D. Galland

L'architecture Java EE

L'architecture ou plate-forme Java EE définit un ensemble de spécifications pour développer et exécuter des applications d'entreprise multi-niveaux et réparties. A ce titre elle comporte :

- Le langage Java et la machine virtuelle pour l'exécution des applications.
- Un ensemble d'API de toutes natures à disposition du développeur
- Des spécifications normalisatrices qui assurent l'interopérabilité des composants

Ces trois éléments, le langage avec la machine virtuelle, les API et les spécifications assurent la portabilité des applications quel que soit l'environnement matériel et système d'exploitation ainsi que l'interopérabilité des composants. On entend par interopérabilité le fait que tout composant JEE pourra s'exécuter et communiquer avec d'autres composants dans tout environnement JEE conforme donc aux spécifications.

Anciennement J2EE, l'architecture Java EE, aujourd'hui Java EE 7, a évolué au travers d'un processus continuel le JCP Java Community Process qui définit des JSR Java Spécification Request. Le JCP regroupe les parties intéressées sous l'égide de grandes sociétés, Sun créateur du langage Java, IBM et autres et bien sûr ORACLE repreneur de SUN et détenteur maintenant des droits sur Java. La quasi totalité des composants Java EE ont des licences Open Source mais ils peuvent être repris dans des produits commerciaux proposés par ces différentes sociétés.

Un processus de certification permet de garantir la conformité d'un composant ou produit à une spécification JEE. Pour une spécification donnée il existe généralement une implémentation de référence RI mais il peut en exister des alternatives, équivalentes bien sûr puisque conformes à la spécification (ce qui n'empêche pas bien sûr un produit d'être « meilleur » qu'un autre dans son implémentation)

L'architecture JEE 7 comporte une trentaine de spécifications, ce document se limitera aux spécifications relatives à la réalisation d'applications d'entreprise multi-niveaux sur support WEB. A ce titre nous distinguerons les niveaux (tiers) suivants :

- La couche de présentation
- La couche métier
- La couche données persistantes

La couche « Web-Tier »

Nous nous limiterons à la couche de présentation client « Web » dans laquelle l'utilisateur accède à l'application au travers d'un navigateur. La spécification de base pour l'exécution du traitement coté serveur est la spécification Servlet 3.0. Les composants servlet et associés s'exécutent dans un serveur de traitement Web ou conteneur Web (l'implémentation de référence étant le serveur TOMCAT). L'ancienne spécification JSP Java Server Pages maintenant JSP 2.1 peut encore être utilisée, elle est complétée par la spécification d'une bibliothèque de balises JSTL JSP Standard Tag Library et la définition d'un langage d'expression EL permettent d'accéder dans les pages JSP à des objets Java visibles dans le contexte. Ce langage d'expression est également utilisé dans JSF.

En restant dans le même contexte Servlet mais en abandonnant les pages JSP, la spécification JSF Java Server Faces permet une conception MVC. Les composants UI de la vue sont décrit par des balises dans des pages xhtml. Le lien avec les classes Java représentant le contrôleur étant assuré par l'expression language EL. JSF 2.0 ajoute également le support d'AJAX. L'implémentation de référence de JSF 2.0 est le produit Mojarra (Oracle).

La couche données persistantes

En premier lieu bien sûr l'API JDBC permet un accès normalisé et uniforme à toutes les bases de données SQL du marché. Elle permet de soumettre à la base de données des statements SQL.

A un plus haut niveau, il est maintenant usuel d'accéder à la base de donnée dans une optique orientée objet en utilisant un framework ORM (Object Relational Mapping).

L'historique des framework ORM en Java a été assez chaotique, les anciennes spécifications, Java Data Object JDO et beans d'entité EJB 2.0, s'étant avérées malcommodes tandis que des produits tiers comme Hibernate faisaient au contraire l'objet d'un large consensus. Ce sont les idées de ces produits qui se retrouvent aujourd'hui dans la spécification JPA Java Persistance API, l'implémentation de référence est Oracle Toplink devenu Eclipse Link, mais Hibernate est également conforme à JPA. La spécification JPA 2.0 définit les beans entités persistants dans la spécification EJB 3.1.

L'API JTA Java Transaction API prend en charge l'aspect transactionnel.

La couche métier « business-tier »

Comme son nom l'indique, un composant métier réalise un traitement applicatif en liaison avec la base de donnée, c'est la partie modèle dans la terminologie MVC. Deux cas de figure peuvent se présenter :

- Dans les cas les plus simples même si l'on adopte une conception modulaire par classes avec les interfaces ad-hoc le composant métier est directement lié au contrôleur de la couche Web, il s'exécute au sein du même conteneur Web et accède à la base de donnée avec JPA.
- Dans une conception multi-niveaux complète, les composants métier sont totalement indépendants du contrôleur de la couche Web, ce sont des beans de session (stateless ou statefull) qui s'exécutent dans un serveur EJB de traitement transactionnel en exposant leur interface soit en local soit en tant qu'objets distants. Au prix d'une plus grande complexité on bénéficiera alors des possibilité offertes par ces serveurs de traitement JEE.

L'implémentation de référence d'un serveur EJB 3.1 est le produit Glassfish (Oracle) mais il existe des alternatives, les version récentes de WebSphere chez IBM ou le Jboss Application Server.

Java EE 7

Par rapport aux versions antérieures JEE 7 a été simplifié dans son utilisation et enrichi de nouvelles API, il se veut un framework léger accélérant le développement.

Configuration par annotation et par convention

Dans les versions JEE antérieures la configuration (par exemple celle d'une servlet, d'un bean entité, etc...) était décrite dans des fichiers de configuration XML. JEE 7 fait maintenant un usage intensif des annotations Java, les informations de configuration étant au plus près des classes, méthodes ou attributs concernés.

De plus un grand nombre d'informations de configuration sont optionnelles, les valeurs prises par défaut étant satisfaisantes dans la plupart des cas (par exemple dans JPA le nom de la table est par défaut celui de la classe entité).

Contextes et Injection de dépendances CDI 1.0

L'injection de dépendance et la gestion des contextes consiste à transférer au conteneur la responsabilité de la création des objets requis par d'autres objets ainsi que celle de la gestion de leur durée de vie, le tout contrôlé par les annotations adéquates. Ceci va permettre une intégration plus facile entre les différents niveaux et avec les services du conteneur.

Bean Validation 1.0

Cette API permet de spécifier des contraintes de validité directement au niveau d'un bean. Ces contraintes sont intégrées avec les autres API et pourront par exemple être utilisée par JSF pour vérifier les saisies ou bien par JPA au moment de l'écriture dans la base de données.

Web Services

Deux API permettent d'exposer des beans comme Web Services

- Java API for XML-based Web Services (JAX-WS) 2.2 pour les services SOAP
- Java API for RESTful Web Services (JAX-RS) 1.1 pour les services REST

L'API Java Architecture for XML Binding (JAXB) 2.2 permet la sérialisation/dé sérialisation de beans en XML

La couche de persistance et l'API JPA

Les beans entitées et les annotations de persistance

Un bean entité est un simple POJO avec des annotations qui précisent le mapping entre ce bean et la table de la base de données.

