name="complete" to="complete"/>
   </page>

   <page name="complete" view-id="/complete.xhtml" redirect="true">
      <end-conversation />
   </page>

</pageflow-definition>
```

Un éditeur graphique permet de créer simplement le pageflow, de manière visuelle :



Sur la page, des liens appelleront des méthodes des beans managés, qui retourneront simplement la prochaine étape du pageflow : par exemple :

```
<h:commandButton action="#{orderManager.completeOrder}"/>
```

La méthode checkout retournera la prochaine action : « complete »

Business process

Le lancement du process se fait comme pour une conversation.

```
@Begin(join=true, processDefinition="orderManagement")
public String order() {
    …
}
```

Imaginons que pour une commande de plus de 100€, une personne doive approuver.
Une méthode doit vérifier si la commande doit être approuvée ou pas :

```
@Name("orderApproval")
public class OrderApprovalDecision {
   @In float amount;
   public String getHowLargeIsOrder() {
      return amount > 100 ? "large order" : "small order";
   }
}
```

La définition du processus se fait en XML :

```
<process-definition name="orderManagement">
    <start-state>
        <transition to="decide"/>
    </start-state>

  <decision name="decide" expression="#{orderApproval.howLargeIsOrder}">
        <transition name="large order" to="approval"/>
        <transition name="small order" to="process"/>
    </decision>
```

```
    <task-node name="approval">
        <task name="approve">
           <assignment pooled-actors="reviewers" />
        </task>
        <transition name="approve" to="process"/>
        <transition name="reject" to="complete"/>
    </task-node>

  <task-node name="process">
        <task name="ship">
           <assignment pooled-actors="shippers,reviewers"/>
        </task>
        <transition name="shipped" to="complete">
            <action expression="#{afterShipping.log}"/>
        </transition>
    </task-node>

  <end-state name="complete"/>
</process-definition>
```

Tests Unitaires
Environnement d'exécution

Les tests d'intégration sont en général assez complexes à mettre en œuvre pour les EJBs : en effet, nous ne pouvons pas passer outre un conteneur d'EJB, mais nous ne souhaitons pas pour autant le déployer sur un serveur pour lancer les tests !
Il faut donc un moyen de reproduire le fonctionnement d'un serveur d'application, dans lequel l'application sera déployée, pour pouvoir lancer les tests automatisés.

Seam utilise un conteneur léger pour l'exécution des tests : JBoss Embedded.
Ce dernier permet de lancer un noyau JBoss 5, dans lequel l'application sera déployée et utilisée pour lancer les tests unitaires.
SeamTest utilise TestNG, mais il est possible d'utiliser d'autres framework (dans ce cas, il faudra prévoir un peu de développement).

SeamTest supporte les bases de données HSQL et MySQL. Il est possible d'utiliser d'autres bases de données, mais il faudra prévoir un peu de développement.

Voici un exemple de test :

```
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();
    }
    }...
```

Tests d'intégration avec une base de données
Si les tests nécessitent d'insérer des données avant de se lancer, et de les nettoyer à la fin du test, il est possible d'utiliser l'intégration de DBUnit de Seam, en héritant de DBUnitSeamTest.

Pour charger les données, il suffit d'écrire un fichier au format DBUnit :

```
<dataset>
    <ARTIST id="1" name="Metallica" />
    <DISC id="1" name="Kill 'Em All" artist_id="1" />
    <DISC id="2" name="Ride the Lightning" artist_id="1" />
    <DISC id="2" name="Master of Puppets" artist_id="1" />
</dataset>
```

Le chargement se fait en surchargeant la méthode `prepareDBUnitOperations()` :

```
protected void prepareDBUnitOperations() {
    beforeTestOperations.add(
        new DataSetOperation("my/datasets/BaseData.xml")
    );
}
```

Le nettoyage des données doit être fait en surchargeant la méthode `afterTestOperations()`.

Les mocks

Parfois, une application a besoin de ressources qui ne sont pas accessibles au moment des tests. Par exemple, une application peut avoir besoin de se connecter à un système de paiement.

