Formation à *esup-commons*

Pascal AUBRY
IFSIC / Université de Rennes 1

Raymond BOURGES
CRI / Université de Rennes 1

Ce document est la version 1.4 du support des formations intitulées « Formation à *esup-commons* ». Il correspond à la version 0.17.2 de esup-commons.

La dernière version est accessible en ligne à l'*URL* suivante :
http://perso.univ-rennes1.fr/pascal.aubry/formation/esup-commons.

Pour toute remarque, merci de vous adresser à commons-devel@esup-portail.org.

Copyright

Table des matières

Table des exercices

1. Description de la formation

1.1. Informations générales

Public concerné : les ingénieurs et techniciens souhaitant apprendre à développer avec *esup-commons*, le framework de développement du consortium ESUP-Portail.

Durée : trois journées (21h).

Nombre de stagiaires : de 12 à 16. Seront acceptées en priorité les personnes souhaitant développer une application à court ou moyen terme au bénéfice de la communauté, par exemple en le mettant à disposition dans l'incubateur du projet ESUP-Portail.

Intervenants :

- Pascal AUBRY (IFSIC/Université de Rennes 1, pascal.aubry@univ-rennes1.fr)
- Raymond BOURGES (CRI/Université de Rennes 1, raymond.bourges@univ-rennes1.fr)

1.2. Objectifs

Les objectifs sont :

- Acquérir la méthodologie de *esup-commons* (séparation des couches, programmation par interface, respect des règles de programmation, utilisation systématique de subversion, …) et la maîtrise des outils nécessaires.
- Dès la fin de la formation, être capable de développer une application sous forme de *portlet* ou *servlet*, en utilisant *esup-commons*.

1.3. Pré requis

Cette formation adresse toutes couches de programmation depuis l'accès aux données et aux services, jusqu'à l'interface homme/machine en passant par les services métier et la logique applicative.

Les techniques de développement abordées sont nombreuses, parfois complexes ; pour pleinement profiter de cette formation et remplir les objectifs annoncés, les stagiaires devront au minimum être familier de la programmation en Java sous l'environnement *Eclipse*.

De plus, la maîtrise des éléments suivants est recommandée :

- *JSP* ou *JSF*
- *Hibernate* ou *Ibatis*
- *Spring*

Les développeurs sur-connaissants profiteront d'autant mieux de cette formation ;-)

1.4. Références

- Le projet ESUP-Portail (http://www.esup-portail.org).
- Le site du projet *esup-commons* (http://sourcesup.cru.fr/*esup-commons*)
- La page de la formation (http://perso.univ-rennes1.fr/pascal.aubry/formation/esup-commons)
- esup-commons : un *framework* de développement pour le projet ESUP-Portail, ESUP-Days 3, janvier 2007 (http://perso.univ-rennes1.fr/pascal.aubry/presentations/commons-esupdays3)

1.5. Programme

1er jour, 10h-12h30 : DÉCOUVRIR (généralités, installation de l'environnement)

- Présentation générale
- Installation/configuration Eclipse
- Organisation des fichiers

1er jour, 13h30-18h : COMPRENDRE (les bases du framework)

- Spring
- Déploiement en quick-start
- JSF
- Internationalisation
- Gestion des exceptions

2ème jour, 8h30-12h30 : METTRE EN ŒUVRE (fonctionnalités indispensables)

- Accès aux données
- Pagination
- Accès au S.I. (LDAP)
- Accès au S.I. (portail)
- Numérotation des versions

2ème jour, 13h30-16h30 : MAÎTRISER (fonctionnalités avancées)

- Distribution d'une application
- Commandes batch
- Web services
- Liens hypertextes directs
- Téléchargement de fichiers

3ème jour, 8h00-12h00 : MAÎTRISER (fonctionnalités avancées)

- Authentification
- Déploiement en servlet/portlet
- Gestion des caches
- Courier électronique
- FCK Editor

3ème jour, 13h00-16h00 : METTRE EN PRATIQUE

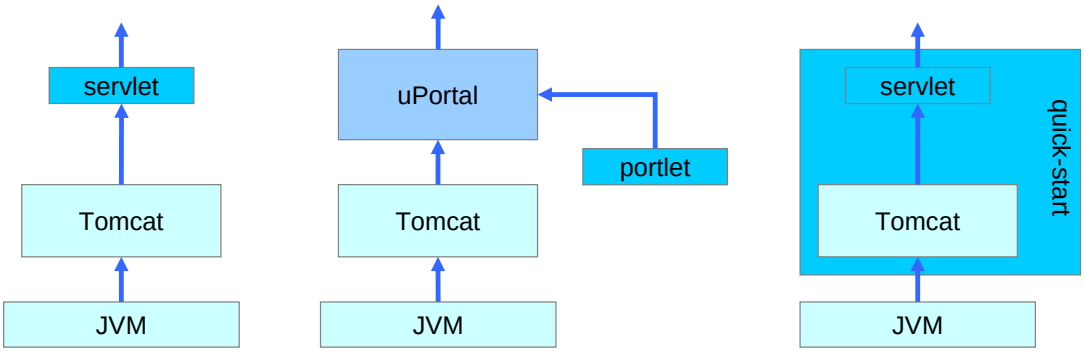
- La dernière demi-journée est libre : les formateurs aident les stagiaires à commencer un développement bâti sur esup-commons, en mettant l'accent sur les fonctionnalités qui les intéressent le plus.La première demi-journée est consacrée à la présentation générale de *esup-commons* et l'installation de l'environnement de travail. Les parties suivantes de ce support seront abordées :

2. Généralités

2.1. Déploiement des applications

Il existe avec *esup-commons* trois manières de déployer les applications :

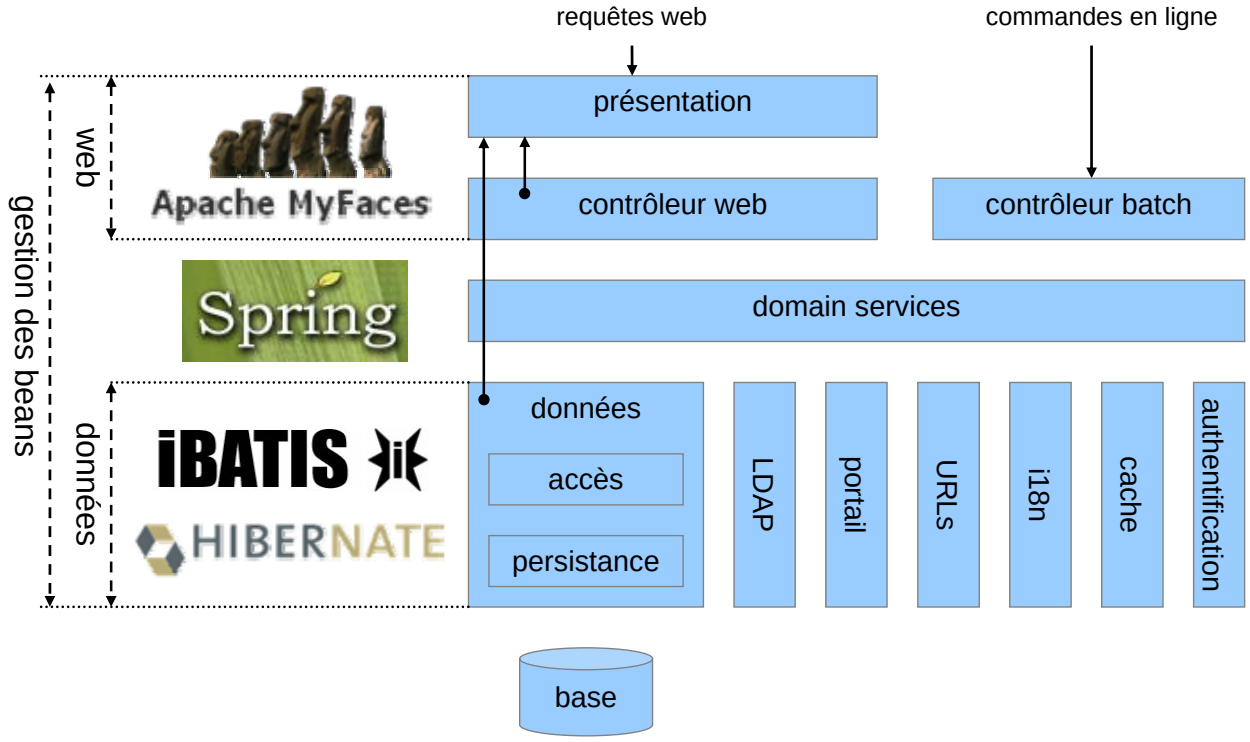
Servlet / portlet / quick-start



2.2. Architecture logicielle

Toute application bâtie sur *esup-commons* est articulée en couches séparées et interchangeables grâce à :

- Une programmation systématique par interfaces *Java*,
- L'utilisation de *Spring* pour l'injection de données.



3. Commencer à travailler avec *esup-commons*

Cette partie montre :

- Les prérequis pour développer avec *esup-commons*
- Comment installer l'environnement de développement *Eclipse*
- Comment configurer *Eclipse*
- Comment créer le projet *esup-commons*

Deux applications bâties sur *esup-commons* sont fournies sur le dépôt SVN :

- *esup-example*, qui est un exemple d'application qui illustre comment peuvent être utilisées toutes les fonctionnalités de *esup-commons*,
- *esup-blank*, qui est une application vide, et qui constitue un canevas de départ pour le développement d'une nouvelle application à partir de *esup-commons*.

La fin de cette partie montre comment créer le projet *esup-blank*.

3.1. *Installation d'*Eclipse

Pour la formation à *esup-commons*, les postes de travail sont pré-configurés, et *Eclipse* est déjà installé avec les *plugins* demandés. Les stagiaires doivent néanmoins configurer *Eclipse* et ses *plugins*.

JDK

Installer un *JDK* 1.5.0 (machine virtuelle *Java* et outils de développement).

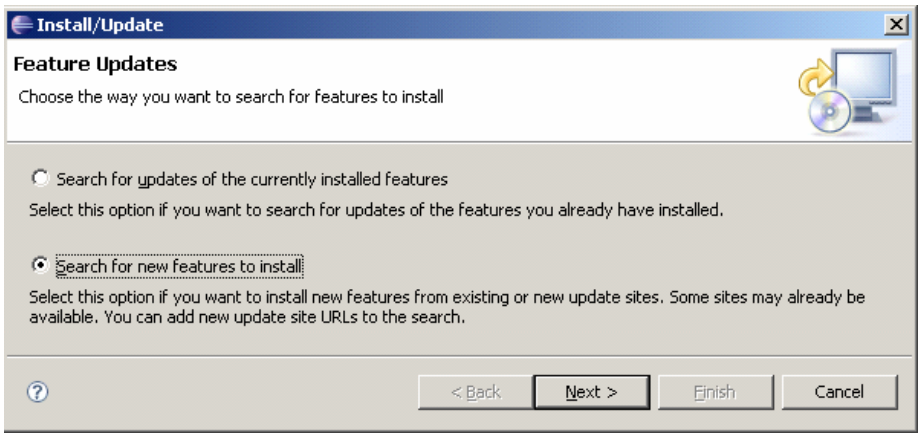
Suivre « *Popular downloads* » puis « *Java EE 5 SDK* » sur http://java.sun.com.

Eclipse

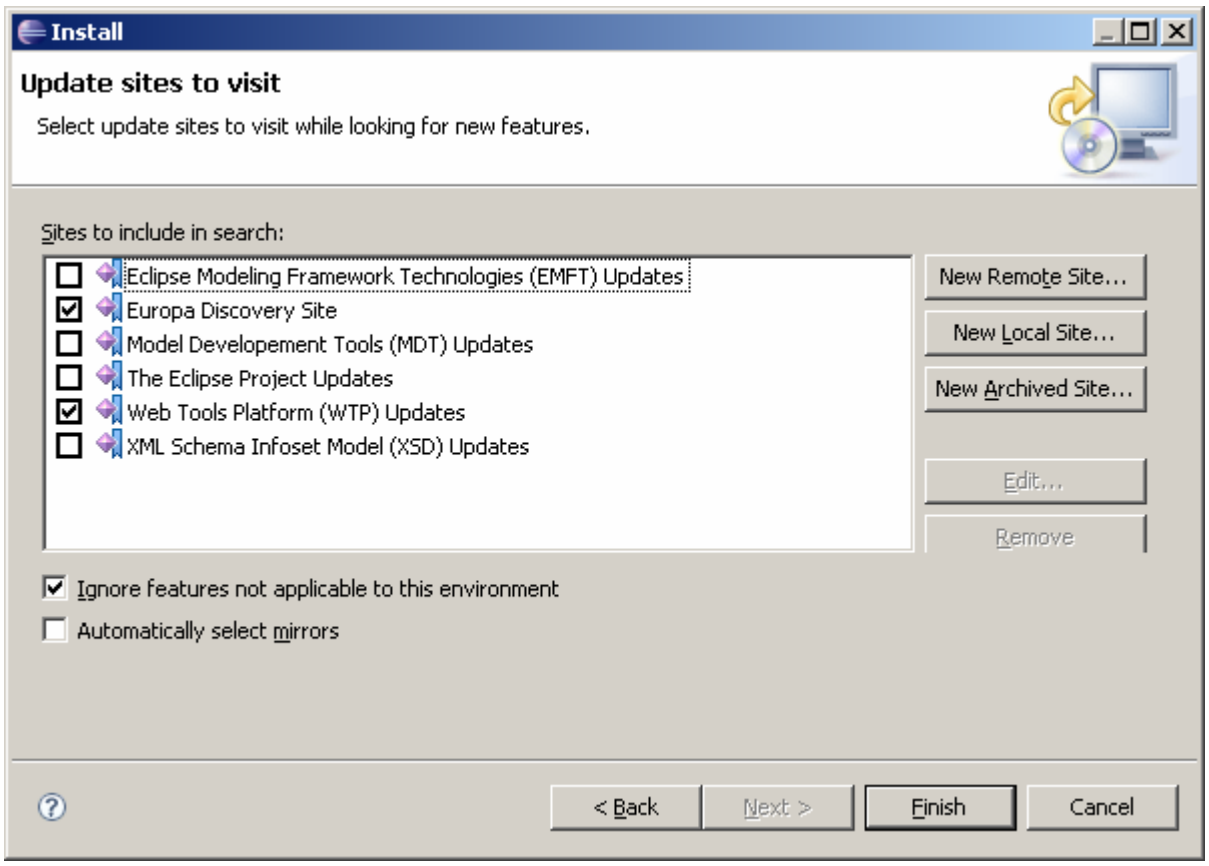
Installer le package *Europa* de nom : *Eclipse IDE for Java Developers* depuis http://www.eclipse.org/downloads/

Ajouts EUROPA

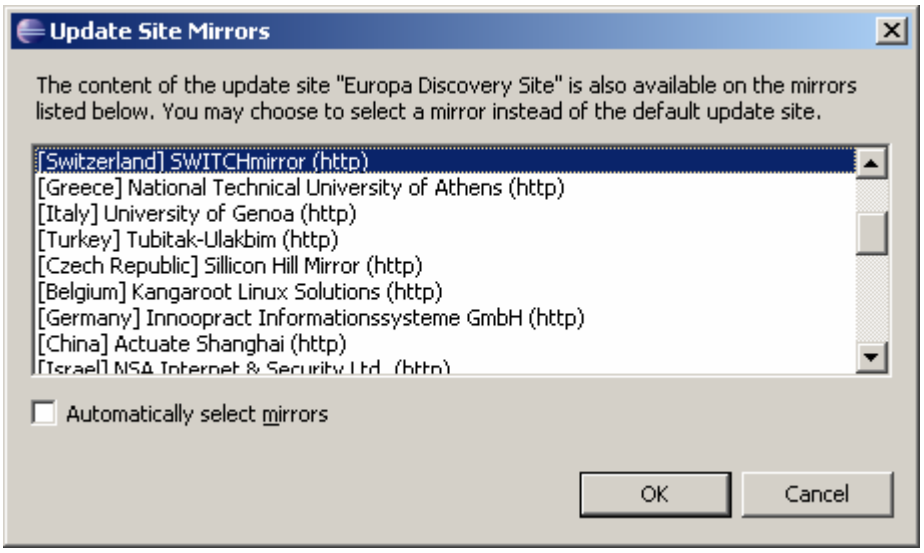
Dans *Eclipse*, menu « *Help* », puis « *Software Updates* », « *Find and install* », puis « *New features* » :