Les seules annotations indispensable sont l'annotation `@Entity` et l'annotation `@Id` qui indique la clé identité généralement générée automatiquement. Pour faciliter on peut définir une première classe avec cette clé identité :

```
package entities;
import javax.persistence.*;
@MappedSuperclass
public class Persistent {
  private Long id;

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

Puis par exemple une classe `Client` :

```
@Entity
@NamedQuery(name = "Client.findAll", query = "Select c From Client c")
public class Client extends Persistent {
  private String login;
  private String password;

  public String getLogin() {
    return login;
  }
  public void setLogin(String login) {
    this.login = login;
  }
  public String getPassword() {
    return password;
  }
  public void setPassword(String password) {
    this.password = password;
  }
}
```

Par défaut, le nom de la table est celui de la classe et tous les attributs du bean sont persistants, tout ceci pouvant être contrôlé plus finement par des annotations. L'annotation `@NamedQuery` permet de spécifier des requêtes pour cette entité.

Les annotations concernant les attributs peuvent se mettre soit au niveau de la déclaration, soit au niveau du getter de cet attribut.

Fichier de configuration JPA

fichier src/META-INF/persistence.xml pour JAVA SE

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" 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_2_0.xsd">
  <persistence-unit name="BookStore" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>entities.Persistent</class>
    <class>entities.Client</class>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost/bookstore;create=true" />
      <property name="javax.persistence.jdbc.user" value="bookstore" />
      <property name="javax.persistence.jdbc.password" value="bookstore" />
    </properties>
  </persistence-unit>
</persistence>
```

Ce fichier définit une persistence-unit qui se connecte directement au serveur de base de données par JDBC , on indique comme d'habitude les paramètres de connexion. Elle est utilisable dans une application Java SE stand-alone ou au sein d'une application web dans Tomcat par exemple.

Fichier src/META-INF/persistence.xml pour une application JEE

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" 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_2_0.xsd">
  <persistence-unit name="BookStore" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/BookStore</jta-data-source>
    <class>entities.Client</class>
  </persistence-unit>
</persistence>
```

Ce fichier définit une persitence-unit géré par le conteneur EJB

Dans les deux cas le provider définit l'implémentation sous-jacente, ici Eclipse Link (anciennement Toplink) mais on pourrait aussi bien utiliser Hibernate.

EntityManager

Les entités persistantes sont gérées par un `EntityManager`. La façon dont on obtient un `EntityManager` diffère suivant que l'on se situe au sein d'un conteneur JEE ou non.

En dehors d'un conteneur JEE, l'`EntityManager` est entièrement géré par l'application qui est aussi responsable de la démarcation des transactions On l'obtient avec une `EntityManagerFactory`, en spécifiant le nom de la persistence-unit Soit par exemple une application Java SE :

```
public class CreateTables {
  public static void main(String[] args) {
    Map<String, String> properties = new HashMap<String, String>();
    properties.put("eclipselink.ddl-generation", "drop-and-create-tables");
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("BookStore", properties);
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    Client client = new Client();
    client.setLogin("galland");
    client.setPassword("dauphine");
    em.persist(client);
    em.getTransaction().commit();
    em.close() ;
  }
}
```

Dans cet exemple on demande de plus à la factory de créer les tables à partir des classes.

Au sein d'un EJB s'exécutant dans un conteneur JEE on utilisera un « Container Managed EntityManager », le contexte de persistance et la transaction JTA sont alors crées, propagés et détruits automatiquement par le conteneur. On obtient l'`EntityManager` par injection (unitName est facultatif s'il n'y a qu'une seule persitence-unit), soit par exemple :

```
@Stateless
public class ClientManager {
  @PersistenceContext(unitName = "BookStore")
  private EntityManager em;

  public ClientManager() {
  }

  public Client login(String login, String password) {
    Client client = null;
    try {
      client = (Client) em.createQuery("select c from Client c where c.login=:login and c.password=:password")
          .setParameter("login", login).setParameter("password", password).getSingleResult();
    } catch (NoResultException e) {
    }
    return client;
  }
}
```

L'`EntityManager` permet les opération de base CRUD (Create, Read, Update, Delete) sur les entités ainsi que les requêtes exprimées en JPQL (Java Persistence Query Language)

Création d'une entité :

```
em.persist(client)
```

Retrouver une entité par la clé identité :

```
Client client = em.find(Client.class, clientId) ;
```

Supprimer une entité :

```
em.remove(client) ;
```

Exemples de requêtes JPQL

```
List l = em.createQuery("select c from Client c").getResultList();
```

```
client = (Client) em.createQuery("select c from Client c where c.login=:login and c.password=:password")
          .setParameter("login", login).setParameter("password", password).getSingleResult();
```

Les objets obtenus à partir d'un `EntityManager` sont dits attachés, leur état sera automatiquement synchronisé avec la base au commit de la transaction (il n'y a donc pas de méthode update explicite). Après cela ces objets sont dits détachés et n'ont plus de liens avec la base de données.

Enfin un objet conservé détaché peut être ré-attaché à l'`EntityManager` (ce qui générera un update ultérieur).

```
em.merge(client) ;
```

La gestion des relations

L'`EntityManager` peut gérer automatiquement les relations 1:1, 1:n, n:n bidirectionnelles ou non entre entités. Ces relations sont définies par des annotations dans les classes.

Exemple d'une relation `1:n` bidirectionnelle entre `Order` et `OrderItem`

Dans la classe `Order` :

```
private List<OrderItem> items = new ArrayList<OrderItem>();
...
@OneToMany(mappedBy="order", cascade=CascadeType.ALL, fetch=FetchType.EAGER)
 public List<OrderItem> getItems() {
  return items;
}
```

Dans la classe `OrderItem` :

```
private Order order;
...
@ManyToOne
 public Order getOrder() {
  return order;
}
```

Exemple d'une relation `n:n` bidirectionnelle entre `Book` et `Author`

Dans la classe `Book` :

```
@ManyToMany
public List<Author> getAuthors() {
  return this.authors;
}
```

Dans la classe `Author` :

```
    @ManyToMany(mappedBy="authors")
    public List<Book> getBooks() {
        return this.books;
    }
```

Dans ces deux exemples le coté avec l'attribut `mappedBy` est le coté inverse ou esclave de la relation, c'est par l'autre coté que devront être établies les relations avant sauvegarde dans la base (en fixant l'`Order` d'un `OrderItem` ou les `Author` d'un `Book`).

Des attributs supplémentaires permettent de spécifier le comportement `CASCADE` des relations, ici par exemple on spécifie qu'une opération create ou delete sur un `Order` entraîne la création ou la destruction de `OrderItem` associés.

Avec ces annotations l'`EntityManager` permet de suivre automatiquement les associations, ainsi par exemple si on a un `Order` il n'est point besoin de faire une query pour obtenir les `OrderItem` associés, il suffit d'écrire :