```
@Name("paymentProcessor")
public class PaymentProcessor {
    public boolean processPayment(Payment payment) {
        ....
    }
}
```

Lors des tests, le serveur ne sera pas accessible, et nous ne pourrons de toutes manières pas payer pour chaque test. La solution est de remplacer ce composant par un Mock au moment des tests :

```
@Name("paymentProcessor")
@Install(precedence=MOCK)
public class MockPaymentProcessor extends PaymentProcessor {
    public boolean processPayment(Payment payment) {
        return true;
    }
}
```

Test des interactions utilisateur

SeamTest permet de simuler un environnement JSF, afin de permettre d'automatiser les tests reproduisant les interactions de l'utilisateur.

```
<html>
  <head>
    <title>Enregistrement d'un nouvel utilisateur</title>
  </head>
  <body>
    <f:view>
      <h:form>
        <table border="0">
          <tr>
            <td>Identifiant</td>
            <td><h:inputText value="#{user.username}"/></td>
          </tr>
          <tr>
            <td>Nom</td>
            <td><h:inputText value="#{user.name}"/></td>
          </tr>
          <tr>
            <td>Mot de passe</td>
            <td><h:inputSecret value="#{user.password}"/></td>
          </tr>
        </table>
        <h:messages/>
        <h:commandButton type="submit" value="Enregistrer"
                         action="#{register.register}"/>
      </h:form>
    </f:view>
  </body>
</html>
```

Afin de tester ce formulaire, nous allons simuler un cycle JSF à travers un test :

```
public class RegisterTest extends SeamTest {

    @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("lovthafew");
                assert getValue("#{user.name}").equals("Gavin King");
                assert getValue("#{user.password}").equals("secret");
            }
        }.run();
    }
    ...
}
```

N.B. : ils est aussi possible de tester les requêtes GET, en héritant de `SeamTest.NonFacesRequest`

Seam et la sécurité

Pour gérer les mécanismes d'authentification utilisateur Seam apporte un ensemble de composants basés sur JAAS et initialisés avec le framework au lancement de l'application.
Néanmoins deux composants sont principalement utilisés pour gérer les méthodes d'authentification : Identity et Credentials.

Il suffit en effet de créer deux entités pour gérer la persistance des rôles et des utilisateurs.
La liaison entre ces entités et le framework est assurée par un ensemble d'annotations très simples à mettre en place.
On commence par écrire l'Entité chargée de représenter l'utilisateur :

```
49  @Name("cmUser")
50  @Entity
51  @Table(name = "CmsUser", uniqueConstraints = {
52        @UniqueConstraint(columnNames = "userName"),
53        @UniqueConstraint(columnNames = "email") })
54  public class CmsUser implements Serializable {
55
56     /**
57      * Serial Uid.
58      */
59     private static final long serialVersionUID = 1L;
60
61
62     /** id. */
63     @Id @GeneratedValue
64     private Long id;
65
66     /** name. */
67     @NotNull
68     @UserPrincipal
69     private String userName;
70
71     /** password. */
72     @NotNull
73     @UserPassword(hash = "md5")
74     @Length(min = 4)
75     private String passwordHash;
76
77     /** email. */
78     @NotNull
79     @Email
80     private String email;
81
82
83     /** the user firstname. */
84     @NotNull @Length(max = 40)
85     @UserFirstName
86     private String firstName;
87
88     /** the user lastname. */
89     @NotNull @Length(max = 40)
90     @UserLastName
91     private String lastName;
92
93
94     /** the user roles. */
95     @UserRoles
96     @ManyToMany(targetEntity = CmsRole.class)
97     @JoinTable(name = "UserRoles",
98           joinColumns = @JoinColumn(name = "UserId"),
99           inverseJoinColumns = @JoinColumn(name = "RoleId"))
100    private Set<CmsRole> userRoles = new HashSet<CmsRole>();
```

Puis on définit les rôles :