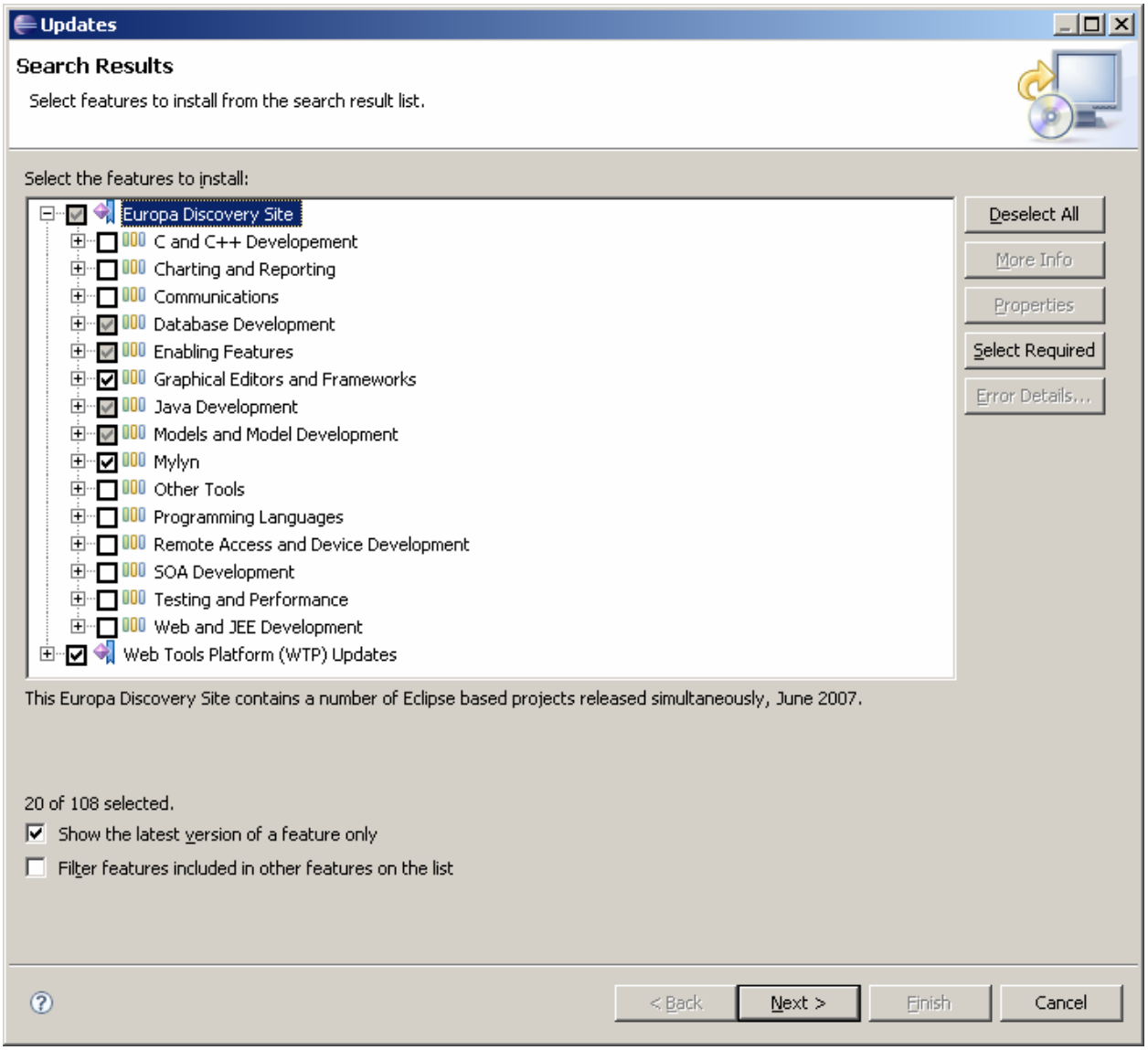
Choisir « *Europa discovery site* » et « *Web Tools Platform (WTP) Updates* » :



Choisir un miroir :

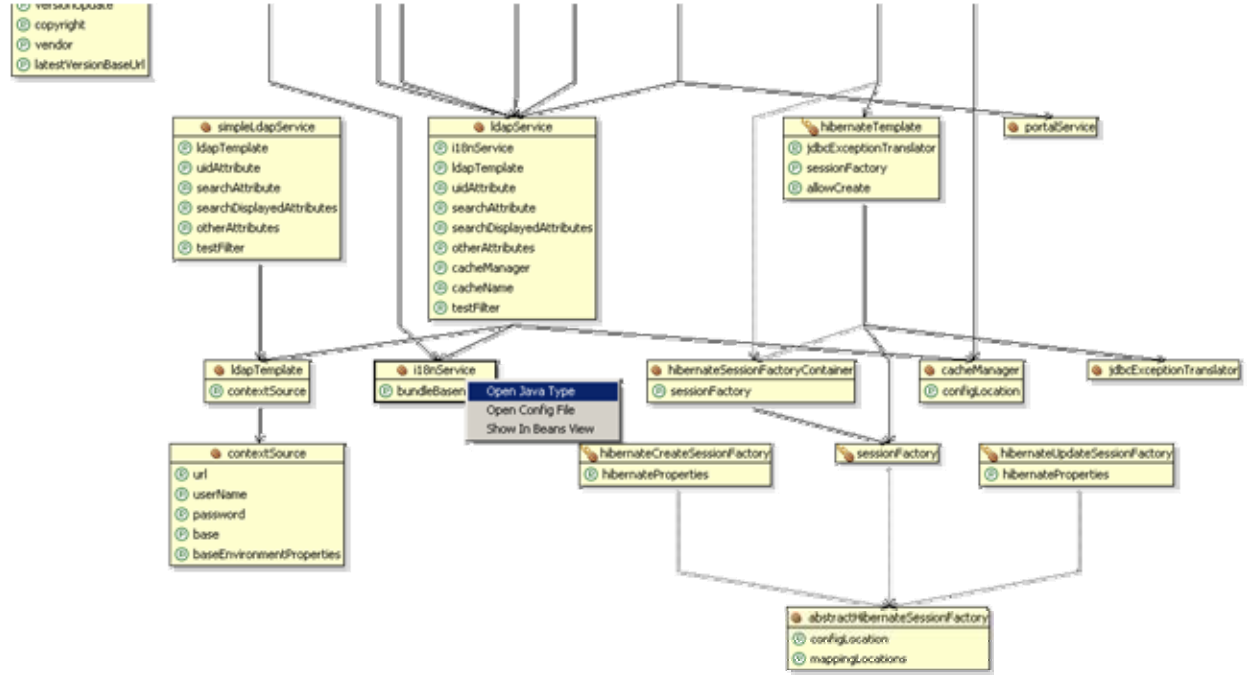


Sélectionnez (au minimum), pour *Europa*, les éditeurs graphiques et *Mylyn*, puis Sélectionner tout WTP. Enfin, ajoutez tous les éléments requis :



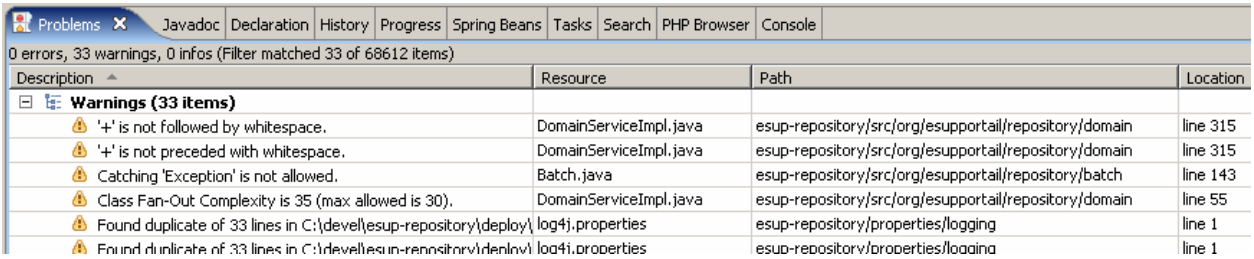
Spring IDE

Dans *Eclipse*, menu « *Help* », puis « *Software Updates* », « *Find and install* », « *New features* », « *New remote site* », puis http://www.springide.org.



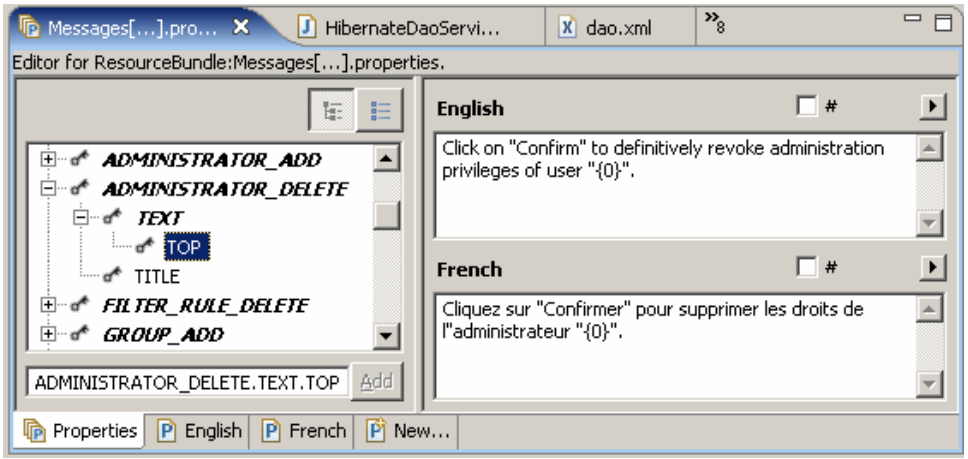
Checkstyle

Dans *Eclipse*, menu « *Help* », puis « *Software Updates* », « *Find and install* », « *New features* », « *New remote site* », puis http://eclipse-cs.sourceforge.net/update.



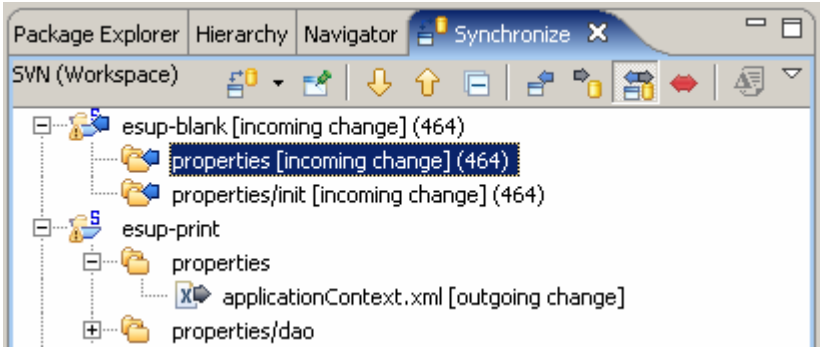
Resource Bundle Editor (RBE)

Télécharger depuis http://www.resourcebundleeditor.com, décompresser à la racine d'*Eclipse* et redémarrer *Eclipse*.



Eclipse SubVersion Plugin

Dans *Eclipse*, menu « *Help* », puis « *Software Updates* », « *Find and install* », « *New features* », « *New remote site* », puis http://subclipse.tigris.org/update_1.2.x.



3.2. Configuration d'Eclipse

Checkstyle

esup-commons utilise des règles de vérification de syntaxe assez strictes, basées sur le fichier **`/utils/checkstyle/checkstyle.xml`**.

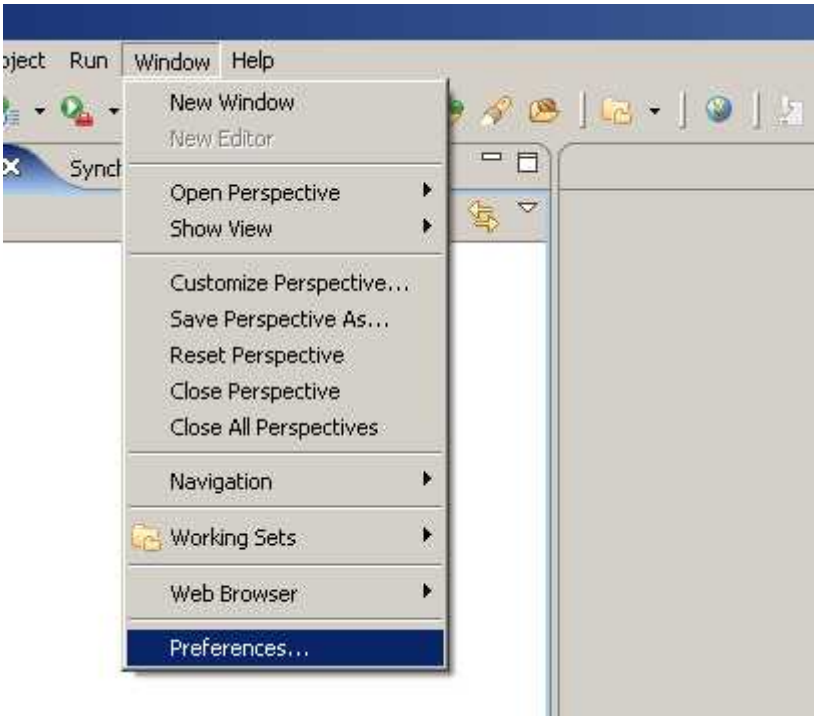
Nous montrons ici comment :

- Créer une configuration *Checkstyle* dans *Eclipse* qui correspond aux critères du projet *esup-commons*.
- Appliquer une configuration *Checkstyle* existante à un projet.

Exercice 1 : Créer une configuration *Checkstyle* pour *Eclipse*

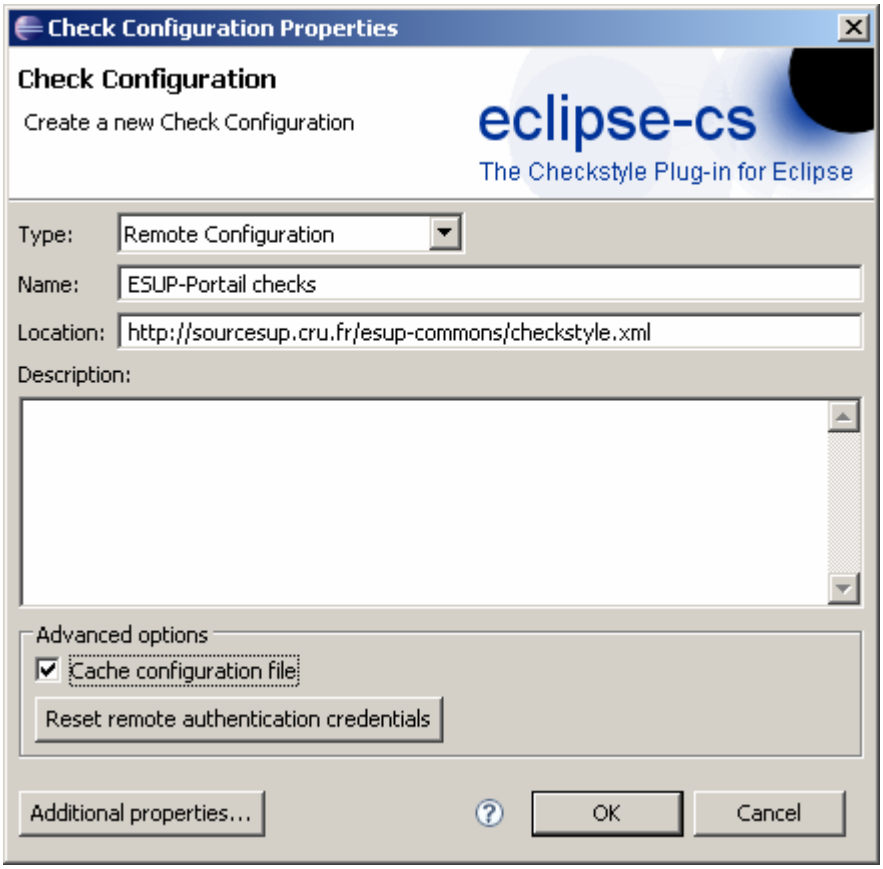
Suivre les instructions données ci-dessous pour créer la configuration correspondant aux vérifications ESUP-Portail dans *Eclipse*.