```
order.getItems()
```

Cependant pour ces associations il y a deux stratégies de chargement :

- Immédiat ou EAGER les entités associées sont lues immédiatement après la lecture de l'entité qui les référence
- Différé ou LAZY les entités associées sont lues uniquement lorsqu'elles sont accédées.

Par défaut les associations qui retournent des collections sont LAZY, celle retournant un seul objet sont EAGER, on peut changer ce comportement avec une annotation spécifique comme dans l'exemple ci-dessus pour les `OrderItem` d'un `Order`.

Questions à se poser sur les relations et le LAZY loading

L'accès LAZY à partir d'une entité détachée pose un problème qui n'est pas traité de la même façon par toutes les implémentations JPA.

LAZY ou non, on pourrait conseiller de traiter automatiquement les relations qui retournent des `List` quand la liaison est vraiment forte et que la taille de cette liste est limitée, par exemple cela paraît naturel pour les `OrderItem` d'un `Order` et peut être moins pour les `Order` d'un `Client` ? On peut bien sûr ne pas définir une relation et faire une requête explicite pour avoir par exemple les `Order` d'un `Client`.

Dans une conception stricte multi niveaux les entités sont traitées dans le tiers métier, si on les utilise dans contrôleur et la vue il faut considérer qu'elle sont alors détachées et donc a priori ne pas utiliser le LAZY.

ATTENTION : Bien comprendre la progression dans le polycopié ci-dessous :

1. Application WEB/JSF uniquement dans un contexte Java SE
2. Dans un serveur JEE complet comme Glassfish utilisation de CDI
3. Utilisation des EJB pour le tiers métier

Le cas 3 correspond à l'utilisation de toutes les possibilités de l'architecture JEE

Application Web dans un contexte Java SE

Remarque importante :

Concernant une application Web JEE le premier choix que vous devez faire concerne l'organisation en niveaux et le choix du conteneur :

- Vous pouvez développer une application Web dans un contexte Java SE avec un simple conteneur WEB comme Tomcat par exemple
- Vous pouvez développer une application dans le contexte d'un serveur JEE complet comme Glassfish.

Dans le premier cas vous devrez gérer vous même l'`EntityManager` et les transactions, dans le second cas vous pourrez pour la couche métier utiliser des EJB et bénéficier de l'injection de dépendance CDI. Noter que dans le premier cas vous pourrez utiliser un framework reconnu comme Spring qui vous offrirait des possibilités équivalente à JEE sans être un standard JEE.

Nous commencerons dans un premier temps par décrire une application Web dans un contexte Java SE

Le tier Web en architecture JEE

Dans l'architecture JEE, le composant de traitement de base coté serveur dans une application Web est la servlet. La servlet reçoit la requête HTML et ses paramètres et retourne au navigateur un contenu, le plus souvent une page HTML, au navigateur. Ainsi le cycle de base du traitement comporte :

- Vérification et conversion des paramètres de la requête (contrôleur)
- Exécution des traitements métier en liaison avec la base de données (contrôleur puis modèle)
- Préparation de la vue, la page HTML qui doit être retournée au navigateur.

La servlet s'exécute dans un conteneur de traitement Web comme le serveur TOMCAT.

Dans une optique plus orientée présentation une page JSP est une page qui contient du HTML et du code Java pour les traitements et pour la génération des éléments dynamiques de la vue. La page JSP est compilé en une servlet.

Tout en conservant JSP on peut adopter une méthodologie plus conforme aux principes d'une conception MVC, commencer le traitement dans une servlet contrôleur puis enchaîner sur la vue en JSP. Le langage d'expression EL étant utilisé pour dans la vue accéder à des objets Java dans le scope (application, request ou session). De plus on peut utiliser dans la vue des balises de haut niveau, (à contenu algorithmique comme une itération ou à contenu visuel), comme celles de la JSTL. On peut ainsi réaliser des pages JSP avec 0% Java.

Les framework de développement MVC systématisent cette approche, une servlet contrôleur reçoit la requête, vérifie et convertit les paramètres, déclenche les traitements et enchaîne sur le rendu de la vue.

Il existe de nombreux framework de développement MVC pour des applications Web en Java, par exemple le très ancien Struts, Spring MVC etc.... Le framework JSF, Java Server Faces, est l'un des plus récents, c'est le framework « officiel » en architecture JEE. Il est maintenant dans sa version JSF 2.2 plus aboutie et qui apporte des évolutions majeures par rapport aux versions antérieures. L'implémentation la plus usuelle étant mojarra (Sun/Oracle).

Présentation de JSF

Les développeurs d'applications graphiques non WEB, par exemple des applications Java utilisant Swing, sont habitués à la notion de composant graphique d'interface utilisateur, par exemple un champ de saisie. Ils peuvent créer ces composants, les imbriquer les uns dans les autres, formant ainsi une arborescence de composants, et manipuler leurs propriétés, éventuellement au sein d'une environnement de développement « visuel ».

La « vue » présentée à l'utilisateur dans son navigateur sera donc rendue à partir d'un arbre de composants d'interface (`UIComponent`) imbriqués. Leurs propriétés et les événements qu'ils vont générer seront reliés à des objets Java, bean de backing ou « Managed beans » jouant le rôle de contrôleur et donc points d'entrée vers le modèle.

Le cycle requête-réponse JSF est le suivant:

- Quand la requête est soumise, les valeurs saisies sont validées, converties et transférées dans les beans de backing
- Les actions des composants « action » (boutons ou liens) sont reliées à des méthodes action des beans de backing
- Le résultat de l'action détermine la prochaine vue (navigation)
- Lors du rendu de la vue les valeurs affichées des composants d'interface sont obtenus à partir des beans de backing

On devra donc réaliser et paramétrer:

- Les pages décrivant par des balises les composants d'interface.
- Les classes Java des « backing beans » ou « managed beans » dont les attributs et les méthodes seront associés aux propriétés et aux actions des composants d'interface.
- Un fichier de configuration usuellement appelé `WEB-INF/faces-config.xml` contenant quelques déclarations et la description de la navigation dans l'application.
- Le tout s'exécutant bien entendu dans un serveur d'application Web comme Tomcat. Le fichier de configuration de Tomcat `web.xml` doit comporter la définition de la servlet Faces, les URL qu'elle traite ainsi que divers autres paramètres.

En JEE6 ces deux fichiers XML se réduisent grâce à l'usage des annotations.

Description et rendu de la vue, Facelets

A partir du moment où l'on fait du JSF, il est finalement peu adéquat de décrire la vue avec une page JSP sans aucun code Java (compilée en une servlet !). Il est donc très usuel d'utiliser, en complément de JSF « Facelets » qui fournit un renderer qui génère la vue directement à partir de sa description dans un fichier xhtml. De plus Facelets facilite la composition de pages, l'inclusion dynamique ainsi que la construction de composants par assemblage. En JSF 2.0 Facelets est la méthode de rendu par défaut.

JSF notions de base, Un exemple JSF

Template de mise en page

Les pages de l'application seront toutes conformes à un modèle `template.xhtml` qui définit une mise en page, ici du type top, left, content.

Fichier template.xhtml

```
<!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:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
<f:view contentType="text/html">
  <h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Exemple</title>
    <h:outputStylesheet library="css" name="stylesheet.css" />
  </h:head>
  <h:body>
    <div id="top">
      <ui:insert name="top" />
    </div>
    <div id="left">
      <ui:insert name="left" />
    </div>
    <div id="content">
      <ui:insert name="content" />
    </div>
  </h:body>
</f:view>
</html>
```

Le style de ces `div` provient d'une feuille de style css. A noter que JSF définit un répertoire conventionnel `resources` pour contenir images, css etc...

Une page de login

Soit une page de login conforme à ce template :

Fichier login .xhtml

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html" template="template.xhtml">
  <ui:define name="top">
    <ui:include src="top.xhtml" />
  </ui:define>
  <ui:define name="left">
    <ui:include src="left.xhtml" />
  </ui:define>
  <ui:define name="content">
    <h:form id="contentForm">
      <h:panelGrid columns="3">
        <h:outputText value="#{strings['Client.login']}" />
        <h:inputText id="login" value="#{clientController.login}" required="true" />
        <h:message for="login"></h:message>
        <h:outputText value="#{strings['Client.password']}" />
        <h:inputSecret id="password" value="#{clientController.password}" required="true" />
        <h:message for="password"></h:message>
      </h:panelGrid>
      <h:commandButton action="#{clientController.doLogin}" value="#{strings['button.login']}"/>
      <h:messages globalOnly="true"/>
    </h:form>
  </ui:define>
</ui:composition>
```

Noter que cette page de `login.xhtml` est accessible par l'URL `login.jsf` ou bien `faces/login.xhtml` selon le mapping défini pour la servlet Faces dans `web.xml`.

Elle inclut les fragments top et left et définit le content.

Ne pas oublier que tout les composants interactifs doivent être inclus au sein d'une `h:form`

On met en page dans un tableau `h:panelGrid` à trois colonnes un label, un champ de saisie et un message d'erreur associé., les messages d'erreur globaux (non associés à un composant) sont affiché en dessous.

Les labels sont obtenus à partir d'un bundle internationalisable messages comme expliqué plus loin.

Les champs de saisie sont liés à des attributs du bean de backing `clientController`.

L'action du bouton est liée à la méthode `doLogin` du même bean de backing.