```
35 @Name("cmsRole")
36 @Entity
37 @Table(name = "CmsRole", uniqueConstraints = @UniqueConstraint(columnNames = "rolename"))
38 public class CmsRole implements Serializable {
39
40     /**
41      * serial uid.
42      */
43     private static final long serialVersionUID = 1L;
44
45     /** id. */
46     @Id @GeneratedValue
47     private Long roleId;
48
49     /** the role rolename;. */
50     @RoleName
51     private String roleName;
52
```

Et on a plus qu'à écrire une méthode d'authentification dans un composant injectant Identity et Credentials :

```
52     @In
53     private Identity identity;
54
55     /**
56      * The Credentials.
57      */
58     @In
59     private Credentials credentials;
60
61     /**
62      *  authentification method.
63      * @return true if the user is registered.
64      */
65     public boolean authenticate() {
66         try {
67             log.info("authenticating {0}", credentials.getUsername());
68             Query query = entityManager.createQuery("from CmsUser where username = :username "
69                     + "and passwordHash = :password");
70             query.setParameter("username", credentials.getUsername());
71             query.setParameter("password", HashManager.instance().hash(credentials.getPassword()));
72
73             CmsUser user = (CmsUser) query.getSingleResult();
74
75             if (user.getUserRoles() != null) {
76                 for (CmsRole mr : user.getUserRoles()) {
77                     identity.addRole(mr.getRoleName());
78                 }
79             }
80             return true;
81         } catch (NoResultException ex) {
82             return false;
83         }
84     }
```

Enfin pour gérer les rôles et permissions au sein de l'application, des composants JSF ont été écris pour faire des tests coté vue, des annotations tel que @Restrict paramètrée par une EL pour restreindre l'utilisation d'une méthode, et il est aussi possible d'utiliser des règles Drools pour définir les règles de sécurité et les appeler via les annotations ou les composants JSF (comme le <s:hasPermission /> ou le <s:hasRole />.

Une fois les éléments précédents écrits, la méthode d'authentification est à déclarer dans le fichier component.xml pour la passer à Seam qui se chargera de manière transparente d'utiliser JAAS pour gérer la sécurité.

Pour sécuriser les pages il suffit de spécifier dans les pages.xml, que le login est requis.

Comme vous pouvez le voir c'est très simple. Un mécanisme pour générer des Captcha est aussi présent dans le framework si nécessaire.
Bien d'autres mécanismes sont en place dans le Framework pour gérer toute sorte de situation ayant besoin d'être sécurisée, je vous invite à regarder la documentation officielle très bien écrite à ce sujet.

Seam Remoting

Alternativement aux composants Ajax de richfaces, ou d'icefaces vous pouvez utiliser Seam Remoting. Ce mécanisme permet d'invoquer des méthodes distantes coté serveur via javascript, comme si elles étaient locales au browser. L'interaction avec l'objet coté serveur est réalisée via des requêtes Ajax mais les requêtes sont encapsulées dans des objets proxy générés dynamiquement par javascript, donc il n'est jamais nécessaire d'interagir avec l'objet XMLHttpRequest.

Via Seam Remoting, le client et le serveur sont fusionnés, en établissant un lien continu entre les opérations locales et distantes.
L'échange ressemble à un appel de procédure distante tel que RPC ou java RMI ou SOAP, excepté que le javascript remoting est beaucoup plus léger et transparent pour le développeur.
Pour qu'une méthode soit utilisable via ce mécanisme il suffit de l'annoter `@WebRemote`, celle-ci pouvant être paramètrée.

Coté javascript pour récupérer une référence à un objet il suffit de faire :
```
var myComponent = Seam.Component.getInstance(«myComponent»);
```
puis pour appeler la méthode :
```
myComponent.myMethode();
```

Les possibilités d'intégration

Les possibilités d'intégration de Seam sont très nombreuses, Seam est prêt à communiquer avec de nombreux autres frameworks aussi bien graphique que métiers.
Il est possible d'utiliser Wicket, Tapestry, GWT, ou Flex en lieu et place de JSF;

On l'a vu coté service on peut utiliser EJB, mais aussi les composants Seam, des Pojo classiques voir même des composant Spring.
Spring est parfaitement intégrable, et on peut injecter des composant Seam dans des Spring Beans et inversement.

Seam a intégré son propre mécanisme d'URL rewritting, mais se marie très bien avec Tuckey aussi.

Seam intègre aussi JExcelApi pour manipuler des documents excels.
YARFRAW a été intégrée pour ajouter le support du RSS.
iText permet la génération simplifiée de PDF et est présent dans le framework.

Jboss-Seam-Mail permet l'écriture de mail via des contrôles JSF, ce qui rend leur mise en page dynamique très simple.

De nombreuses simplifications sont disponibles pour l'écritures de web services.

JBoss Seam Mail

Envoi de mails

Seam utilise facelets pour l'envoi de mail. Il n'est donc pas nécessaire de connaître d'autre langage de templating.