Ouvrir les préférences d'*Eclipse* :



Ouvrir les préférences *Checkstyle* et cliquer sur « Nouveau » pour créer une nouvelle configuration *Checkstyle* :

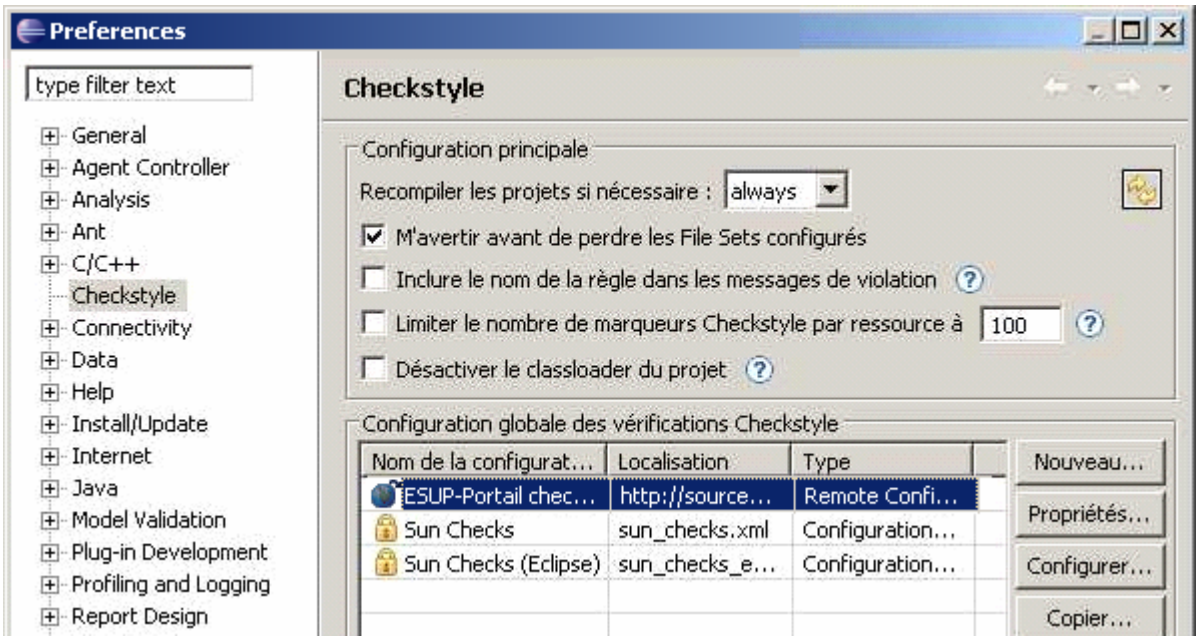
Créer une configuration distante basée sur le fichier de configuration situé sur le site de référence du projet (**`http://sourcesup.cru.fr/esup-commons/checkstyle.xml`**) :



Note : cacher le fichier de configuration pour pouvoir travailler hors connexion.

TODO : mettre le fichier checkstyle sur le site web.

Les vérifications *esup-commons* doivent désormais apparaître dans la liste de vos configurations :



Cette configuration sera appliquée ultérieurement au projet *esup-commons*, ainsi qu'à tous les projets dépendant de *esup-commons*.

Erreurs/avertissements du compilateur Java

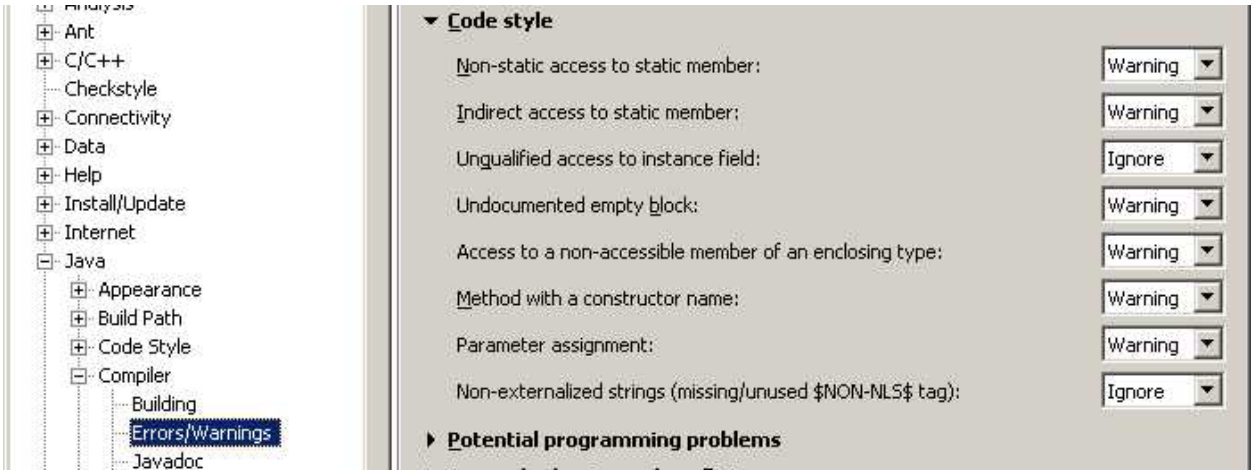
esup-commons utilise des options de compilation assez strictes pour détecter les faiblesses de code (en plus des problèmes de syntaxes détectés par *Checkstyle*).

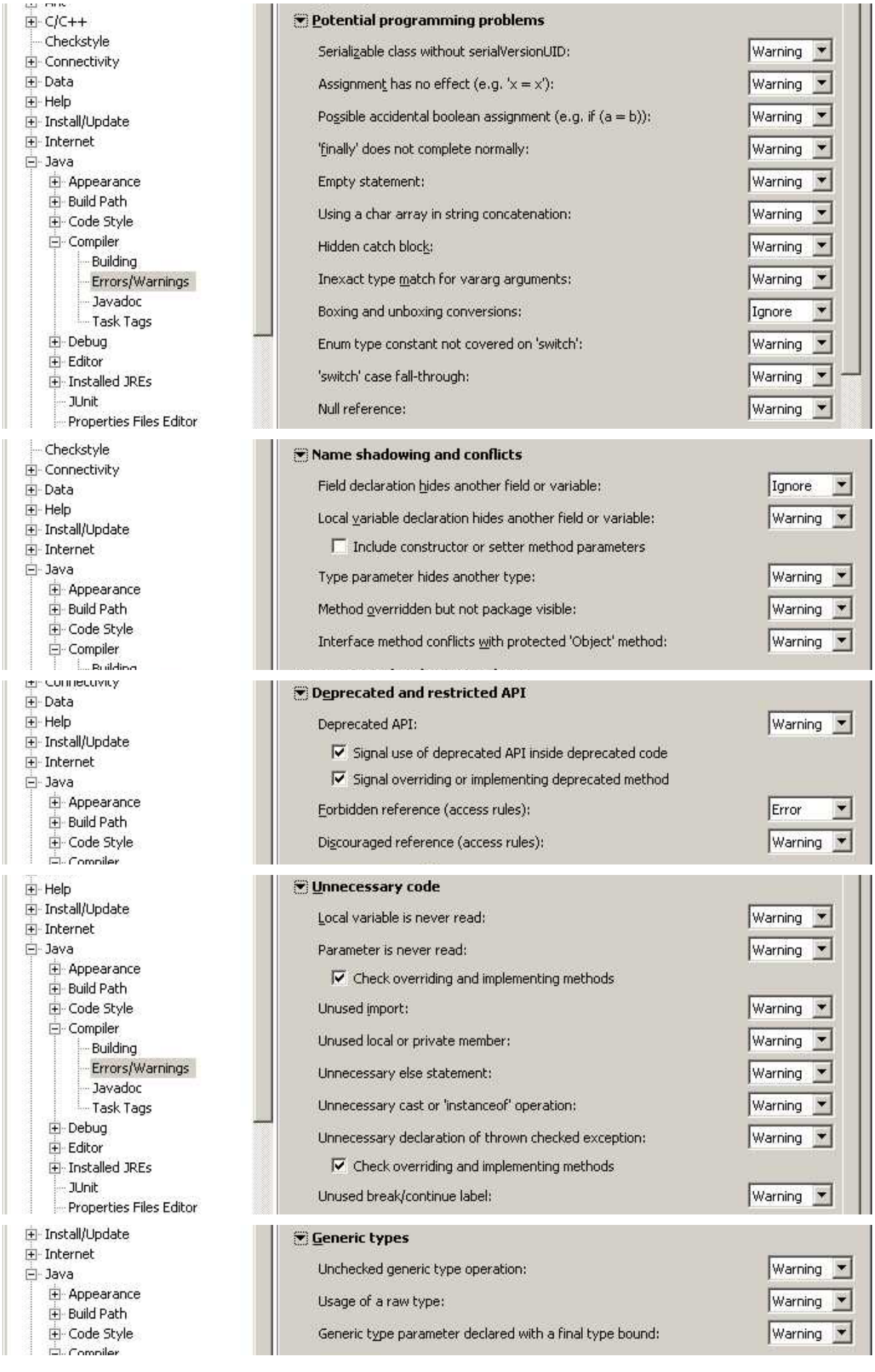
Nous montrons ici comment indiquer ces options dans *Eclipse*.

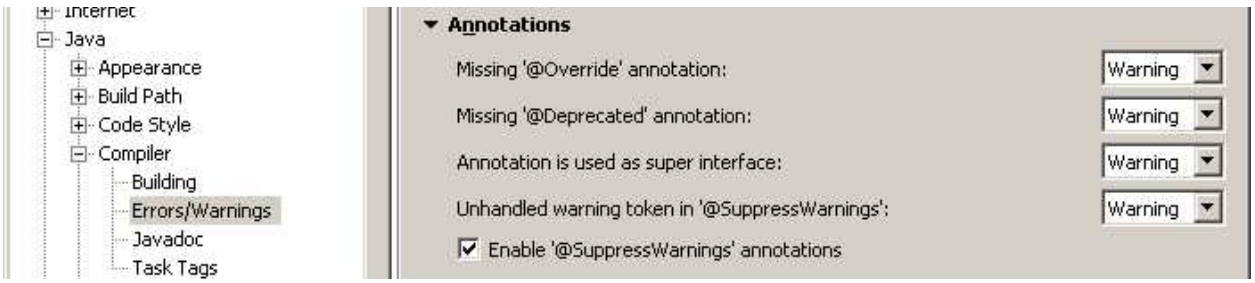
Exercice 2 : Configurer les erreurs/avertissements du compilateur *Java*

Configurer les erreurs/avertissements du compilateur *Java* comme indiqué sur les saisies d'écran ci-dessous.

Ouvrir les préférences d'*Eclipse* puis modifier les options de compilation (« *Java* » puis « Compilateur ») :





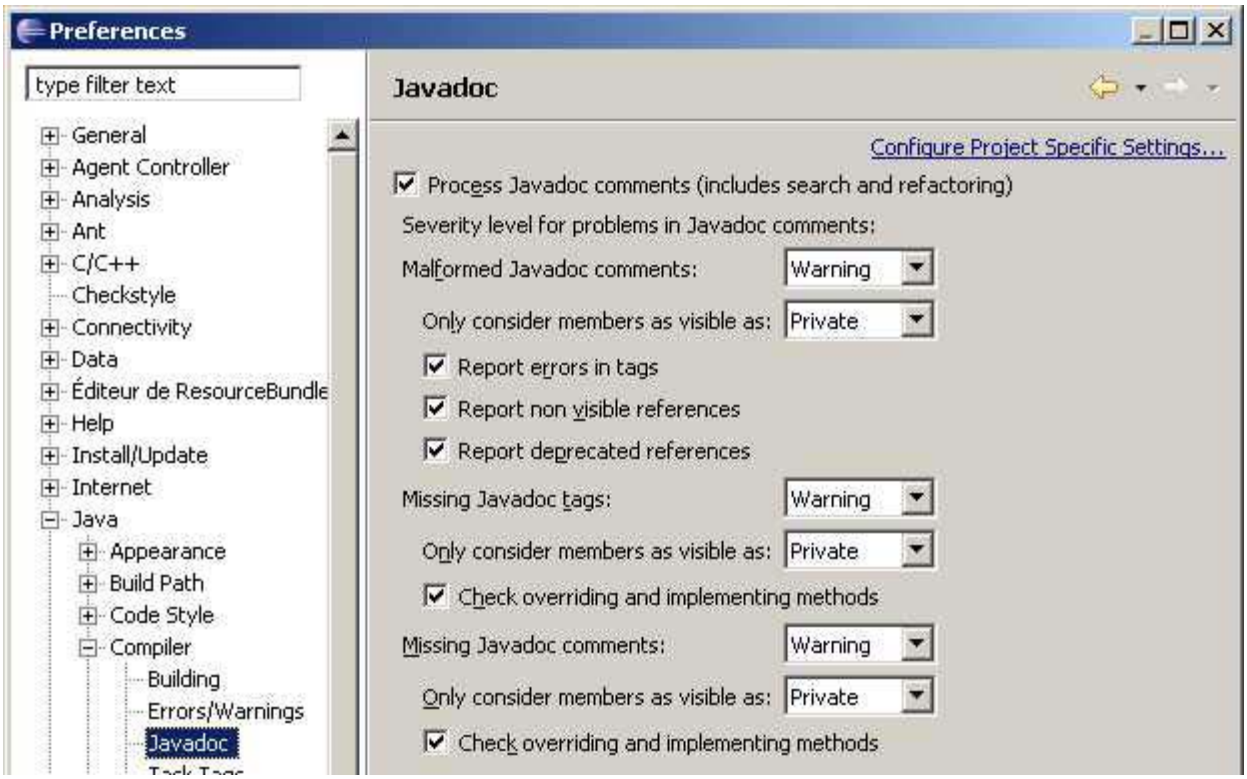


Vérifications de Javadoc

esup-commons utilise également des règles assez strictes pour *Javadoc*,

Exercice 3 : Configurer le vérificateur de *Javadoc*

Configurer le vérificateur de *Javadoc* comme indiqué ci-dessous.