Un ManagedBean

Fichier ClientController.java

```
@ManagedBean
@SessionScoped
public class ClientController {
  private static Logger log = LoggerFactory.getLogger(ClientController.class);

  private String login;
  private String password;
  private Client client;

  getters et setters usuels

   public String doLogin() {
    EntityManager em = JpaFilter.getEntityManager();
    try {
      client = (Client) em.createQuery("select c from Client c where c.login=:login and c.password=:password")
          .setParameter("login", login).setParameter("password", password).getSingleResult();
    } catch (NoResultException e) {
      FacesContext facesContext = FacesContext.getCurrentInstance();
      ResourceBundle bundle = (ResourceBundle) facesContext.getApplication()
          .getResourceBundle(facesContext, "messages");
      String summary = bundle.getString("clientNotFound");
      log.debug(summary);
      FacesMessage msg = new FacesMessage(FacesMessage.SEVERITY_ERROR, summary, summary);
      facesContext.addMessage(null, msg);
      return null;

    }
    return "welcome"; // Une règle de navigation
  }

  public String doLogout() {
    client = null;
    return "login"; // Une règle de navigation
  }
}
```

L'annotation `@ManagedBean` indique un bean de backing dont le nom par défaut est celui de la classe, premier caractère en minuscule.

Quand on arrivera dans la méthode `doLogin`, JSF aura déjà transféré les valeurs validées des champs de saisie dans les attributs `login` et `password`. Les contraintes de validation (`required= "true"`) auront déjà été validées et un message d'erreur affiché dans le cas contraire.

On effectue une query pour vérifier le login, si oui le client est conservé dans le bean, ce bean étant de scope session `@SessionScoped`, il pourra être utilisé dans toutes les pages de l'application pour afficher les informations du client, comme par exemple `#{clientController.client.login}`

Messages d'erreur

Nous avons déjà indiqué que dans un premier temps JSF traite déjà automatiquement les messages d'erreurs associés aux contraintes de validation.

Dans les méthodes action on peut générer des message d'erreur qui seront affichés par le même mécanisme, ces message d'erreurs sont soit globaux, soit associés à un composant.

Dans l'exemple ci-dessus, si le login échoue, on reste sur la vue en construisant un message d'erreur global qui sera affiché par la balise `h:messages` ce message est bien entendu recherché dans un bundle internationalisé. Notez que l'on peut raffiner en distinguant un message abrégé ou détaillé. On peut de plus utiliser le package `java.text.Format` pour des paramètres substituables.

Construire un message d'erreur associé à un composant est un peu plus compliqué, en effet il faut retrouver le composant dans la vue, on peut le retrouver par son id mais ce n'est pas simple car JSF ajoute à l'`id` du composant l'`id` de la forme contenant le composant, ainsi, pour un composant `id= "login "` contenu dans une forme `id="contentForm"`, l'id du composant sera `contentForm:login`. La méthode la plus sûre est d'utiliser un binding permettant d'associer le composant dans la vue avec un composant `UIInput` dans la classe Java :

```
<h:inputText id="login" value="#{clientController.login}" required="true" binding="#{clientController.uiLogin}"/>
```

et dans `ClientController`

```
private UIInput UiLogin;
getter et setter
```

et pour générer le message associé au composant :

```
facesContext.addMessage(UiLogin.getClientId(facesContext), msg);
```

Cet attribut `binding` peut servir dans d'autre cas chaque fois que l'on veut référencer dans le code Java un composant de la vue.

Internationalisation

En cas d'erreur de login, on génère un message d'erreur global de Faces. Les bundles sont définis dans le fichier `faces-config.xml`

```
  <application>
  <locale-config>
      <supported-locale>en_US</supported-locale>
      <supported-locale>fr_FR</supported-locale>
    </locale-config>
    <resource-bundle>
      <base-name>resources.strings</base-name>
      <var>strings</var>
    </resource-bundle>
    <resource-bundle>
      <base-name>resources.messages</base-name>
      <var>messages</var>
    </resource-bundle>
    <message-bundle>resources.messages</message-bundle>
  </application>
```

On définit ici deux bundle, le bundle strings est un simple bundle de libellés fixes, le bundle messages est un bundle de messages d'erreurs qui permet de redéfinir si besoin les messages d'erreur standards de Faces. Ces bundles sont considérés comme des ressources Java donc ici dans le

package ressources, dans ce package on aura les fichiers internationalisés, par exemple

Fichier messages_fr_FR.properties

```
javax.faces.component.UIInput.REQUIRED=Valeur requise
clientNotFound=Login inexistant
...
```

Fichier strings_fr_FR.properties

```
Client.login=Identifiant
Client.password=Password
button.login=Login
button.logout=Logout
...
```

et les mêmes avec les suffixes en_US par exemple

Nous verrons ci-dessous comment on peut changer la langue de l'interface.

Navigation

Une méthode action retourne une `String` qui détermine la navigation c'est à dire la prochaine vue. Si la méthode retourne `null`, ici en cas d'erreur, on reste sur la même vue. Sinon cette `String` peut être directement le nom du fichier xhtml, par exemple `welcome.xhtml` ou bien une `String` par exemple `welcome` correspondant à une règle de navigation dans le fichier `faces-config.xml`.

```
 <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-outcome>welcome</from-outcome>
      <to-view-id>/welcome.xhtml</to-view-id>
    </navigation-case>
 </navigation-rule>
```

Cycle de vie de l'EntityManager et de la transaction

La gestion de l'`EntityManager` en dehors d'une conteneur JEE complet pose un problème, quand le créer, le fermer, comment démarquer la transaction ?

On adopte ici une solution simple le pattern « Open session per Request » appelé également « Open session in view », un filtre de servlet `JPAFilter` s'interpose avant et après le traitement de toute requête JSF. L'`EntityManager` est créé avant le traitement de la requête, il est stocké dans le contexte de la thread (car il n'est pas thread-safe et que la requête est traitée dans une thread), la transaction est également ouverte. A la fin du traitement de la requête la transaction est commitée et l'`EntityManager` fermé. La méthode statique `JpaFilter.getEntityManager()` retourne ce même `EntityManager` toujours disponible donc tout au long du traitement de la requête.

Le filtre est une classe Java implémentant `javax.servlet.Filter` et les URL aux-quelles il s'applique sont définies dans le fichier de configuration `web.xml`.

```
public class JpaFilter implements Filter {
  private static Logger log = LoggerFactory.getLogger(JpaFilter.class);
  private static EntityManagerFactory emf;
  private static ThreadLocal<EntityManager> currentEntityManager = new ThreadLocal<EntityManager>();
  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
    EntityManager em = emf.createEntityManager();
    log.debug("creating EntityManager"+em.toString());
    currentEntityManager.set(em);
    try {
      em.getTransaction().begin();
      chain.doFilter(request, response);
      em.getTransaction().commit();
    } catch (Throwable ex) {
      em.getTransaction().rollback();
    }
    currentEntityManager.set(null);
    em.close();
  }
  public void init(FilterConfig filterConfig) throws ServletException {
    log.debug("Initializing filter...");
    String persistenceUnit = filterConfig.getInitParameter("persistence-unit");
    emf = Persistence.createEntityManagerFactory(persistenceUnit);
  }
  public void destroy() {
    emf.close();
  }
  public static EntityManager getEntityManager(){
    return currentEntityManager.get();
  }
}
```

et dans web.xml

```
  <filter>
    <filter-name>JpaFilter</filter-name>
    <filter-class>jpautil.JpaFilter</filter-class>
    <init-param>
      <param-name>persistence-unit</param-name>
      <param-value>BookStore</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>JpaFilter</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
  </filter-mapping>
```

