Explications supplémentaires sur le cours de JEE7 sur JPA2

Table des matières

Propos....3
Les fonctionnalités....3
Utilisation de l'application....3
Conception....6
Les trois couches applicatives....6
Couche persistence....7
Les concepts....7
Le fichier peristence.xml....7
SQLite Browser....8
EntityManagerExecutor....8
Le package « fr.cours.isima.fill.database »....8
Couche business....9
Couche présentation....10
Les choix....10
Comprendre les enchainements d'écrans....10
Le package « fr.cours.isima.presentation »....11
editArticle.jsp....11
Couche transverse....14

Propos

Ce document est un document de conception technique, dont le but est de mettre en œuvre un CRUD avec JPA2 et SQLite. Le projet « cours-jee-7-jpa » est complet et propose en plus d'exemples sur JPA, une interface Web complète avec tous les éléments que nous avons utilisés.

Ce document a valeur explicative afin de comprendre les subtilités et la conception du projet.

Les fonctionnalités

Notre modèle est très simple, nous voulons créer des « ***articles*** ». Ces « ***articles*** » sont liés à des « ***catégories*** ».

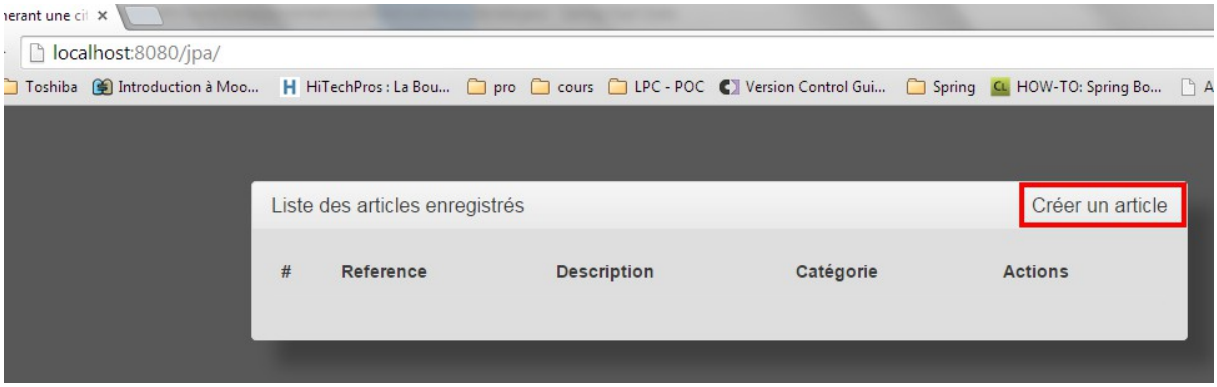
Pour respecter le modèle CRUD, le projet permet de :

- Créer des « ***articles »*** ;
- Les lister ;
- Les modifier ;
- Les supprimer ;

Utilisation de l'application

Après avoir installé l'application, vous pourrez la lancer via l'url http://localhost:8080/jpa/

Vous devriez voir apparaître l'écran suivant :



Le bouton de création de l'article permet d'accéder au formulaire de création afin d'ajouter un article :

Un clic sur le bouton enregistrer nous montre le système de contrôle des champs.

Corrigeons la saisie, pour un article valide :

Enregistrons, et découvrons notre liste, sur laquelle nous nous redirigeons automatiquement :

Le lien « supprimer » permet d'effacer l'article, et le bouton « modifier » ré-affiche le formulaire précédent, permettant à l'utilisateur de modifier un article.

Conception

Les trois couches applicatives

Ce point est évoqué dans le cours, mais l'application suit un découpage rigoureux pour respecter la séparation en couches suivantes :

Chaque classe d'un package appartient à une seule couche. La couche présentation appelle la couche métier qui elle-même se repose sur la couche persistance.

Couche persistence

La couche persistence est le cœur de notre 7ème cours, c'est donc le premier package présenté dans ce document.

Les concepts

Dans une couche persistence, il existe deux types d'objet :

- Les beans qui correspondent en général à une table ;
- Les dao qui effectuent les requêtes complexes pour récupérer les objets.