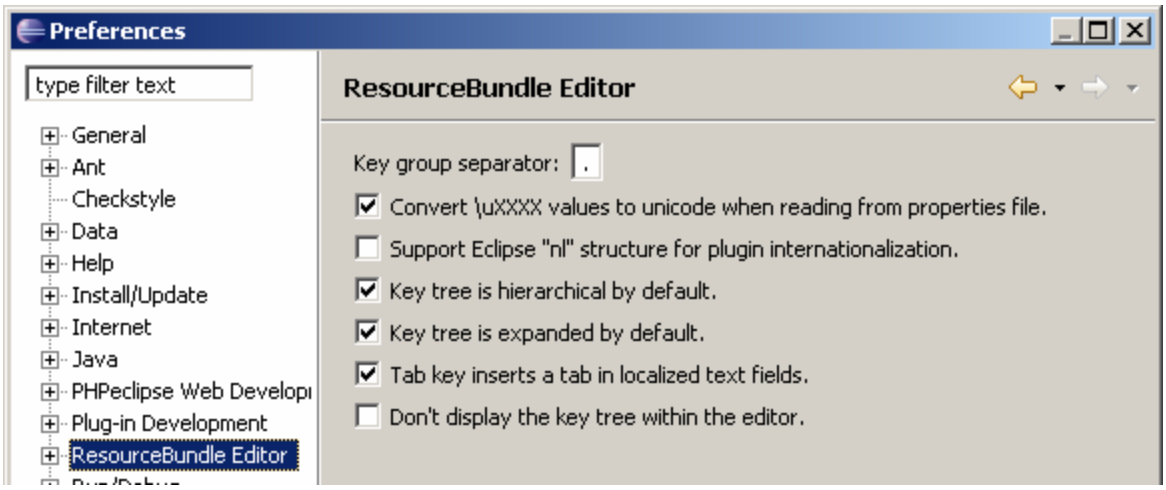


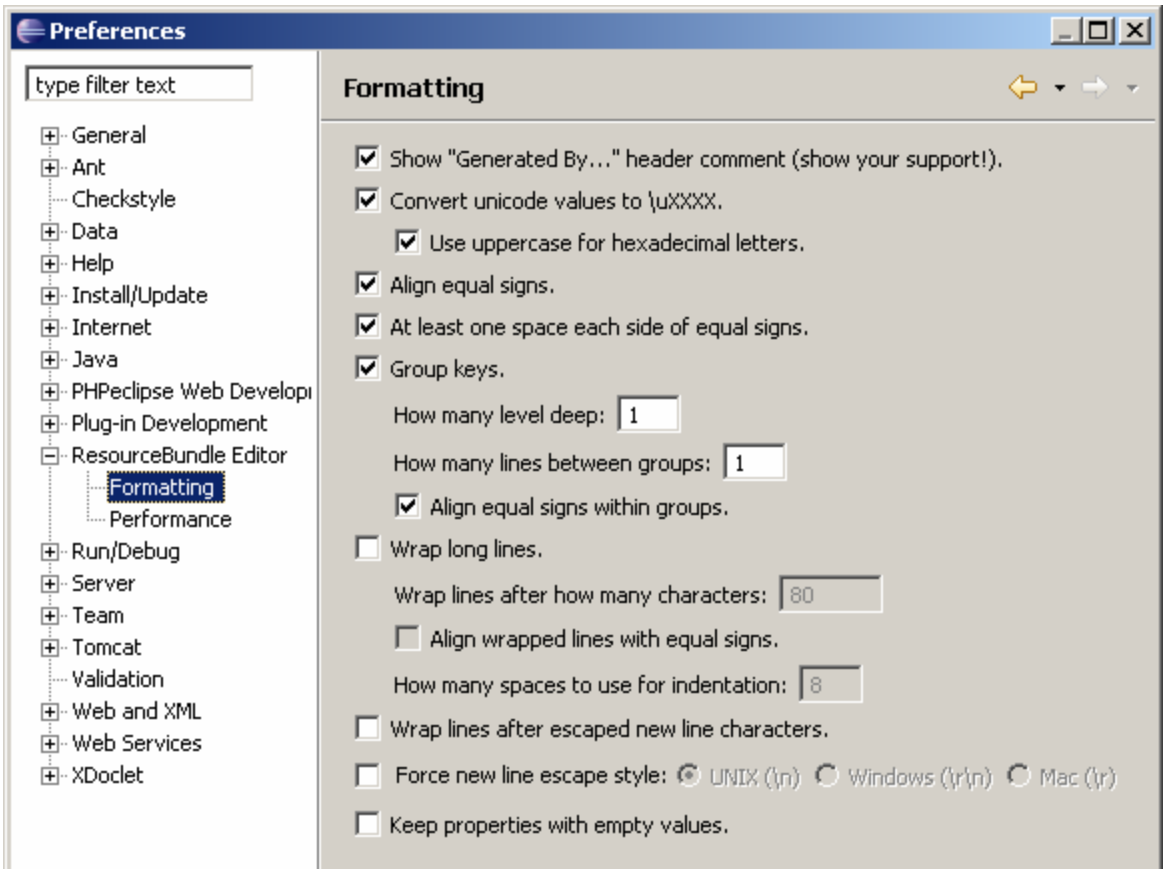
RBE

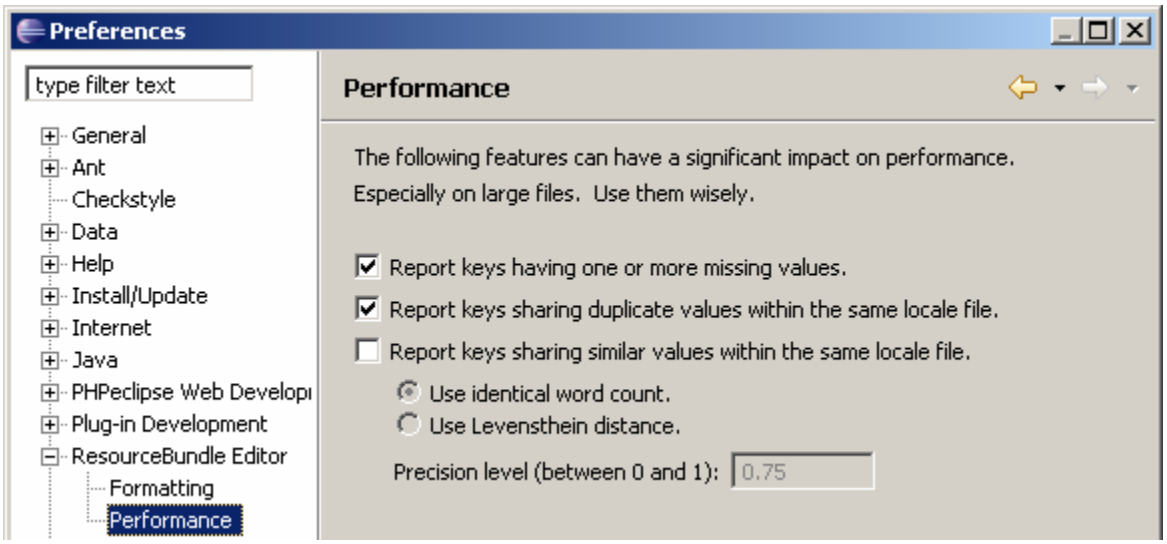
RBE (*Resource Bundle Editor*) est utilise pour éditer les fichiers de ressources de l'internationalisation. Il doit être configuré de la même manière par tous les développeurs pour obtenir le même formatage et ainsi éviter les problèmes de conflits sur le dépôt *SVN*.

Exercice 4 : Configurer le *plugin RBE*

Configurer le *plugin RBE* comme indiqué sur les saisies d'écran ci-dessous.







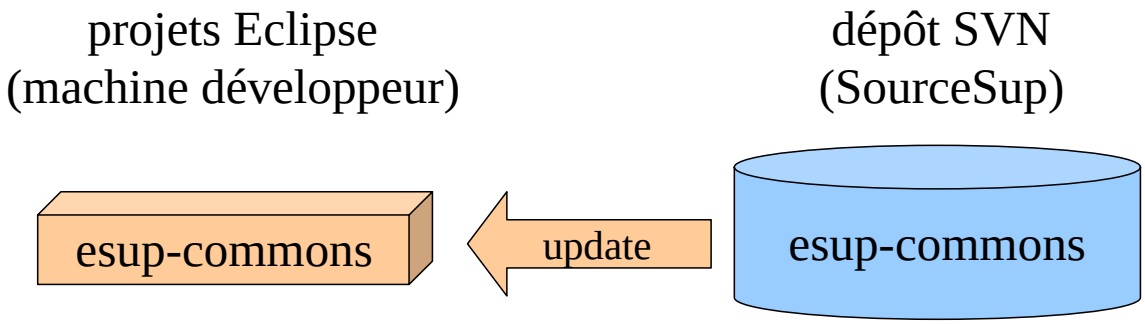
3.3. Démarrer un nouveau développement avec esup-commons

Cette partie montre le principe du démarrage d'un nouveau développement basé sur *esup-commons*.

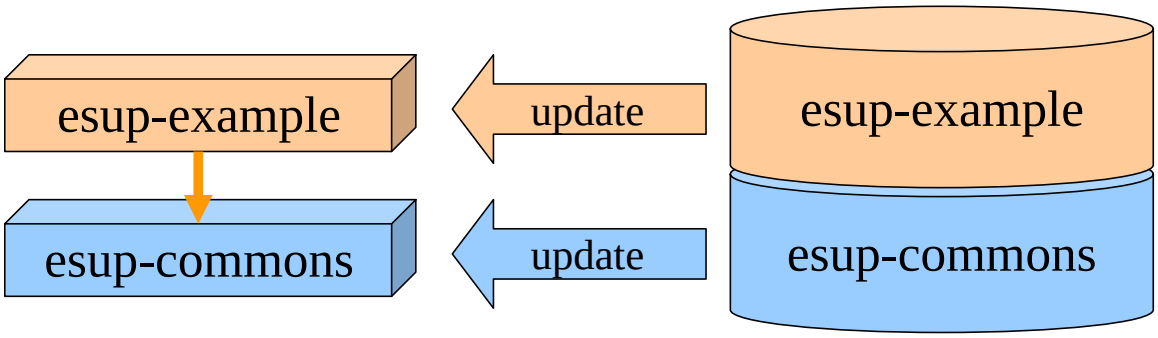
Le dépôt SVN du projet *esup-commons* est constitué de trois sous-projets :

- La bibliothèque *esup-commons*,
- L'application d'exemple *esup-example*,
- L'application blanche *esup-blank*.

Le sous-projet *esup-commons* rassemble tous les éléments communs à toutes les applications basés sur *esup-commons*. Il est importé depuis le dépôt SVN, et la liaison SVN est conservée pour faire évoluer les applications en fonctions des évolutions de *esup-commons*.

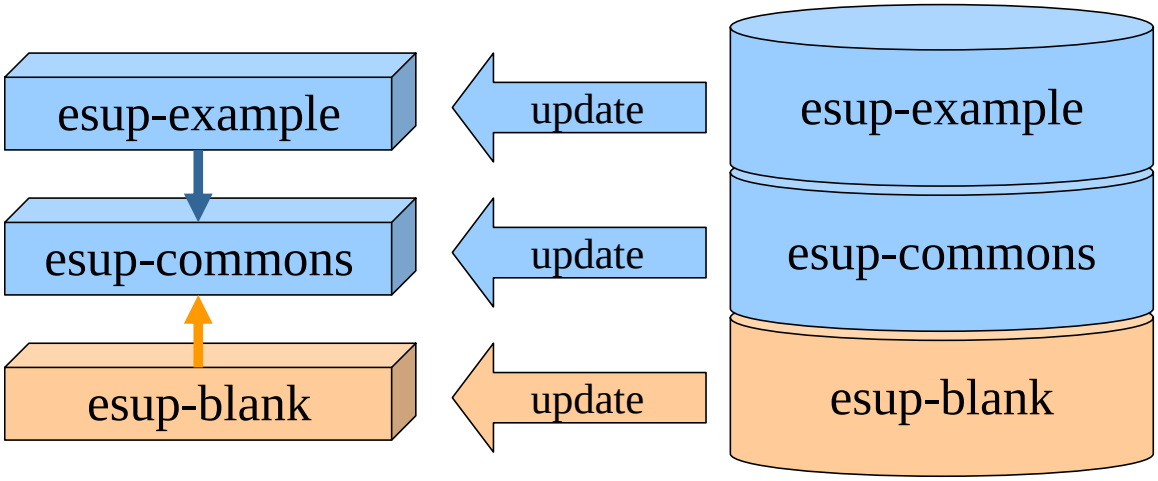


Le sous-projet *esup-example* est une application d'exemple qui montre comment utiliser les différentes fonctionnalités de *esup-commons*. *esup-example* est importé depuis le dépôt SVN, et la liaison SVN est conservée :

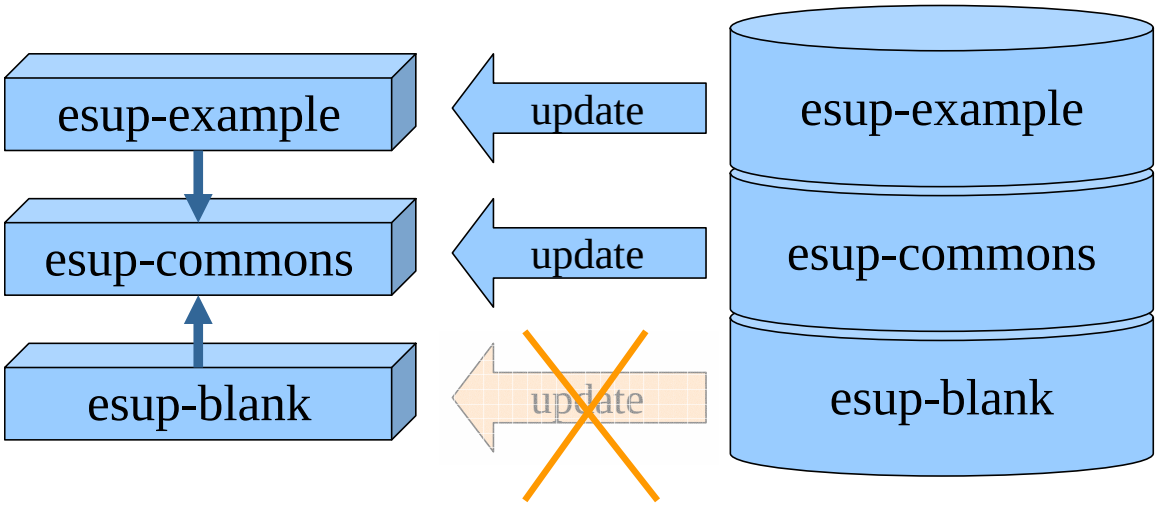


Sur la machine du développeur, le projet *esup-example* dépend du projet *esup-commons*.

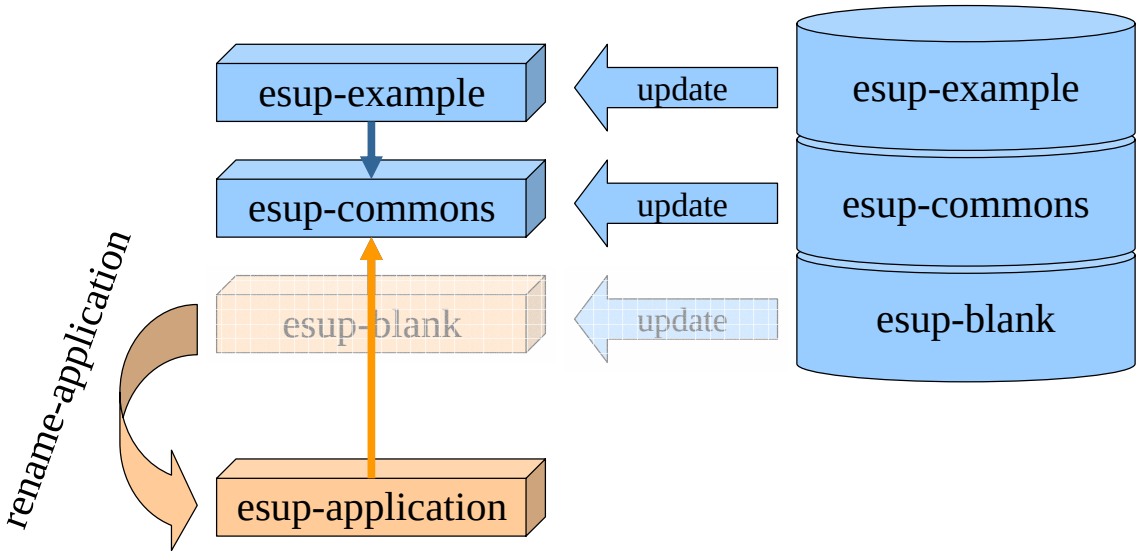
Le sous-projet *esup-blank* est une application blanche qui sert de canevas au démarrage de nouvelles applications. *esup-blank* est importé depuis le dépôt SVN pour constituer la base du nouveau développement :



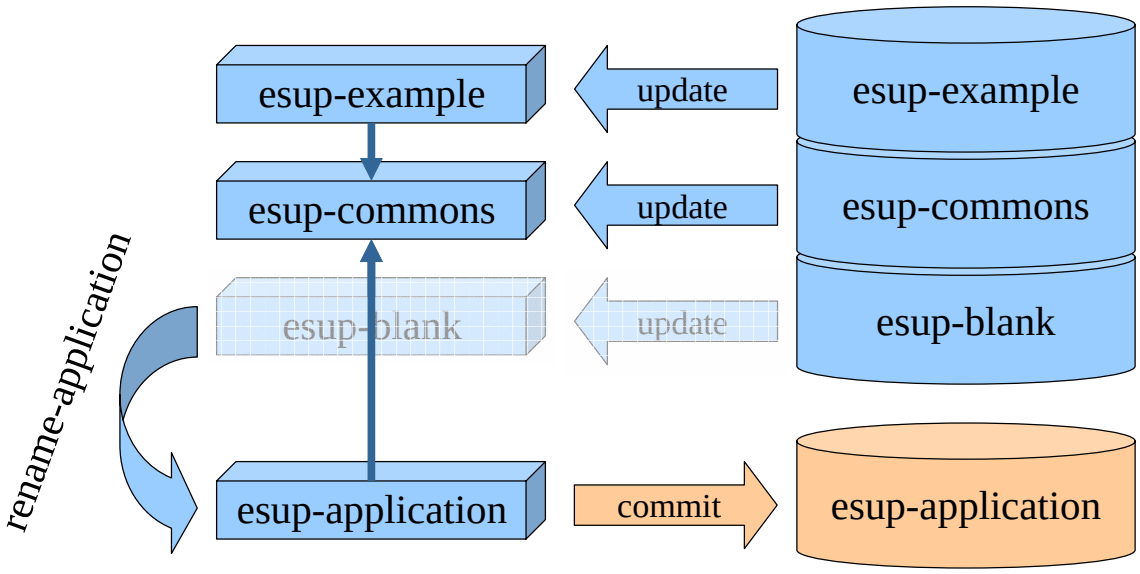
Une fois le projet *esup-blank* créé, il doit être débranché du dépôt SVN pour évoluer indépendamment :



Le projet *esup-blank* doit ensuite être renommé pour donner vie à la nouvelle application, à l'aide de la tâche *ant* **`rename-application`** :



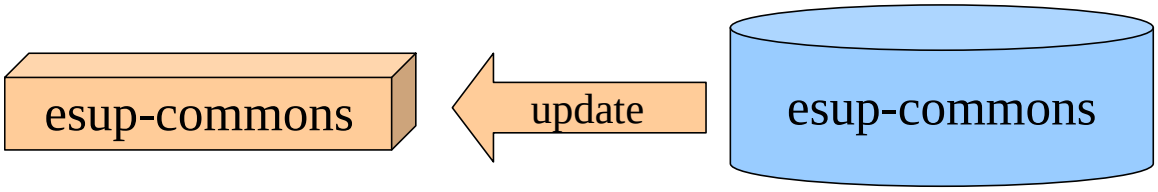
La nouvelle application peut ensuite être sauvegardée vers son propre dépôt SVN :



La suite de ce chapitre montre comment on crée pratiquement le projet *esup-application*.