Le paramètre étant le nom de la persistence-unit

Logging

Il est déconseillé dans une application serveur d'utiliser la sortie standard `System.out`, la bonne règle est d'utiliser un framework de logging qui dirigera ces sorties dans les journaux de log du serveur et surtout permettra de contrôler le niveau de log, fort en développement et faible en production. Malheureusement, l'historique des framework de logging en Java est assez chaotique et il existe d'autres produits que l'API standard `java.util.logging` (par exemple `log4j`, très utilisé). De plus il faut s'adapter à celui utilisé par le serveur, dans notre cas GlassFish. On utilise ici `slf4j` une facade qui s'adapte au système de logging sous-jacent. Enfin, il faut trouver l'emplacement du fichier `logging.properties` qui définit les niveaux de log. Dans notre cas, il se situe dans le répertoire `config` du domaine de Glassfish, et dans le cas d'un Glassfish intégré à Eclipse dans le workspace ! Dans ce fichier on ajoutera à la fin :

```
controller.level=FINE
```

Ceci spécifiant que pour les classes du package `controller` le niveau est FINE (correspondant à debug dans slf4j)

Moyennant tous ces efforts on pourra écrire dans notre `ClientController` :

```
  private static Logger log = LoggerFactory.getLogger(ClientController.class);
et
  log.debug("Login succès");
```

Quelques composant UI usuels

DataTable

Très utilisé, le composant `h:dataTable` permet de construire une table à partir d'une `Collection` (ou tout autre type itérable), soit par exemple à afficher une liste de clients :

```
<h:dataTable value="#{clientController.clientList}" var="client">
  <f:facet name="header">Clients</f:facet>
  <h:column>
    <h:commandLink action="#{clientController.selectClient(client.id)}">
      <h:outputText value="#{client.login}" />
    </h:commandLink>
  </h:column>
</h:dataTable>
```

On construit la `dataTable` à partir d'une liste obtenue à partir de `clientController`, l'itérateur étant `client`.

La balise `h:colum` définit une colonne, ici une seule, qui contient un lien avec le `login` du `client`

Les balises `h:header` ou `h:footer` permettent de définir les en-tête ou pieds au niveau de la table ou de la colonne.

Le pattern master->detail

Très souvent une colonne d'une table contient un lien qui correspond à une sélection, sélection d'un client dans l'exemple ci-dessus pour en afficher les détails. Plusieurs méthodes sont possibles pour transmettre à une méthode action l'id du client sélectionné .

Le plus simple est d'utiliser comme ci-dessus la possibilité de passer directement des paramètres à une méthode action.

```
public String selectClient(Long id){
  EntityManager em = JpaFilter.getEntityManager();
  selectedClient = (Client)em.find(Client.class, id);
  return "client.xhtml";
}
```

D'autres possibilités sont offertes pour ce pattern Master/Detail, par exemple attacher un listener au `commandLink`. Cependant on devra bien comprendre le mécanisme fondamental de JSF, les requêtes JSF sont des requêtes POST, à l'arrivée de la requête la view est restaurée avec le modèle de données sous-jacent, ici la liste, ce qui peut poser problème dans certains cas si le bean qui la fournit, ici `clientController`, est `RequestScoped`.

C'est pour cette raison que JSF proposait pour un `ManagedBean` le scope `ViewScoped` qui n'est malheureusement pas compatible avec CDI en JSF 2.1, le scope `ViewScoped` indispensable pour ne pas abuser du scope de Session est prévu en JSF2.2 pour les beans `@Named` CDI..

On peut vouloir dans certain cas traiter la navigation de façon plus traditionnelle avec une simple requête GET et ses paramètres . On écrirait alors :

fichier `category.xhtml` affichage de la liste des livres de la catégorie

```
<h:dataTable value="#{categoryBean.books}" var="book">
   <h:column>
    <h:link value="#{book.title}" outcome="book">
      <f:param name="selectedEntityId" value="#{book.id}"/>
    </h:link>
   </h:column>
 </h:dataTable>
```

Les balises `h:link` ou bien `h:button` spécifient une simple navigation sans traitement de la vue actuelle, `f:param` définit des paramètres de la requête. Dans la page d'arrivée on peut traiter les paramètres, par exemple :

```
<f:metadata>
  <f:viewParam name="selectedEntityId" value="#{bookBean.selectedEntityId}"></f:viewParam>
  <f:viewAction action="#{bookBean.selectEntity}" />
</f:metadata>
```

Les select en JSF 2.0

JSF 2.0 donne la possibilité de construire directement un select sur une liste , par exemple sur une liste de `Client`

```
<h:selectOneMenu value="#{clientController.selectedId}" valueChangeListener="#{clientController.selectIdChanged}">>
  <f:selectItem noSelectionOption="true" itemLabel="#{strings['list.choose']}" />
  <f:selectItems value="#{clientController.clientList}" var="client" itemLabel="#{client.login}" itemValue="#{client.id}" />
</h:selectOneMenu>
```

On indique ce qui doit être affiché dans le select, le `login`, et ce qui doit être retourné, l'`id`

Ici on a associé au select un listener dans lequel on peut traiter le changement d'id indépendamment de la méthode action de la forme.

Dans le bean `clientController` on aurait

```
public List getClientList(){
   EntityManager em = JpaFilter.getEntityManager();
   return em.createQuery("select c from Client c").getResultList();
}
private Long selectedId;
private Client selectedClient;

public Long getSelectedId() {
   return selectedId;
}
public void setSelectedId(Long selectedId) {
   this.selectedId = selectedId;
}
public void selectIdChanged(ValueChangeEvent e){
   EntityManager em = JpaFilter.getEntityManager();
   selectedId = (Long)e.getNewValue();
   if(selectedId != null)
     selectedClient = (Client)em.find(Client.class,selectedId);
   else
     selectedClient = null;
}
public Client getSelectedClient() {
   return selectedClient;
}
public void setSelectedClient(Client selectedClient) {
   this.selectedClient = selectedClient;
}
```

Compléments JSF

Composants composites

Un composant composite est une composition facelet qui regroupe un ensemble de composants, éventuellement associés à un bean de backing. Ce composant peut ensuite être réutilisé dans toute l 'application.

Exemple : un éditeur HTML

Considérons par exemple la construction d'un composant incluant du code javascript comme l'éditeur HTML TinyMCE, on pourra construire le composant composite :

Fichier resources/galland/tinymce.xhtml

```
<<ui:component 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:composite="http://java.sun.com/jsf/composite">
  <composite:interface>
    <composite:attribute name="value" />
  </composite:interface>
  <composite:implementation>
    <script src="#{request.contextPath}/resources/tiny_mce/tiny_mce.js" />
    <script>
       tinyMCE.init({
        mode : "specific_textareas",
        theme : "simple",
        editor_selector : "tinymce"
       });
    </script>
    <h:inputTextarea rows="5" cols="80" styleClass="tinymce" value="#{cc.attrs.value}" />
  </composite:implementation>
</ui:component>
```

Noter la localisation du composant dans un sous-répertoire du répertoire standard des ressources JSF.

Le composant composite définit une interface avec les attributs requis, puis une implémentation.