Pour la persistence nous utiliserons SQLite. Un des avantages est que l'intégration est telle que nous n'aurons même pas à nous préoccuper de la création des tables !

Le fichier peristence.xml

C'est un fichier de configuration centrale qui permet de définir l'emplacement de votre base et de signaler vos beans.

Comme dans l'exemple posez le dans « src/main/resources/META-INF/persistence.xml »

Observons celui de l'exemple comme point de départ :

```
<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"
	version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
	<persistence-unit name="jpa-exemple" transaction-type="RESOURCE_LOCAL">
		<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
		<class>fr.cours.isima.persistence.ArticleBean</class>
		<class>fr.cours.isima.persistence.CategorieBean</class>
		<validation-mode>CALLBACK</validation-mode>
		<properties>
			<property name="javax.persistence.jdbc.driver"
value="org.sqlite.JDBC" />
			<property name="javax.persistence.jdbc.url"
value="jdbc:sqlite:/REMPLACER PAR LE CHEMIN DE VOTRE FICHER DE BASE" />
			<property name="eclipselink.logging.level" value="ALL" />
			<property name="eclipselink.ddl-generation" value="create-
tables" />
		</properties>
	</persistence-unit>
</persistence>
```

Nous allons détailler les points que vous devez reconfigurer :

Persistence-unit

```
<persistence-unit name="jpa-exemple" transaction-type="RESOURCE_LOCAL">
```

Ne changez surtout pas le type de transaction. Le nom est important puisqu'il est utilisé par la suite dans le code pour récupérer la configuration.

Les beans

```
<class>fr.cours.isima.persistence.ArticleBean</class>
```

A chaque fois que vous souhaitez rajouter un bean, vous devez également le faire au niveau du persistence.xml afin que les annotations soient bien parcourues.

javax.persistence.jdbc.url

```
<property name="javax.persistence.jdbc.url" value="jdbc:sqlite:/REMPLACER PAR LE CHEMIN DE VOTRE FICHER DE BASE" />
```

Cette propriété vous permet de définir le chemin vers votre base de données. C'est un simple fichier. S'il n'existe pas, votre application le créera automatiquement. Par exemple : *D:/Logiciel/databases/test.db.*

eclipselink.logging.level

```
<property name="eclipselink.logging.level" value="ALL" />
```

Vous pouvez réduire le niveau de log par exemple à error si vous trouvez que l'affichage est trop verbeux.

SQLite Browser

Il existe un outil parfait pour visualiser la base de données SQLite : c'est le sqlLite Browser. Il se trouve à l'url suivante :

http://sqlitebrowser.org/

EntityManagerExecutor

Cette classe sert à exécuter les requêtes SQL.

Changez le peristence-unit

Si vous souhaitez changer le peristence-unit, il vous suffit d'utiliser le constructeur avec un String de EntityManagerExecutor. Voir le paragraphe sur « persistence-unit »

handleUniqueContraints

Parmi les méthodes rédigées, une s'appelle handleUniqueConstraints. Elle permet de gérer les contraintes d'unicité qui ne le sont pas correctement par défaut. Il y a tout de même quelques contraintes avec cette implémentation :

- Le nom du champ doit être le même que celui de la colonne de la table ;
- La contrainte d'unicité doit être sur un seul champ.

Le package « fr.cours.isima.fill.database »

Ce package permet d'initialiser une base vierge avec des objets Java. Pour vos projets je vous demanderai de reprendre une structure analogue avec une classe créant la base avec toutes les données dont vous pourriez avoir besoin.

Il est simplement composé d'une seule classe FillDataBase qui a une méthode main. En faisant un run as application dans Eclipse vous pourrez l'exécuter.

Couche business

La couche métier est assez simple dans notre application, elle est composée de deux classes de services ***Articles*** et ***Categories*** qui permettent d'accéder aux articles et catégories.

Les classes « ***Article*** » et « ***Categorie*** » sont les objets métiers. Ils encapsulent la couche persistence pour effectuer leurs traitements. L'objectif est de faciliter l'écriture de services. Les contrôles complexes ont lieu à ce niveau.