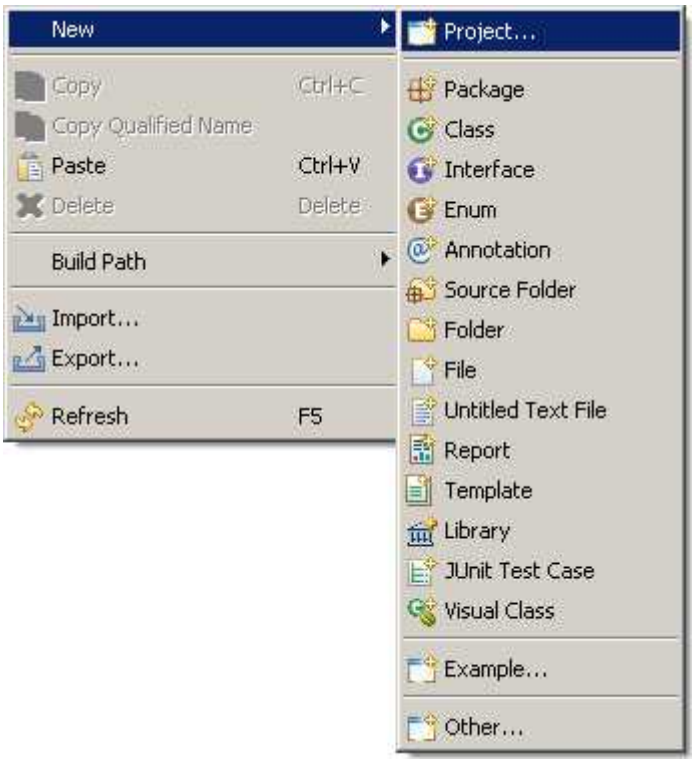
3.4. Création du projet esup-commons

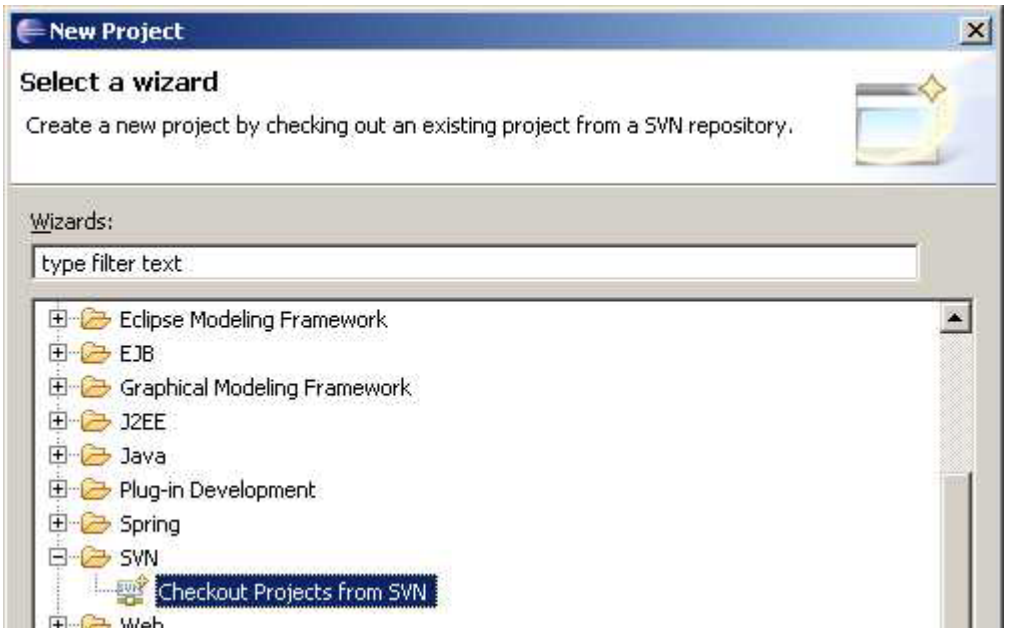
Rapatriement des données à partir du dépôt SVN

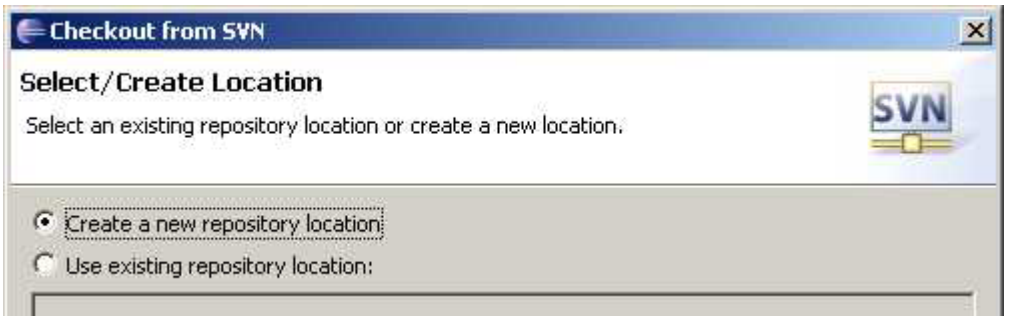


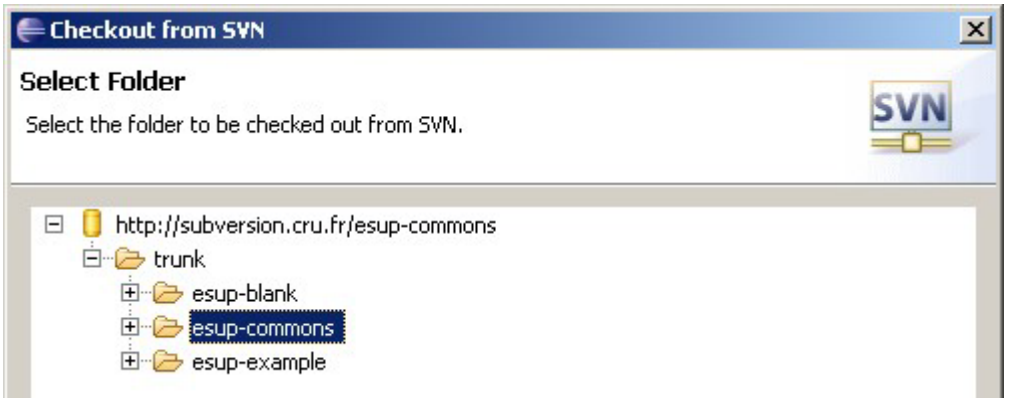
Exercice 5 : Créer le projet *esup-commons* à partir du dépôt SVN

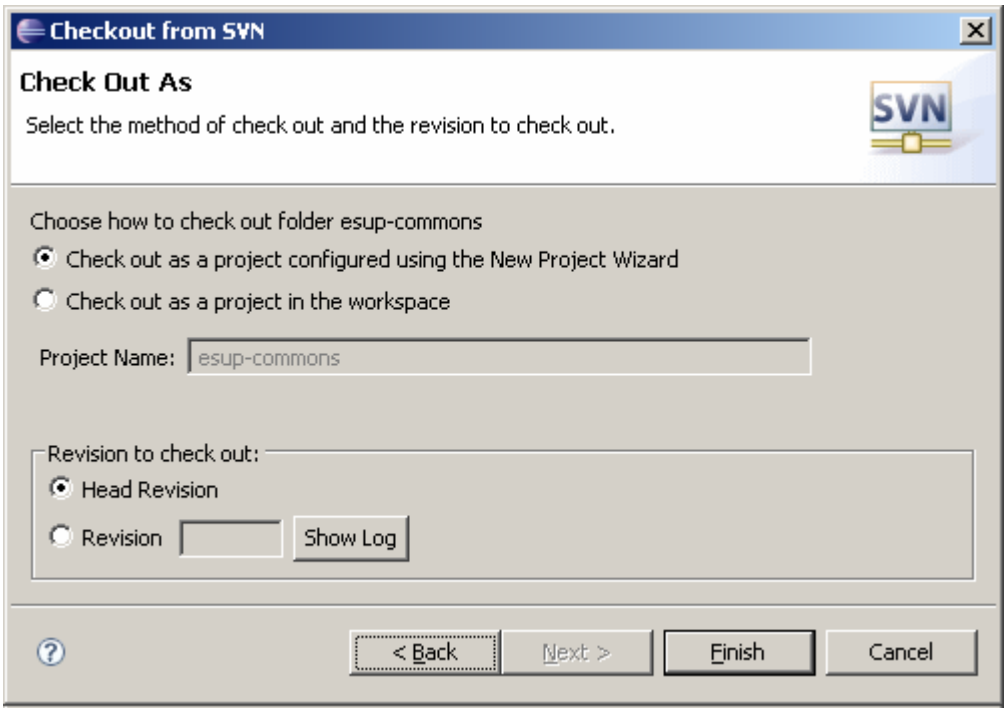
Suivre les instructions des saisies d'écran ci-dessous pour créer le projet *esup-commons*.

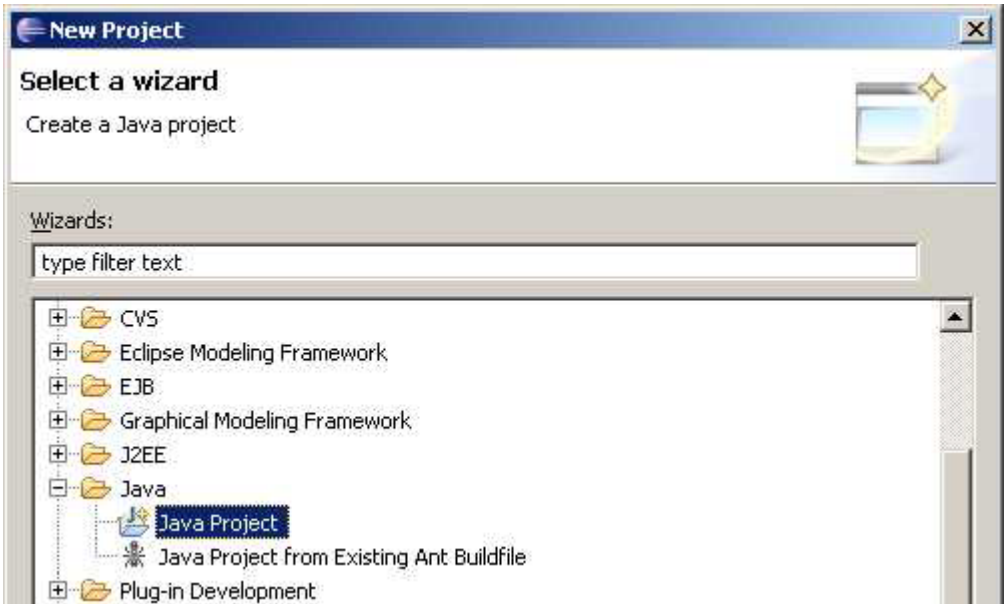




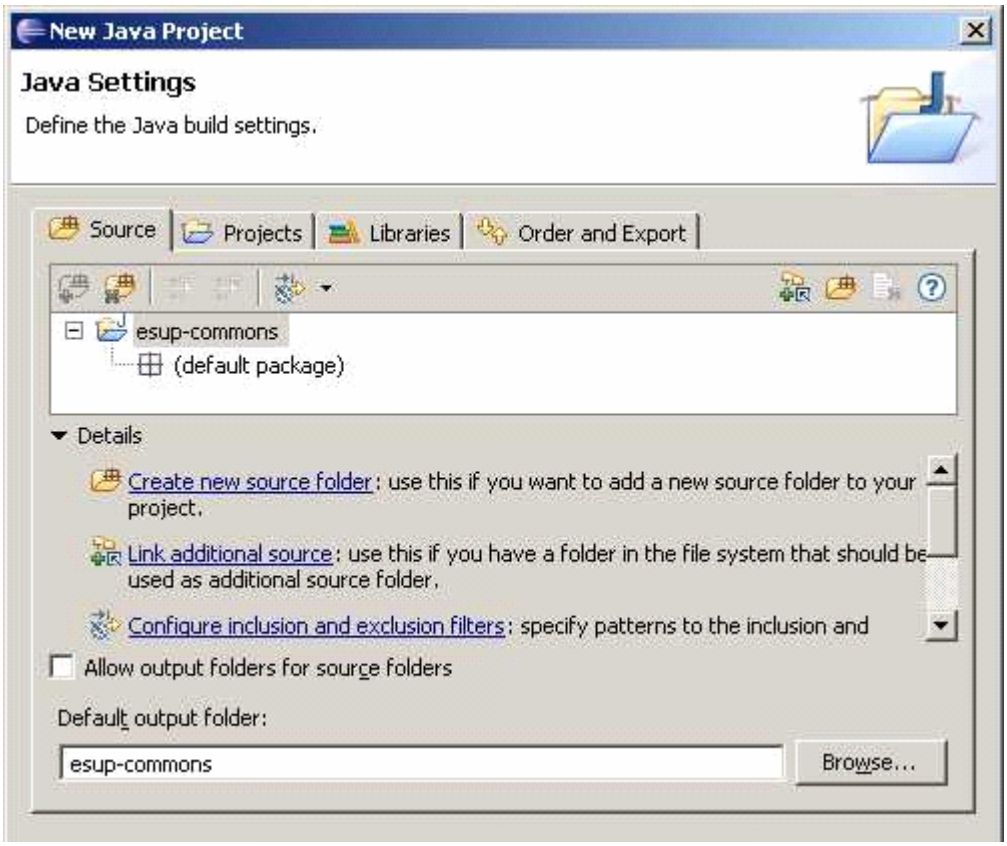




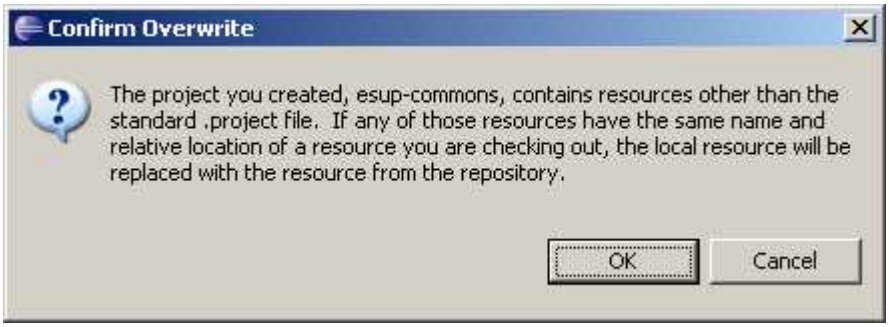


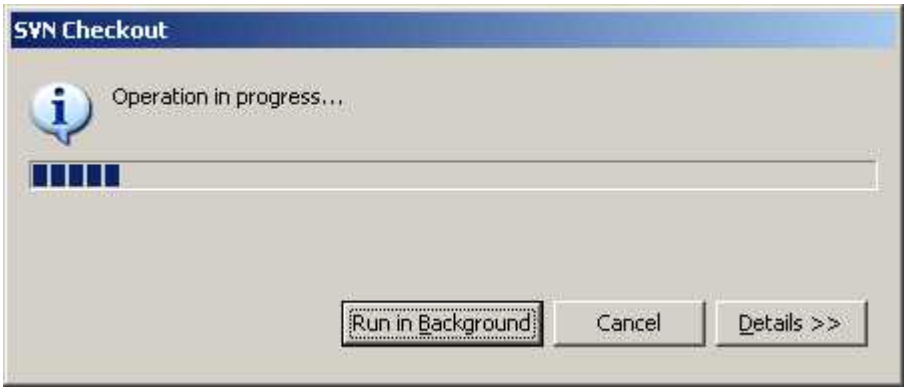


Créer un nouveau répertoire pour le projet (**`c:\devel\esup-commons`**), puis créer le projet lui-même :



Note : à cette étape, il n'y a rien dans le projet, le *build path* sera configuré ultérieurement.