```
<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="noreply@example.com" address="noreply@example.com"/>
  <m:to name="#{person.firstname} #{person.lastname}">#{person.address}</m:to>
  <m:subject>Votre inscription</m:subject>
  <m:body>
    <p><h:outputText value="Bonjour #{person.firstname}"/>,</p>
    <p>Nous sommes heureux de confirmer votre inscription.</p>
  </m:body>
</m:message>
```

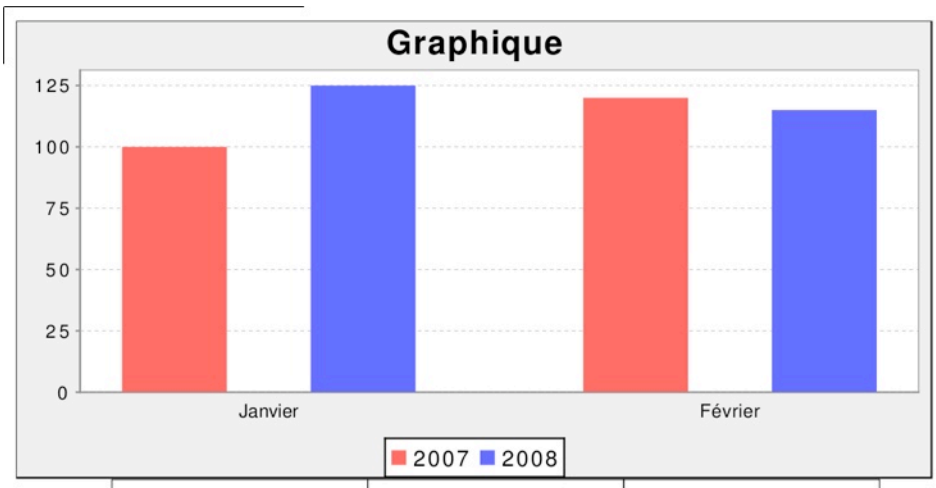
L'envoi du mail se fait simplement :

```
@In(create=true)
private Renderer renderer;

public void send() {
    try {
        renderer.render("/registrationEmail.xhtml");
    } catch (Exception e) {
        facesMessages.add("Erreur lors de l'envoi du mail d'enregistrement");
    }
}
```

Réception de mails

Une application peut facilement recevoir des mails, grâce à un MDB (Message Driven Bean) :

```
@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());
     }
}
```



Année	Mois	Valeur
2008	100	120
2009	125	115

Création de PDF

Les PDFs sont générés grâce à iText, là aussi grâce à facelets :

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
            xmlns:ui="http://java.sun.com/jsf/facelets"
            title="Hello">
  <p:barchart title="Graphique" legend="true" width="500" height="250">
    <p:series key="2007">
      <p:data columnKey="0" key="Janvier" value="100" />
      <p:data columnKey="1" key="Février" value="120" />
    </p:series>
    <p:series key="2008">
      <p:data columnKey="0" key="Janvier" value="125" />
      <p:data columnKey="1" key="Février" value="115" />
    </p:series>
  </p:barchart>
  <p:paragraph/>
  <p:table columns="3" headerRows="1">
    <p:cell>Année</p:cell>
    <p:cell>Mois</p:cell>
    <p:cell>Valeur</p:cell>
    <p:cell>2008</p:cell>
    <p:cell>100</p:cell>
    <p:cell>120</p:cell>
    <p:cell>2009</p:cell>
    <p:cell>125</p:cell>
    <p:cell>115</p:cell>
  </p:table>
</p:document>
```