Ensuite dans une page de l'application on pourra définir le namespace

```
xmlns:dgc="http://java.sun.com/jsf/composite/galland"
```

et utiliser le composant :

```
<dgc:tinymce value="#{unBean.texte}"/>omposants JSFComposants JSF
```

Noter quand même que la plupart des bibliothèques de composants complémentaires incluent un tel composant éditeur

Exemple : un LanguageChooser

Autre exemple, un composant composite pour changer la langue :

Fichier resources/galland/localeChooser.xhtml

```
<ui:component 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:composite="http://java.sun.com/jsf/composite">
  <composite:interface>
  </composite:interface>
  <composite:implementation>
    <ui:repeat var="locale" value="#{localeBean.locales}">
      <h:commandLink action="#{localeBean.changeLocale(locale)}">
        <h:graphicImage name="#{locale}.png" library="flags" />
```

```
      </h:commandLink>
    </ui:repeat>
  </composite:implementation>
</ui:component>
```

en supposant que les drapeaux sont dans un sous-répertoire `flags` du répertoire `resources` et utilisant le bean

```
@ManagedBean
@SessionScoped
public class LocaleBean {
  private List<Locale> locales;
  private Locale locale;

  public LocaleBean() {
    locale = FacesContext.getCurrentInstance().getViewRoot().getLocale();
    if (locale == null)
      FacesContext.getCurrentInstance().getApplication().getDefaultLocale();
    Iterator<Locale> itl = FacesContext.getCurrentInstance().getApplication().getSupportedLocales();
    locales = new ArrayList<Locale>();
    while (itl.hasNext()) {
      locales.add(itl.next());
    }
  }

getter et setters usuels

  public String changeLocale(Locale locale){
    this.locale = locale;
    FacesContext.getCurrentInstance().getViewRoot().setLocale(locale);
    return null;
  }
}
```

Dans la définition de la vue (template.xhtml) on aura :

```
<f:view contentType="text/html" locale="#{localeBean.locale}">
```

et pour avoir un `localeChooser` dans une page de l'application :

```
<dgc:localeChooser/>
```

Réalisation de composants JSF en Java

On peut également définir des composants JSF comme des classes Java dérivant des classes de base de JSF, soit pour les composants UI si un composant composite ne convient pas, soit pour d'autres types de composants comme les `Converter` ou `Validator`. D'autres composants permettent d'intervenir dans le cycle de vie de JSF comme un `PhaseListener` ou bien un `NavigationHandler`

Comportement AJAX

Donnons maintenant un comportement Ajax au select :

```
<h:selectOneMenu value="#{clientController.selectedId}" valueChangeListener="#{clientController.selectIdChanged}">>
  <f:selectItem noSelectionOption="true" itemLabel="#{strings['list.choose']}" />
  <f:selectItems value="#{clientController.clientList}" var="client" itemLabel="#{client.login}" itemValue="#{client.id}" />
  <f:ajax execute="@this" render="client"></f:ajax>
</h:selectOneMenu>
<h:outputText id="client" value="#{clientController.selectedClient.login}"/>
```

La balise `f:ajax` indique le ou les composants qui seront transmis et traités (`execute`) dans une requête AJAX et le ou les composants qui seront rerendus (`render`) après la réponse. Un attribut `event` permet de spécifier l'événement JavaScript qui provoquera la requête AJAX, `valueChange` par défaut pour un select.

Les bibliothèques complémentaires

JSF fournit le mécanisme de base mais les balises `h:` qui définissent les composants UI sont limitées, de plus il n'y a aucun style CSS prédéfini.

On peut utiliser des bibliothèques complémentaires qui entre autres :

- Offriront des versions plus fonctionnelles des composant de base par exemple une `rich:dataTable` de la bibliothèque RichFaces est plus fonctionnelle que la simple `h:dataTable`.
- Offriront de nouveaux composants calendar, colorChooser, liste gauche-droite, menu etc... en prenant en charge tout le javascript associé.
- Offriront des composants au comportement AJAX, par exemple un suggest.
- Amélioreront les fonctionnalités AJAX
- Définiront un style CSS pour les composants pour donner un « skin » à l'application

Il existe plusieurs de ces bibliothèques, chacune avec ses avantages et inconvénients (fonctionnalités, licences, ..). Citons entre autres, RichFaces (Jboss), PrimeFaces et IceFaces. Il est conseillé de consulter la démonstration en ligne de ces bibliothèques.

Dans les versions antérieures de JSF le comportement AJAX n'était pas normalisé et pouvait amener des incompatibilités entre bibliothèques, avec JSF 2.0 on peut envisager d'utiliser simultanément des composants de diverses bibliothèques bien que cela ne soit pas forcément très cohérent.

Application Web dans un contexte JEE complet

Dans un contexte JEE complet :

- CDI gère le cycle de vie et la mise en relation des composants
- Les composants du tier métier sont des EJB transactionnels utilisant un EntityManager géré par le conteneur.

Context Dependency Injection

L'injection de dépendance est une nouveauté majeure de JEE7. Si le conteneur le supporte, l'injection de dépendances rend utilisable un composant dans un autre composant mais c'est le conteneur qui se charge de l'instantiation et du cycle de vie du composant injecté, tout ceci contrôlé par des annotations.

Noter qu'un conteneur Web simple comme Tomcat ne supporte pas l'injection de dépendance mais que l'on peut lui ajouter les bibliothèques pour ce faire.

Le couplage ainsi établi entre les composants est un couplage faible au sens où l'on va utiliser le composant sans savoir précisément comment il est fabriqué. Ainsi, changer la façon dont le composant injecté est obtenu n'a pas de conséquences sur le composant qui l'utilise.

L'injection de dépendance en JEE6 est typée, ce n'est pas une simple injection d'objet référencés par une clé

Attention :

L'utilisation de CDI suppose la présence dans WEB-INF d'un fichier `beans.xml` qui peut être vide mais qui peut aussi contenir de la configuration CDI complémentaire aux annotations

Quand on utilise CDI L'annotation `@Named` remplace l'annotation `@ManagedBean` Attention également à bien importer pour les annotations les packages CDI `javax.enterprise` et non `javax.faces` pour des raisons historiques les mêmes annotations se retrouvent dans les deux packages, choisir le mauvais amènerait des dysfonctionnements.

Reprenons quelques éléments de l'exemple précédent mais en utilisant CDI.

Soit par exemple un simple bean pour la saisie du login et du password :

```
@Named
@RequestScoped
public class LoginForm {
  private String login;
  private String password;

  @NotNull
  @Length(min = 3, max = 25)
  public String getLogin() {
    return login;
  }
  public void setLogin(String login) {
    this.login = login;
  }
  @NotNull
  @Length(min = 3, max = 25)
  public String getPassword() {
    return password;
  }
  public void setPassword(String password) {
    this.password = password;
  }
}
```

Notez les contraintes de validation, elle seront prises en compte automatiquement par JSF

`loginForm` sera référencé dans les champs de saisie de la page `login.xhtml`, par exemple `#{loginForm.login}`

La classe `ClientController` fait maintenant un large usage de l'injection :

```
@Named      //Avec CDI remplace l'annotation ManagedBean
@SessionScoped
public class ClientController implements Serializable {
  @Inject
  private LoginForm loginForm;
  @Inject
  private transient EntityManager em;
  @Inject
  private Logger log;
  @Inject
  private  MessageBean messageBean;

  private Client currentClient;
  @Produces  @LoggedIn @Named
  public Client getCurrentClient() {
    return currentClient;
  }

  public String doLogin() {
    log.debug("doLogin"+em.toString());
    try {
      currentClient = (Client) em.createQuery("select c from Client c where c.login=:login and c.password=:password")
          .setParameter("login", loginForm.getLogin()).setParameter("password", loginForm.getPassword())
          .getSingleResult();
    } catch (NoResultException e) {
        messageBean.addMessage("clientNotFound");
        return null;
    }
    return "welcome";
```

```
  }
  public boolean isLoggedIn(){
    return currentClient != null;
  }
  public String doLogout() {
    currentClient = null;
    return "welcome";
  }
}
```

Un bean `@SessionScoped` doit être sérialisable pour les besoins internes du conteneur (passivation, sauvegarde, transfert) par contre les attributs qui ne le seraient pas seront déclarés `transient`.

Injection simple

On injecte dans `ClientController` le bean `loginForm` pour récupérer les saisies.