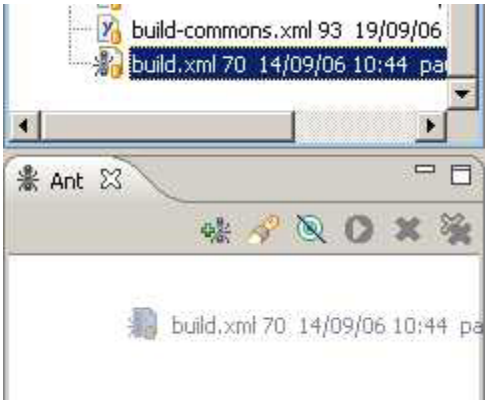


Créer les premiers répertoires

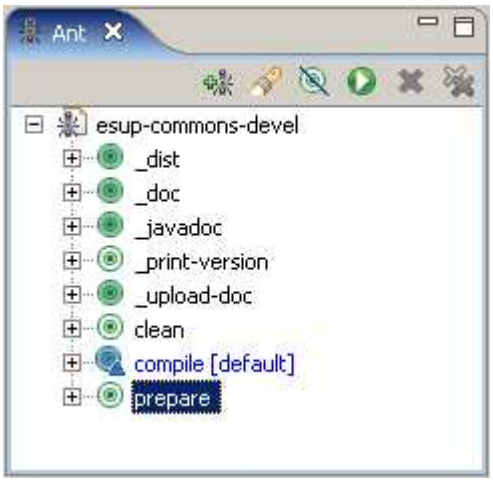
Exercice 6 : Créer les premiers répertoires du projet *esup-commons*

Créer les premiers répertoires du projet *esup-commons* comme indiqué ci-dessous.

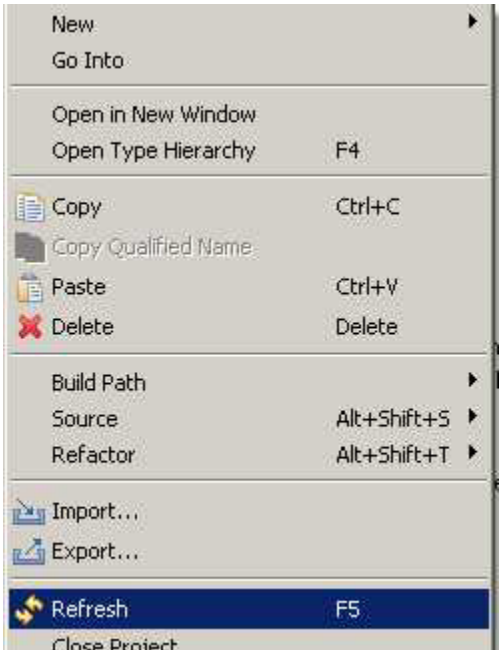
Faire glisser le fichier **`build.xml`** dans la vue ant :



Lancer la tâche **`prepare`** (le répertoire **`/build/WEB-INF/classes`** sera créé, il est nécessaire pour configurer le build path.



Actualiser le projet pour faire apparaître le répertoire **`/build/WEB-INF/classes`** :



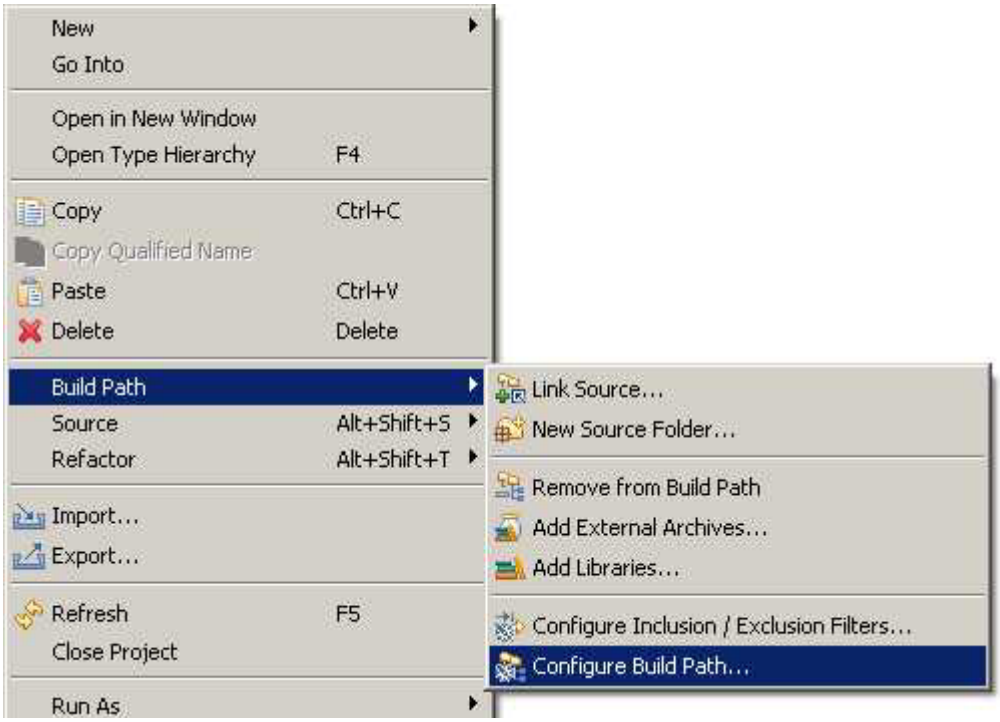
Actualiser la vue *ant* de *Eclipse* pour supprimer d'éventuelles erreurs.

Configuration du répertoire source, du build path et des bibliothèques

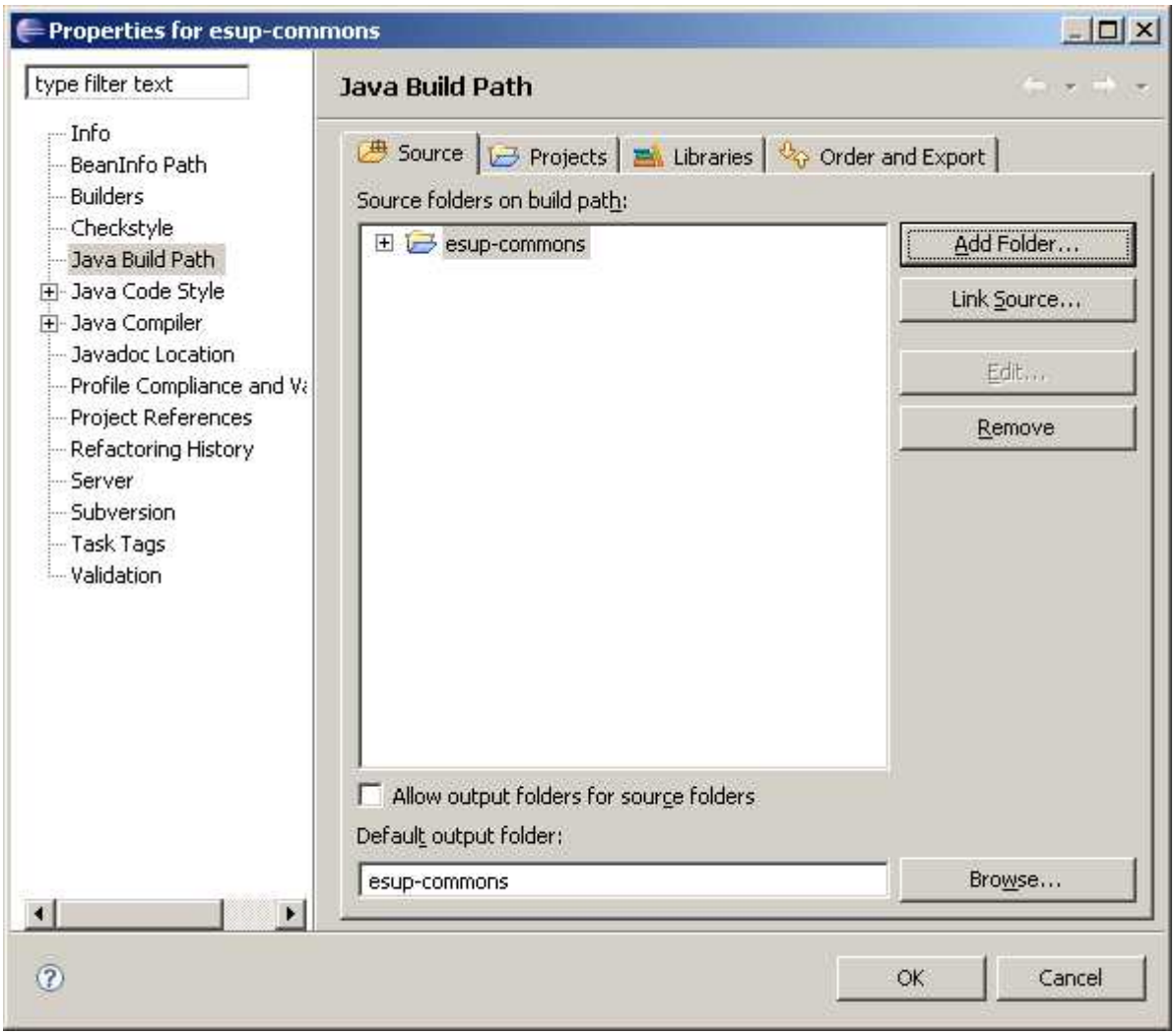
Exercice 7 : Configurer le projet *esup-commons*

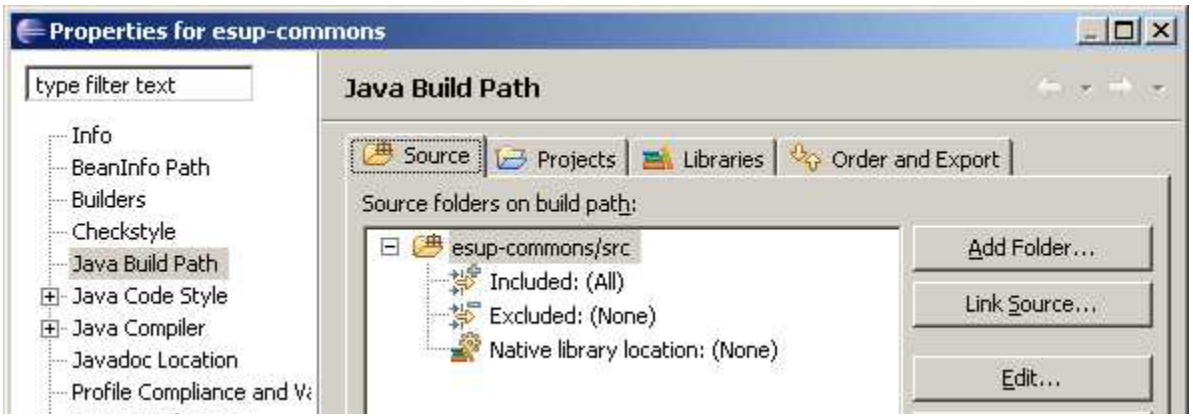
Configurer le projet *esup-commons* comme indiqué ci-dessous.

Maintenant, configurer le *build path* :

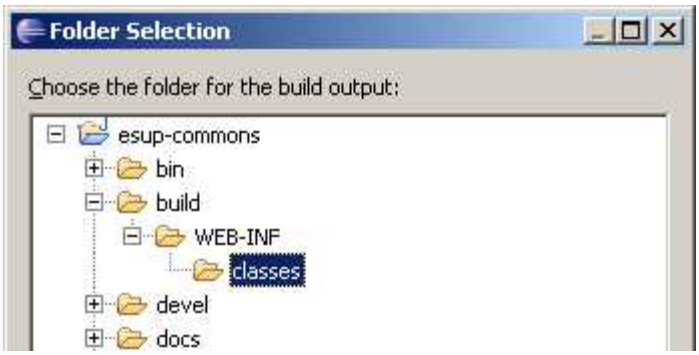


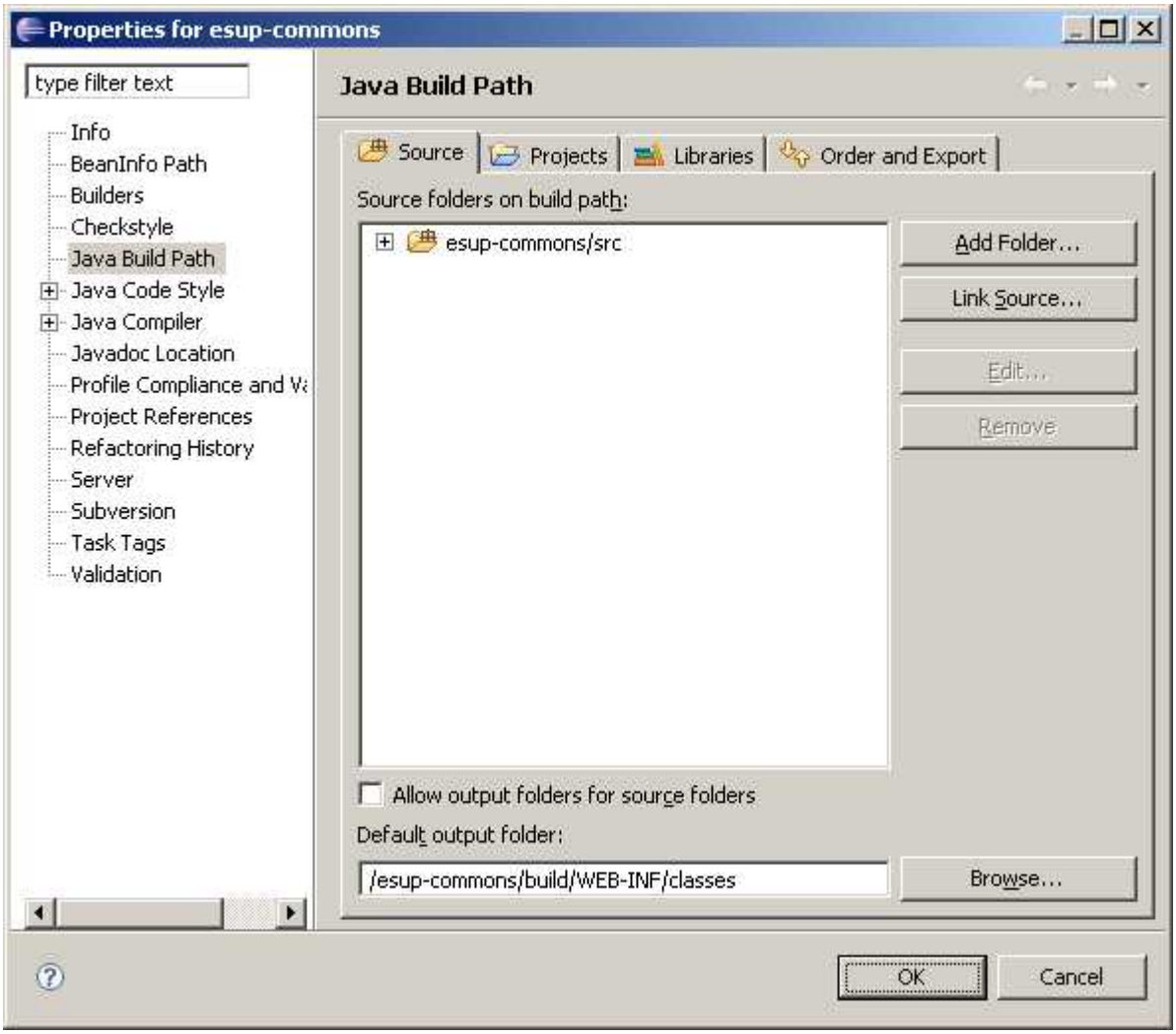
Ajouter le répertoire **`/src`** comme répertoire source du projet :