Vous remarquerez la présence de setter et de getter. Cette couche ne les expose pas forcément : certains champs se trouvant en persistence sont parfois calculés, modifiés… C'est tout l'intérêt d'avoir une couche métier.

Un coup d'œil rapide à ces deux classes qui ne dépassent pas les 100 lignes vous permettront de comprendre leurs fonctionnements.

Couche présentation

Les choix

Dans le prolongement de notre cours, la couche présentation se repose sur des Servlets et des JSP.

De plus quelques classes techniques situées dans le package « fr.cours.isima.presentation » et tous les contrôleurs et listener sont nécessaires au fonctionnement de l'application dans le package « fr.cours.isima.presentation.article ».

C'est sans doute la partie la plus complexe de notre application.

Comprendre les enchainements d'écrans

Le schéma suivant modélise la navigation telle qu'elle existe. N'oublions pas que systématiquement nous avons un enchainement Servlet=>JSP pour générer l'affichage. En bleu les Servlets, en noir les JSP, en jaune les boutons. Les transitions avec succès en vert, celles en échec en rouge. ListArticlesServlet est notre point d'entrée :

Ce schéma riche est en réalité assez simple avec un système reproductible pour d'autres types d'objets.

Le package « fr.cours.isima.presentation »

Classe	Commentaire
AppInit	Ce listener initialise les objets de scope application nécessaires au fonctionnement du CRUD. Il se repose sur ***ApplicationsObjects*** qui contient tous les objets nécessaires à l'exécution de nos services.
ErrorFields	Permet d'afficher des messages d'erreurs dans les formulaires de façon simple. Le paragraphe consacré à editArticle.jsp détaille la méthode utilisée
Field	Liée à ***ErrorFields***, vous n'aurez pas à l'utiliser seule
Form	Cet objet est utilisé dans editArticle.jsp pour afficher à la fois les valeurs correctes et les messages d'erreurs si besoin
NavigableServlet	Toutes les servlets héritent de celle-ci puisqu'elle offre quelques facilités pour réaliser la navigation entre les pages. Elle se repose sur l'objet « Page »
Page	Cet objet encapsule quelque peu l'API JEE pour simplifier la navigation entre les pages. Il permet de transmettre un ***Form*** ou un objet à exploiter dans la jsp.

editArticle.jsp

Cette page permet de créer et de mettre à jour nos articles. Elle gère aussi les messages d'erreurs quand un champ n'est pas valide.

Le formulaire

```
<form role="form" method="post" action="$
{pageContext.request.contextPath}/saveArticle">
```

Il pointe sur le chemin vers saveArticle la servlet de création et de mise à jour des articles.

La déclaration des champs

Voici un exemple de champs tels qu'il est utilisé dans le formulaire :

```
<div class="form-group ${form.errors.reference.kindOfMessage}">

    <input id="reference" name="reference" type="text" placeholder="Référence de
l'article" class="form-control input-sm" value="${form.bean.reference}">
    <c:if test="${form.errors.reference.error}">
        <span class="help-block"> ${form.errors.reference.errorMessage}</span>
    </c:if>
</div>
```

Vous remarquerez que vous avez deux champs :

- **form.bean** qui pointe sur le bean courant (par ex ***ArticleViewBean***)
- **form.errors** qui référence toutes les erreurs de validation. Par exemple ***${form.errors.reference.error}*** permet de tester si il y'a une erreur sur le champ référence tandis que ***${form.errors.reference.errorMessage}*** permet d'afficher le message correspondant.

Pour chaque champ vous pouvez utiliser un fonctionnement analogue à celui trouvé dans la page.

AddNewArticleServlet

La question est comment initialiser vos objets pour que votre formulaire puisse être utilisé comme indiqué ci-dessus. ***AddNewArticleServlet*** est la première page interrogée à l'ouverture du formulaire.

Voyons le code Java correspondant :

```
    protected Page process(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        final Articles articles = getArticles();

        final Article article = articles.createArticle();

        final Form<ArticleViewBean> formViewBean = successForm(new
ArticleViewBean(articles, getCategories(), article));


        // Un petit moyen d'encapsuler

        return forwardOnEdit().withForm(formViewBean).build();

    }
```

Nous créons un nouvel article et un objet form. Comme vous pouvez le voir nous utilisons le successForm() (import static de Form) qui signifie qu'il n'y pas d'erreur de validation (logique puisqu'aucun champ n'a été saisi).