Injection à partir d'une classe Producer

On injecte un `Logger` créé par une classe `LoggerProducer`, annotation `@Produces`

```
public class LoggerProducer {
  @Produces
  public Logger getLogger(InjectionPoint ip) {
      return LoggerFactory.getLogger(ip.getMember().getDeclaringClass());
  }
}
```

Injection avec Qualifier

On injecte un `MessageBean` pour traiter les messages d'erreur :

```
@Named
@RequestScoped
public class MessageBean {
  @Inject
  private FacesContext facesContext;
  @Inject @Messages
  private ResourceBundle bundle;

  public String getMessage(String key){
     return bundle.getString(key);
  }
  public void addMessage(String key){
    String summary = getMessage(key);
    FacesMessage msg = new FacesMessage(FacesMessage.SEVERITY_ERROR, summary, summary);
    facesContext.addMessage(null, msg);
  }
}
```

Ce `MessageBean` injecte lui même un `FacesContext` (créé par un `FacesContextProducer`) ainsi qu'un `ResourceBundle` lui même créé par un `MessageBundleProducer`. Comme il pourrait y avoir plusieurs `ResourceBundle` injectables on utilise une annotation `@Qualifier @Message`

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface Messages {}
```

Dans le `MessageBundleProducer` on précise le qualifier avec `@Produces`

```
public class MessageBundleProducer {
  @Inject
  private FacesContext facesContext;
  @Produces @Messages
  public ResourceBundle getBundle() {
    return (ResourceBundle)facesContext.getApplication().getResourceBundle(facesContext, "messages");
  }
}
```

Injection alternative

Dans `ClientController` on injecte également un `EntityManager`. Il peut y avoir bien sûr plusieurs façon d'obtenir cet `EntityManager` et l'on ne désire par fixer cela dans le code Java. La classe `OpenSessionInViewEntityManagerProducer` est donc marquée comme `@Alternative`

```
@Alternative
public class OpenSessionInViewEntityManagerProducer {
  @Produces
  @RequestScoped
  public EntityManager getEntityManager() {
    return JpaFilter.getEntityManager();
  }
}
```

Puisqu'il est obtenu à partir d'un filtre de requête, on précise ici que l'`EntityManager` injectable est `RequestScoped` (même s'il est injecté dans un bean `SessionScoped`)

Et c'est dans le fichier `beans.xml` que nous définissons l'alternative à choisir pour obtenir l'`EntityManager`

```
<beans
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/beans_1_0.xsd">
  <alternatives>
    <class>producer.OpenSessionInViewEntityManagerProducer</class>
  </alternatives>
</beans>
```

Liens avec la vue JSF

Noter que `ClientController` est lui même producer du client courant après login `currentClient`

```
  private Client currentClient;
  @Produces  @LoggedIn @Named
  public Client getCurrentClient() {
    return currentClient;
  }
```

On a ajouté un qualifier `@LoggedIn` pour la clarté et `@Named` pour que le client courant `currentClient` soit disponible dans l'EL :

Fichier top.xhtml

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
  <h:graphicImage name="logo.png" library="images" alt="" width="179" height="76" />
  <div style="float: right;">
    <h:form id="topForm">
      <h:panelGroup rendered="#{clientController.loggedIn}">
        #{currentClient.login}
        <h:commandLink action="#{clientController.doLogout}" value="Se déconnecter" />
      </h:panelGroup>
    </h:form>
  </div>
</ui:composition>
```

La couche métier, les EJB 3.1

Dans tout ce qui précède, nous n'avons pas abordé le problème du modèle, logique métier et accès à la base de données. La classe `ClientController` accédait directement à la base de donnée et de plus avec le pattern « Open Session In View » l'`EntityManager` est encore disponible lors du rendu de la vue, la transaction n'étant fermée qu'à la fin de la requête ce qui est un peu simpliste. On peut donc vouloir :

- Utiliser un pattern « Data Acces Object » pour séparer la couche métier et BD du contrôleur
- Contrôler plus finement l'`EntityManager` et les transactions

Ces deux aspects sont liés car si l'on abandonne le pattern « Open Session In View » il faudra gérer les transactions dans le DAO, ce qui n'est pas si simple.

La solution « normale » en JEE7 est d'utiliser pour le DAO et le modèle des beans EJB. Pour ces EJB c'est le conteneur qui va gérer automatiquement l'`EntityManager` et l'aspect transactionnel.

Dans les premières spécifications JEE les EJB étaient conçus comme des objets exposant une interface potentiellement remote et utilisant donc RMI/IIOP. Si de tels objets remote sont intéressants dans une optique de répartition, cela complique la réalisation des EJB et a un coût en performance.

La spécification EJB 3.1 permet de définir des EJB Lite, sans définition d'interface locale ni remote, qui s'exécutent dans la même JVM que leur client et donc avec un appel direct. Leur utilisation est donc considérablement simplifiée puisque ce sont de simples classes java, tout en bénéficiant de leurs possibilités transactionnelle et autres.

Il n'y a donc pas de raisons de ne pas utiliser ces EJB qui sont vraiment dans la logique de JEE7, mais rappelons quand même qu'il faut utiliser un conteneur EJB comme Glassfish et non un simple Tomcat.

Un EJB stateless no-interface

Un EJB stateless est sans état c'est à dire qu'il ne conserve aucune information entre deux appels.

```
@Stateless
public class ClientManager {
  @PersistenceContext
  private EntityManager em;

  public Client login(String login, String password) {
    Client client = null;
    try {
      client = (Client) em.createQuery("select c from Client c where c.login=:login and c.password=:password")
          .setParameter("login", login).setParameter("password", password).getSingleResult();
    } catch (NoResultException e) {
    }
    return client;
  }
}
```

La seule annotation `@Stateless` suffit à définir un EJB/Lite sans interface locale ni remote.

Un `EntityManager` géré par le conteneur est injecté.

Dans la couche Web le bean `clientController` injecte puis utilise cet EJB

```
@Inject
private ClientManager clientManager;
…
public String doLogin() {
  currentClient = clientManager.doLogin(loginForm.getLogin(), loginForm.getPassword());
  if (currentClient == null) {
    messageBean.addMessage("clientNotFound");
    return null;
  }
  return "welcome";
}
```

L'injection, de l'EJB est effectuée par CDI, l'ancienne annotation @EJB pourrait aussi être utilisée.

Un EJB de service CRUD générique

L'utilisation des génériques nous permet de définir une classe de base assurant les opération CRUD sur une classe entité. Dans le principe c'est un DAO que nous appellerons plutôt un Service. Commençons par définir une interface :

```
public interface GenericCRUDService<T> {
  public  void create(T t);
  public   T find(Object id);
  public   T update(T t);
  public void delete(Object id);
  public List findWithNamedQuery(String queryName);
  public List findWithNamedQuery(String queryName, Map<String,Object> parameters);
}
```

et son implémentation avec un EJB (Nota : l'utilisation des génériques dans le constructeur pour retrouver la classe entité associée n'est pas évidente ! On pourrait préférer une solution plus simple en la donnant comme argument du constructeur).

```
@Stateless
public class GenericCRUDServiceEJB<T> implements GenericCRUDService<T>{
  @PersistenceContext
  EntityManager em;
  private Class<T> entityClass;

  public GenericCRUDServiceEJB() {
    Type genericType = this.getClass().getGenericSuperclass();  // par exemple ClientServiceEJB
    Type[] params = ((ParameterizedType)genericType).getActualTypeArguments();
    entityClass = (Class)params[0]; //par exemple Client
  }
  public void create(T t) {
```

```
    em.persist(t);
  }
  public T find(Object id) {
    return (T) em.find(entityClass, id);
  }
  public T update(T t) {
    return (T)em.merge(t);
   }
  public void delete( Object id) {
    Object ref = em.getReference(entityClass, id);
    em.remove(ref);
 }
  public List findWithNamedQuery(String namedQueryName) {
    return this.em.createNamedQuery(namedQueryName).getResultList();
  }
  public List findWithNamedQuery(String namedQueryName, Map<String,Object> parameters) {
    Query query = em.createNamedQuery(namedQueryName);
    for(Entry<String, Object> entry : parameters.entrySet())
      query.setParameter(entry.getKey(), entry.getValue());
     return query.getResultList();
  }
}
```