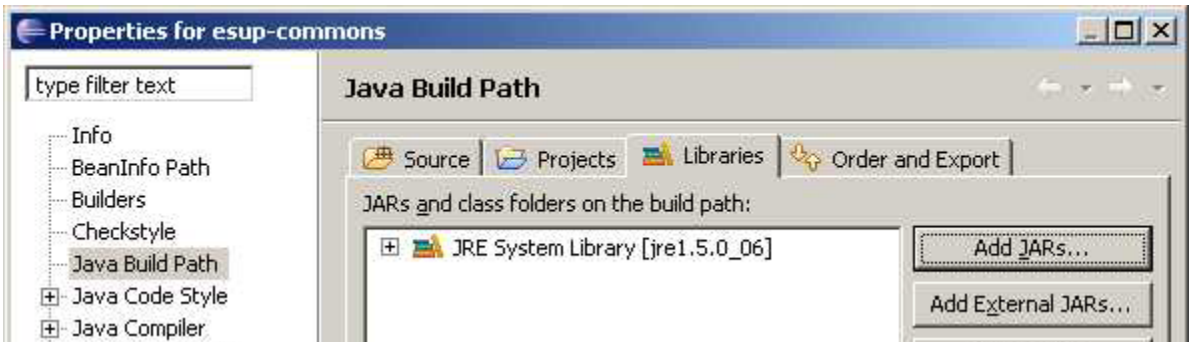




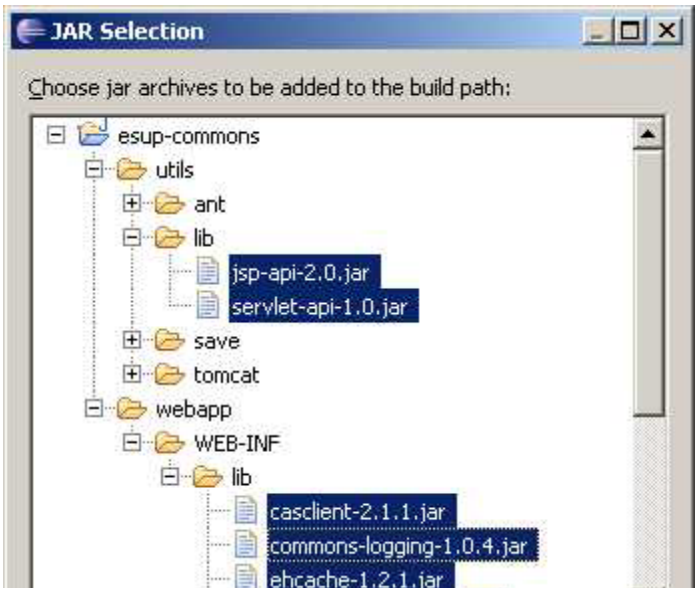
Modifier le répertoire de sortie (utilisé pour la compilation) :



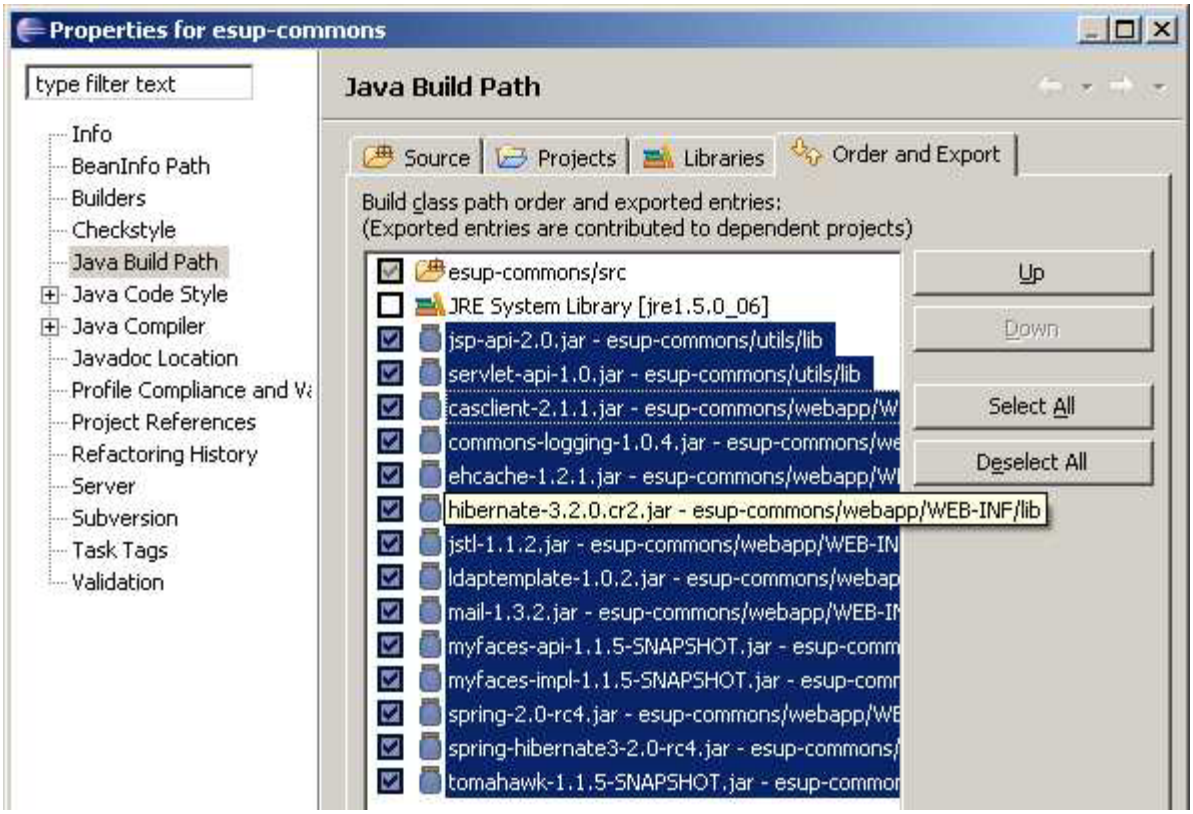




Ajouter toutes les bibliothèques de **`/utils/lib`** et **`/webapp/WEB-INF/lib`** (toutes ne sont pas présentées sur la saisie d'écran ci-dessous) :



Exporter les bibliothèques afin qu'elles soient accessibles aux autres projets en indiquant simplement que ces projets dépendent du projet *esup-commons* :



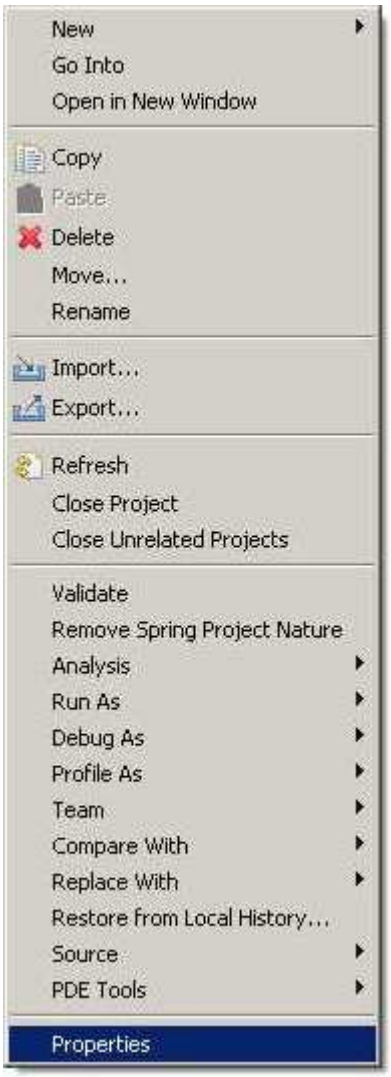
Appliquer les vérifications Checkstyle au projet esup-commons

Note : cette opération sera à appliquer à chaque nouveau projet, juste après sa création.

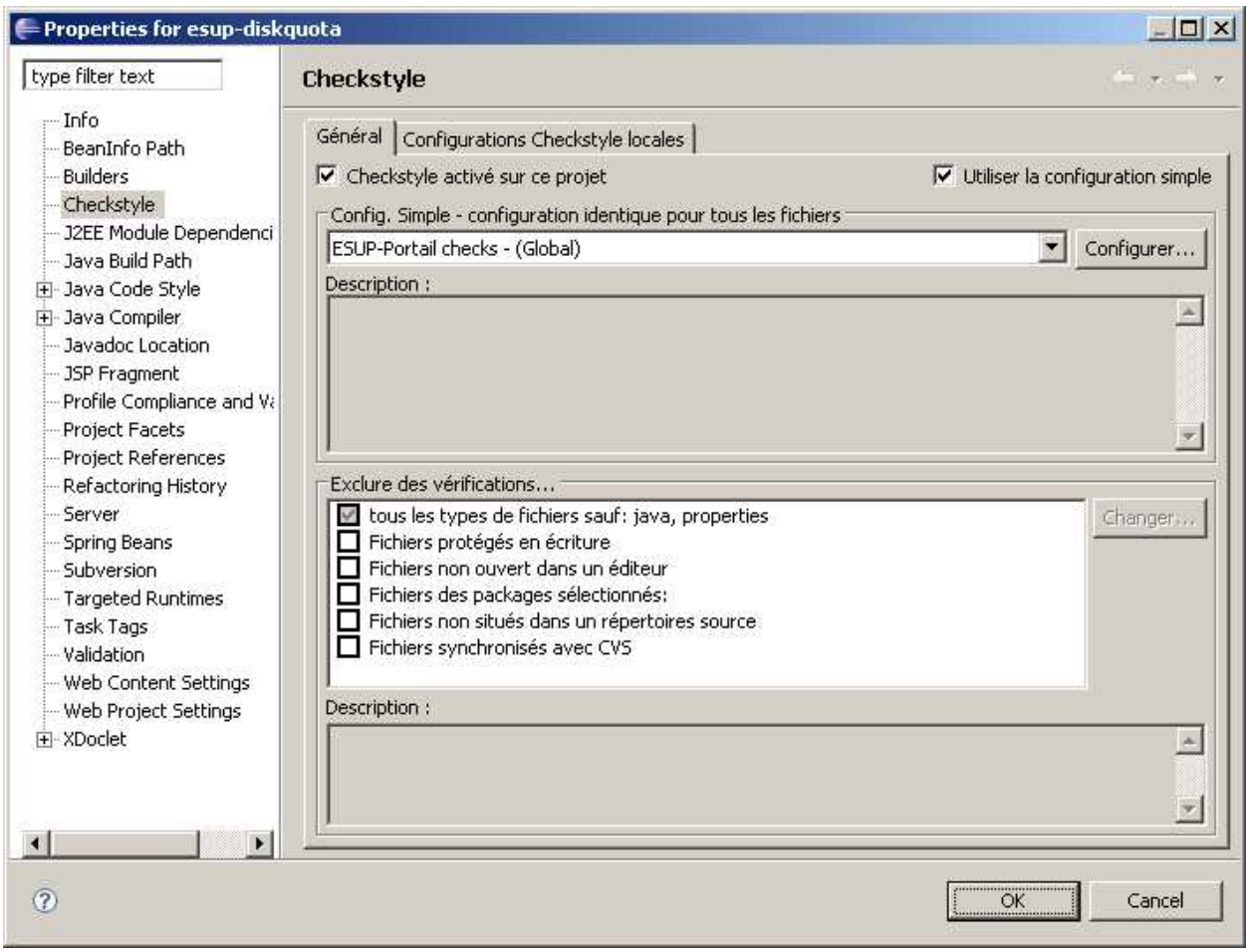
Exercice 8 : Activer *Checkstyle* pour le projet *esup-commons*

Activer *Checkstyle* pour le projet *esup-commons* comme indiqué ci-dessous.

Cliquer avec le bouton droit sur le projet pour éditer ses propriétés :



Activer *Checkstyle* pour le projet :

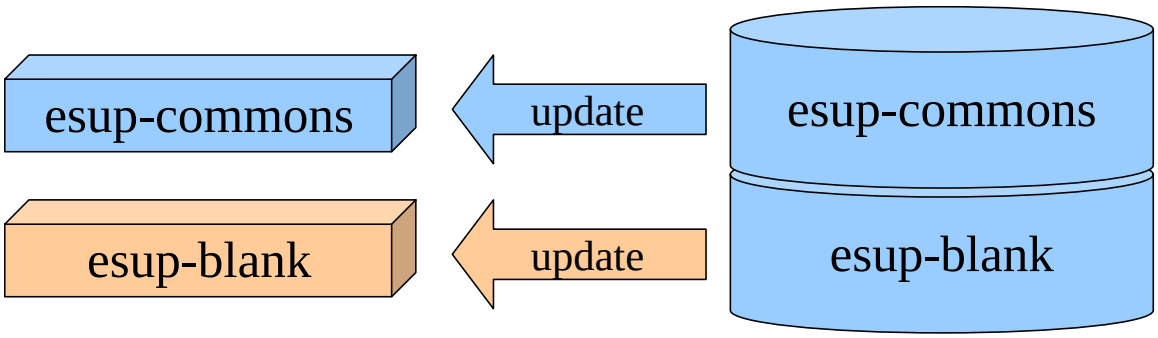


Les messages d'avertissement de *Checkstyle* doivent normalement apparaître dans la vue « Problèmes » d'*Eclipse*.

Votre projet *esup-commons* est prêt à l'utilisation.