La méthode fowardOnEdit() permet de demander la page d'édition et le withForm() permet de préciser que la forme à utiliser est celle que nous avons créée. Remarquez l'usage d'un ViewBean. Ils ont quelques traitements spécifiques pour faciliter l'affichage de notre formulaire.

SaveArticleServlet

Cette Servlet s'occupe de la sauvegarde des articles. Voyons le code :

```
@Override
protected Page process(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
   final long id = NumberUtils.toLong(req.getParameter("id"), -1);
   final Article article = findArticleById(id);
   final Optional<Categorie> optionalCategorie =
      getCategories().findCategoryById(
                  NumberUtils.toLong(req.getParameter("category")));

   if (optionalCategorie.isPresent()) {
      final Categorie categorie = optionalCategorie.get();
      article.setCategory(categorie);
      article.setDescription(req.getParameter("description"));
      article.setReference(req.getParameter("reference"));
      try {
        // Lors de la sauvegarde on controle les champs
         article.save();
         return redirectOnListArticles(req).build();
      } catch (final ConstraintViolationException e) {
         return returnToEditPage(article,
                  newErrorBuilder()
                  .addErrorWithUniqueContraintException(e)
                  .build());
      } catch (final UniqueConstraintException e) {
         return returnToEditPage(article, newErrorBuilder()
                  .addErrorWithUniqueContraintException(e)
                  .build());
      }
   }

   return returnToEditPage(article, newErrorBuilder().addField("category",
"Il faut selectionner une catégorie").build());

}
```

Dans la première partie du code nous saisissons les informations qui nous sont envoyées par la form suite au submit() comme nous l'avons vu précédemment :

```
final long id = NumberUtils.toLong(req.getParameter("id"), -1);
final Article article = findArticleById(id);
final Optional<Categorie> optionalCategorie =
         getCategories().findCategoryById(
            NumberUtils.toLong(req.getParameter("category")));

if (optionalCategorie.isPresent()) {
   final Categorie categorie = optionalCategorie.get();
   article.setCategory(categorie);
   article.setDescription(req.getParameter("description"));
   article.setReference(req.getParameter("reference"));
```

Une fois que l'objet article est rempli, nous pouvons procéder à sa sauvegarde :

```
try {
      // Lors de la sauvegarde on controle les champs
      article.save();
      return redirectOnListArticles(req).build();
}
```

Une fois l'article sauvegardé on retourne vers la liste d'articles. Pour gérer les erreurs de validation, nous utilisons le mécanisme des exceptions très efficace en Java.

Nous avons deux cas d'erreurs :

```
} catch (final ConstraintViolationException e) {
   return returnToEditPage(article, newErrorBuilder()
            .addErrorWithUniqueContraintException(e)
            .build());
}
```

En premier lieu nous gérons les erreurs de validation générée par les beans validators. L'instruction que nous voyons crée l'objet d'erreur à partir des informations de validation.

Le deuxième cas :

```
catch (final UniqueConstraintException e) {
      return returnToEditPage(article, newErrorBuilder()
         .addErrorWithUniqueContraintException(e)
         .build());
}
```

Ainsi nous gérons toutes nos erreurs et nos contraintes de façon simple. Comme vous le voyez les contrôles sont transparents à l'utilisation, il suffit simplement d'attraper les exceptions et de concevoir votre formulaire pour générer un affichage adéquat.

Couche transverse

Le package « common » regroupe les objets transverses. Il est petit pour notre modèle CRUD mais un des objets est particulièrement intéressant : ***ApplicationsObjects***.

Cette classe initialise toutes les classes Dao (***JpaArticleDao***, ***JpaCategorieDao***) et les classes de services (***Articles***, ***Categories***).

Vous pouvez reprendre cette structure dans vos projets. C'est une façon simple de construire vos objets qui se trouve dans l'Application Scope.

Si vous souhaitez mieux comprendre l'usage, je vous invite à consulter la javadoc.