Pour la classe entité `Client` on pourra alors définir l'interface

```
public interface ClientService extends GenericCRUDService<Client>{
  public Client login(String login, String password);
}
```

et son implémentation :

```
@Stateless
@Local(ClientService.class)
public class ClientServiceEJB extends GenericCRUDServiceEJB<Client> implements ClientService{
  public Client login(String login, String password) {
    Client client = null;
    try {
      client = (Client) em.createQuery("select c from Client c where c.login=:login and c.password=:password")
          .setParameter("login", login).setParameter("password", password).getSingleResult();
    } catch (NoResultException e) {
    }
    return client;
  }
}
```

Maintenant dans `ClientController` on pourra injecter un `ClientService`

```
@Inject
private ClientService clientService;
```

et l'utiliser :

```
public String doLogin() {
  currentClient = clientService.login(loginForm.getLogin(), loginForm.getPassword());
  if (currentClient == null) {
    messageBean.addMessage("clientNotFound");
    return null;
  }
  return "success";
}
...
public List getClientList(){
  return clientService.findWithNamedQuery("Client.findAll");
}
```

EJB Remote

Web Services

D'une façon très générale le terme Web Service recouvre la communication Application à Application sur support Web. Contrairement à des protocoles de communication relativement fermés et spécifiques à une architecture (RMI ou Corba IIOP en JEE, DCOM en .NET) on recherche le support de l'hétérogénéité avec des protocoles et standards très ouverts et un codage des données très universel tel que XML

Cependant, derrière ce terme deux visions très différentes s'opposent :

- Les Web Services SOAP sont dans l'idée des appels de méthodes distantes avec leurs paramètres, le tout encodé dans une requête et une réponse SOAP transportée par HTTP. Un document WSDL (Web Service Description Language) décrit très précisément l'interface du Web-Service. Par ce « contrat » Il y a donc un couplage assez fort entre le client et le serveur. Certes XML est utilisé par la couche SOAP pour encoder la requête et la réponse mais pour l'application cela correspond à un appel d'opération distante et elle ne voit pas ce XML.
- Un Web Service REST est similaire à la navigation d'un utilisateur sur le Web. On accède à des « resources » identifiées par une URL avec les types de requêtes usuelles du protocole HTTP (GET, POST, PUT et DELETE), les données de la requête et la réponse sont encodées en XML ou autre format comme JSON.

Ces deux visions sont supportées par JEE6 avec les spécifications JAX-WS pour les services SOAP et JAX-RS pour les services REST. Dans les deux cas, la sérialisation XML est assurée par JAXB.

Exemple de Web Service REST avec JAX-RS

En premier lieu dans le cas de Glassfish/Jersey paramétrer dans web.xml la servlet Jersey l'implémentation de JAX-RS

```
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
```

Cette classe définit un EJB qui assure un Web Service REST accessible par l'URL …/`rest/book?id=1` et qui retournera la représentation XML ou JSON d'un `Book`

```
@Stateless
@Path("/book")
public class BookResource {
  @EJB
  private BookService bookService;

  @GET
  @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
  public Book getBook(@QueryParam("id") String id) {
     return bookService.find(new Long(id));
  }
}
```

La classe `Book` devra être sérialisable par JAXB, il suffit de lui ajouter l'annotation `XmlRootElement`

```
@XmlRootElement
public class Book {
...
```

? Attention !!

Si on sérialise directement une classe `@Entity` (est-ce une bonne idée ?) et qu'il y a des relations définies, la sérialisation va trouver une boucle infinie, `Book->Category->Books->Book`, il faudra alors préciser par l'annotation `XmlTransient` les attributs qui ne doivent pas être sérialisés, par exemple dans `Category` :

```
@XmlTransient
@OneToMany(mappedBy = "category")
public List<Book> getBooks() {
  return this.books;
}
```

Les autres verbes HTTP pourraient être utilisés, par exemple POST pour créer une ressource.

Toute application pouvant soumettre une requête HTTP peut être cliente de ce service, on pourra tester simplement avec un navigateur.

Jersey fournit une API client qui permet de soumettre des requête HTTP, soit par exemple :

```
  public static void main(String[] args) throws JAXBException {
    Client client = ClientBuilder.newClient();
    WebTarget target = client.target("http://localhost:8080/BookStore-JEE6/rest").path("book").queryParam("id", "1");
    InputStream is  = target.request(MediaType.APPLICATION_XML).get(InputStream.class);
    JAXBContext context = JAXBContext.newInstance(Book.class);
    Unmarshaller um = context.createUnmarshaller();
    Book book = (Book) um.unmarshal(is);
    System.out.println(book.getTitle());
  }
}
```

Ici JAXB recrée un objet `Book` à partir de sa représentation en XML.

Un exemple de Web Service SOAP avec JAX-WS

Il existe plusieurs implémentations des Web Services SOAP en Java, par exemple AXIS ou Apache CXF qui ne sont pas conformes à JAX-WS. Nous utiliserons l'implémentation de référence METRO, celle incluse dans Glassfish.

La classe ci-dessous définit avec l'annotation `@WebService` un EJB assurant un Web Service SOAP, ici par défaut avec toutes ses méthodes.

```
@WebService
@Stateless
public class BookWebService {
  @EJB
  private BookService bookService;

  public List<Book> bookList() {
    return bookService.findAll();
  }
}
```

A partir de cette classe des outils (wsgen) vont permettre de créer le WSDL définissant l'interface de ce Web Service et des classes d'adaptation assurant coté serveur le décodage de la requête SOAP. Dans le cas de Glassfish, ceci sera fait automatiquement et dans le journal du serveur on verra au démarrage :

```
INFO: WS00019: EJB Endpoint deployed
 BookStore-JEE6  listening at address at http://dga-laptop:8080/BookWebServiceService/BookWebService
```

montrant bien que le Web Service est déployé.

L'URL :

http://localhost:8080/BookWebServiceService/BookWebService?wsdl

donne accès au WSDL du service et l'URL :

http://localhost:8080/BookWebServiceService/BookWebService?tester

permet de tester une requête et de voir la réponse SOAP

Pour réaliser un client de ce Web Service on part du WSDL, un outil wsimport permet de générer un ensemble de classes jouant le rôle d'un proxy pour ce Web-Service (ce n'est pas encore intégré dans Eclipse pour l'implémentation METRO, on doit appeler wsimport à la main ou dans un script ANT). Ensuite on pourra appeler le service, l'objet port jouant le rôle d'un proxy.

```
  public static void main(String[] args) {
    BookWebServiceService service = new BookWebServiceService();
    BookWebService port = service.getBookWebServicePort();
    List<Book>  books = port.bookList();
    for(Book b : books){
      System.out.println(b.title);
      System.out.println(b.getCategory().title);
      System.out.println(b.getAuthors().get(0).getFirstName());
    }
  }
```

SOAP vs REST

Si on considère les deux exemples ci-dessus on voit que les visions sont très différentes.

Un service REST définit des accès à des « ressources » identifiées par des URL , l'idée est simple et efficace. Cependant coté serveur, la signification de ces ressources dans l'application et la définition des URL associées est totalement à charge du développeur, quant au coté client à lui de voir également ce qu'il convient de faire de la représentation XML ou JSON retournée.

Un service SOAP définit un couplage fort avec contrat d'interface (WSDL) entre un objet de l'application cliente et un objet de l'application serveur

Liens et documentation à consulter

The Java EE 6 Tutorial

http://download.oracle.com/javaee/6/tutorial/doc/

Le tutoriel de référence sur JEE 6

Mojarra 2.1 Release Notes – Overview

Vous donnera en particulier la description des balises de base de JSF h : et f :