Tableau Excel

Seam permet aussi de générer des feuilles de calcul Excel, toujours avec Facelet.

```
<e:workbook>
  <e:worksheet>
    <e:cell value="Hello World" row="0" column="0"/>
  </e:worksheet>
<e:workbook>
```

Il est aussi possible d'exporter facilement le contenu d'un tableau, directement dans une page :

```
<h:form id="theForm">
  <h:dataTable id="theDataTable" value="#{personList.personList}" var="person">
    ...
  </h:dataTable>
</h:form>
...
<h:commandLink value="Export"
               action="#{excelExporter.export('theForm:theDataTable')}" />
```

Hibernate Search

Moteurs de recherches textuels

Les moteurs de recherches textuels, tels que Apache Lucene, sont des technologies très performantes et efficaces.
Hibernate Search se base sur Apache Lucene pour indexer les données de la base de données et maintenir l'index synchronisé avec cette dernière.

Seam intègre Hibernate Search afin de simplifier l'intégration de moteurs de recherche textuel dans une application.

L'indexation se fait directement sur les EJBs, via des annotations :

```
@Entity
@Indexed
public class Book {

  @Id
  @GeneratedValue
  @DocumentId
  private Integer id;

 @Field(index=Index.TOKENIZED, store=Store.NO)
  private String title;

 @Field(index=Index.TOKENIZED, store=Store.NO)
  private String description;

 public Book() {
  }

 // standard getters/setters follow here
  ...
}
```

Il suffit ensuite d'injecter un FullTextEntityManager afin d'effectuer des recherches « full text » :

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch {

    @In
    FullTextEntityManager entityManager;

    public void search(String searchString) {

        // Définit des niveau de priorité par champs
        Map<String,Float> boostPerField = new HashMap<String,Float>();
        boostPerField.put("title", 4f);
        boostPerField.put("description", 1f);

        // Crée le parseur
        QueryParser parser = new MultiFieldQueryParser(
                new String[] {"title", "description"},
                new StandardAnalyzer(), boostPerField);
        parser.setAllowLeadingWildcard(true);

        // Crée la requête et retrouve les résultats
        org.apache.lucene.search.Query luceneQuery = parser.parse(searchString);
        javax.persistence.Query query =
                entityManager.createFullTextQuery(luceneQuery, Book.class);
        searchResults = query
                .setMaxResults(pageSize + 1)
                .setFirstResult(pageSize * currentPage)
                .getResultList();
    }
    ...
}
```

Ce code permet une recherche textuelle, avec une priorité sur le nom du livre.
Le format de requête est celui de Lucene, par exemple, pour retrouver tous les livres dont le titre ou la description contiennent « java » mais pas « php », la requête sera : "+java -php".
Les résultats contenant java dans le titre apparaîtront avant ceux contenant java dans la description.

Seam dans l'avenir

Si Seam est une évolution, les Web beans évoqués plus tôt : JSR 299 sont une révolution.
La JSR 299 est un effort collaboratif influencé par des développeurs de Seam et de Google.
Le but des Web Beans est de standardiser des composants sécurisés qui pourront traverser les couches de l'application.
Le but est d'unifier le tiers web et le tiers EJB, ce qui simplifiera grandement le développement java pour le web.
L'idée pour atteindre cet objectif est de fournir des composants avec un couplage faible, mais un typage fort. Ceci rend un système flexible au changement.
Les WebBeans permettront aussi de spécifier le type de déploiement des composants, par annotations. Par exemple @Production, ou d'autres annotations que vous pourrez définir.
C'est aussi ce type d'annotations qui permettra de déclarer un objet comme Web Beans.

Seam s'articulera aussi autour de JSF 2 dont il est un des participant les plus actifs.