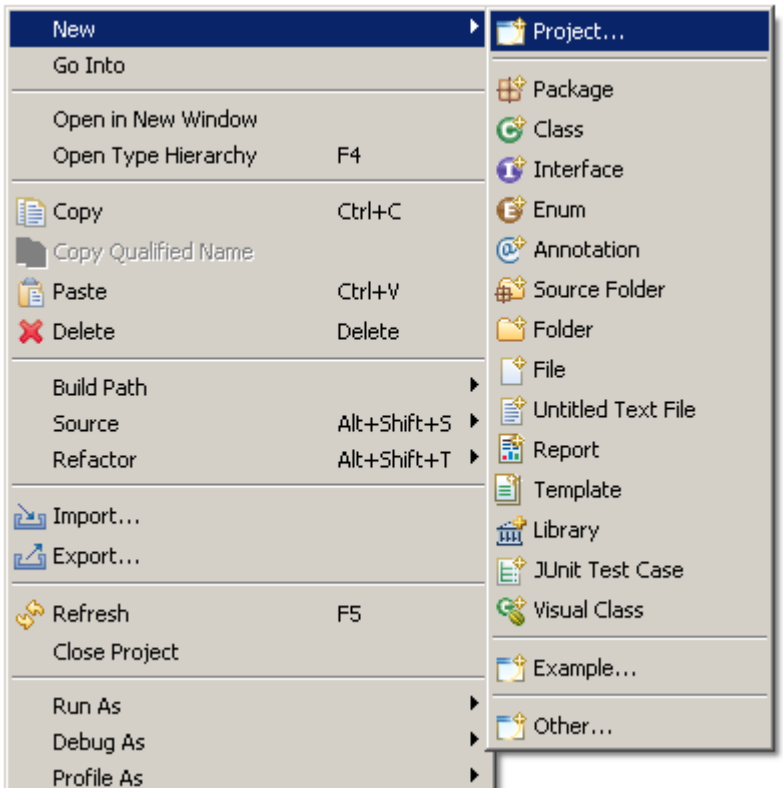
3.5. Création du projet esup-blank

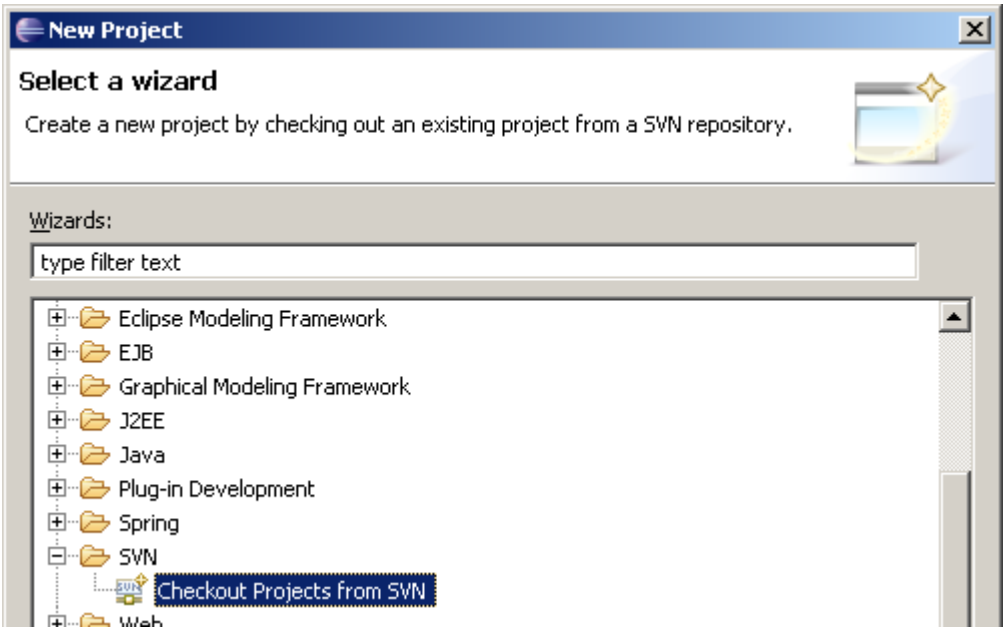
Rapatriement des données à partir du dépôt SVN

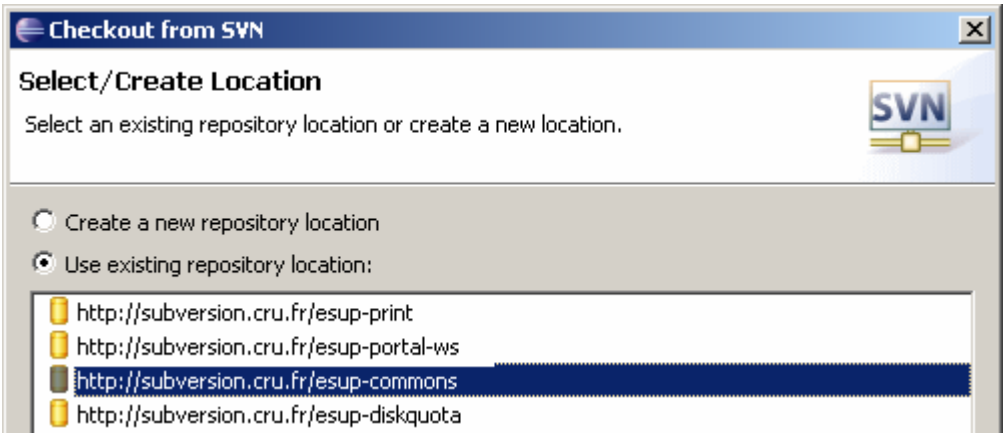


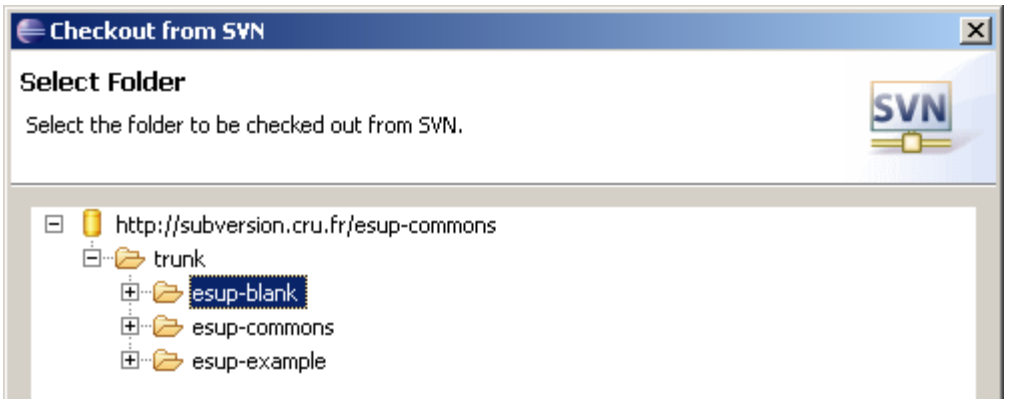
Exercice 9 : Créer le projet *esup-blank* à partir du dépôt SVN

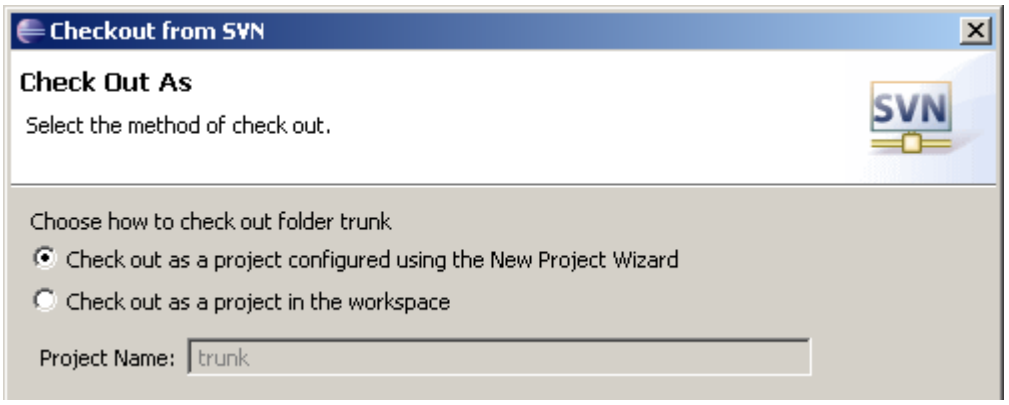
Suivre les instructions des saisies d'écran ci-dessous pour créer le projet *esup-blank*.

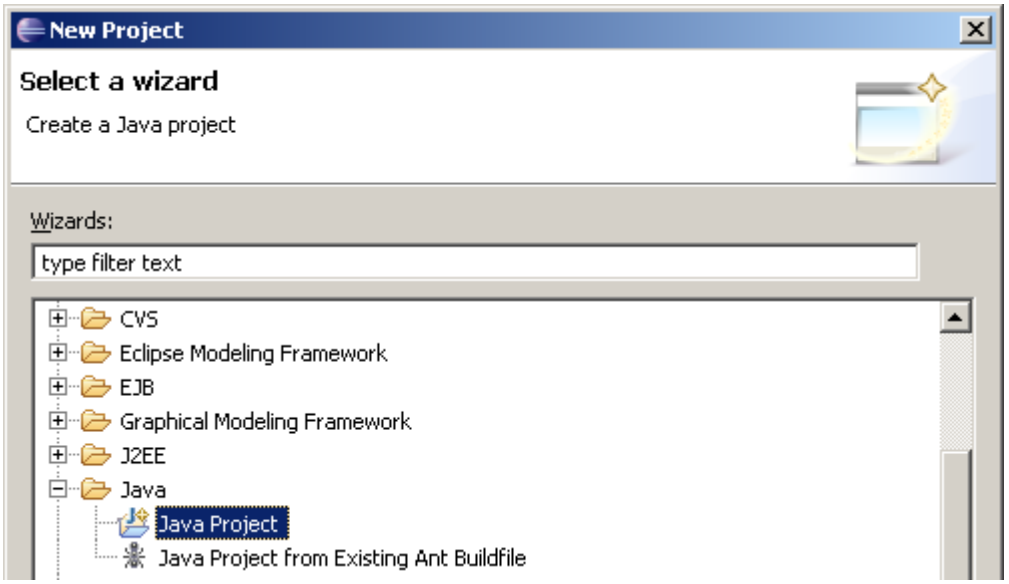




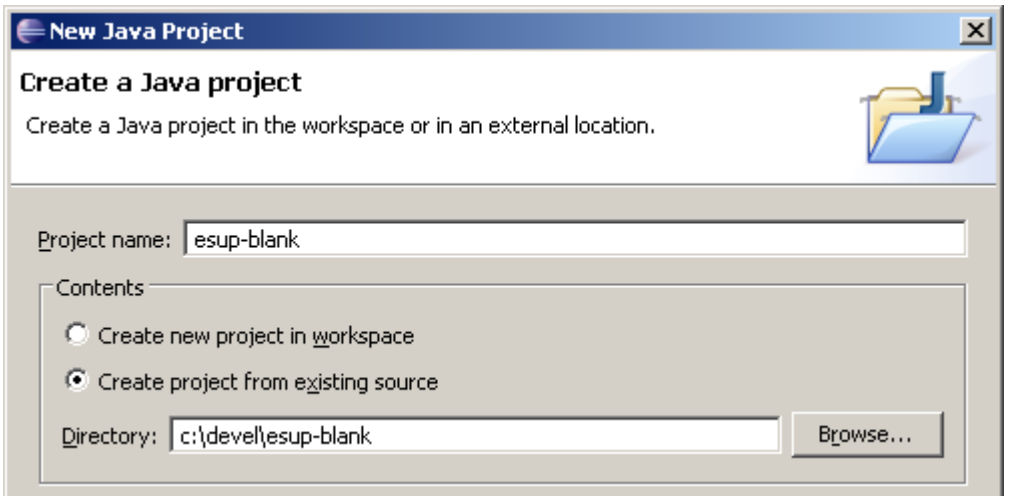


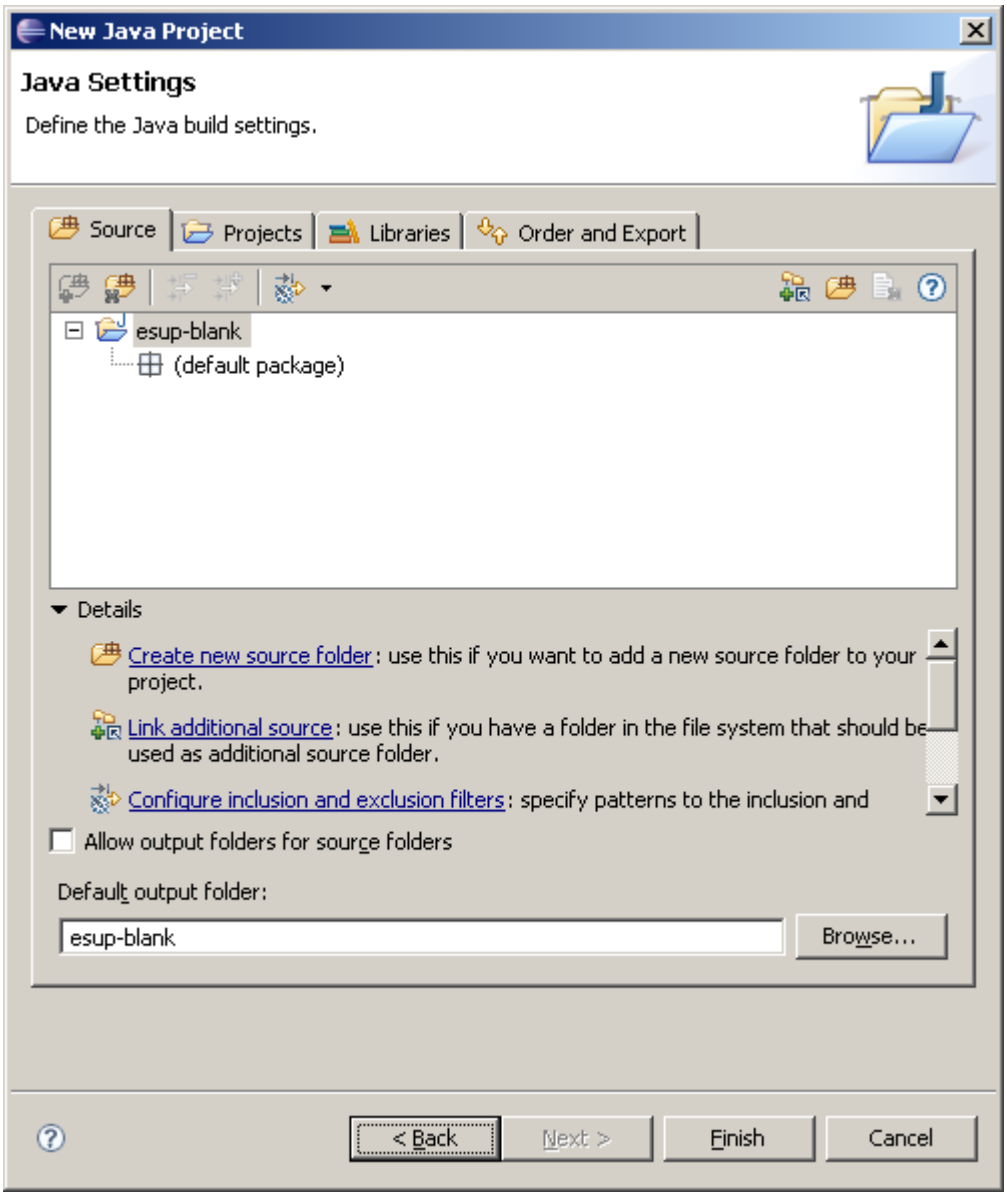


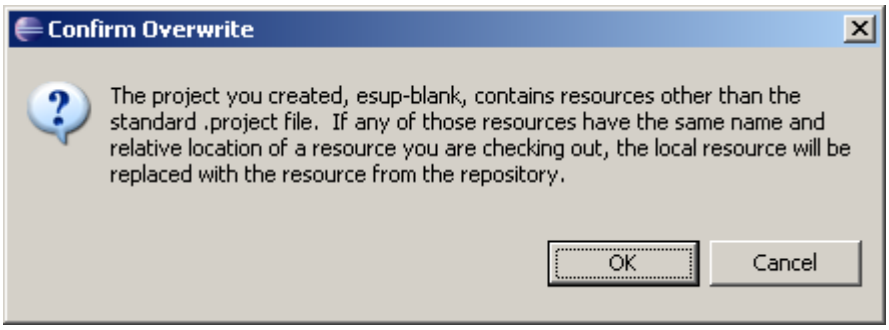


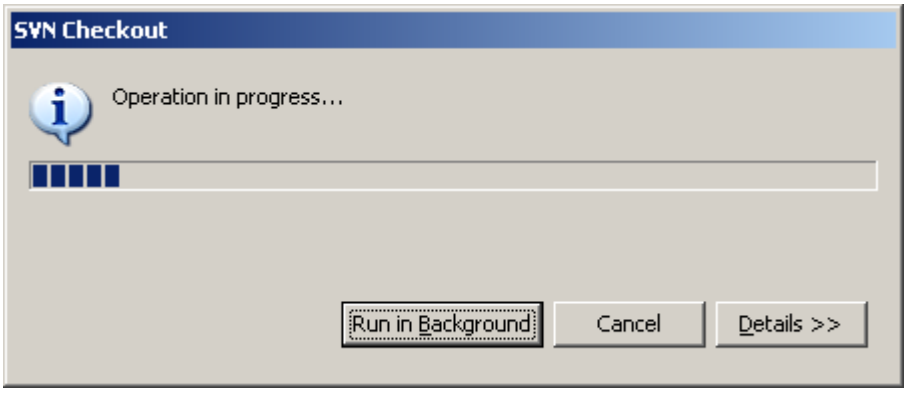


Créer un nouveau répertoire (**`c:\devel\esup-blank`**) puis créer le projet lui-même :





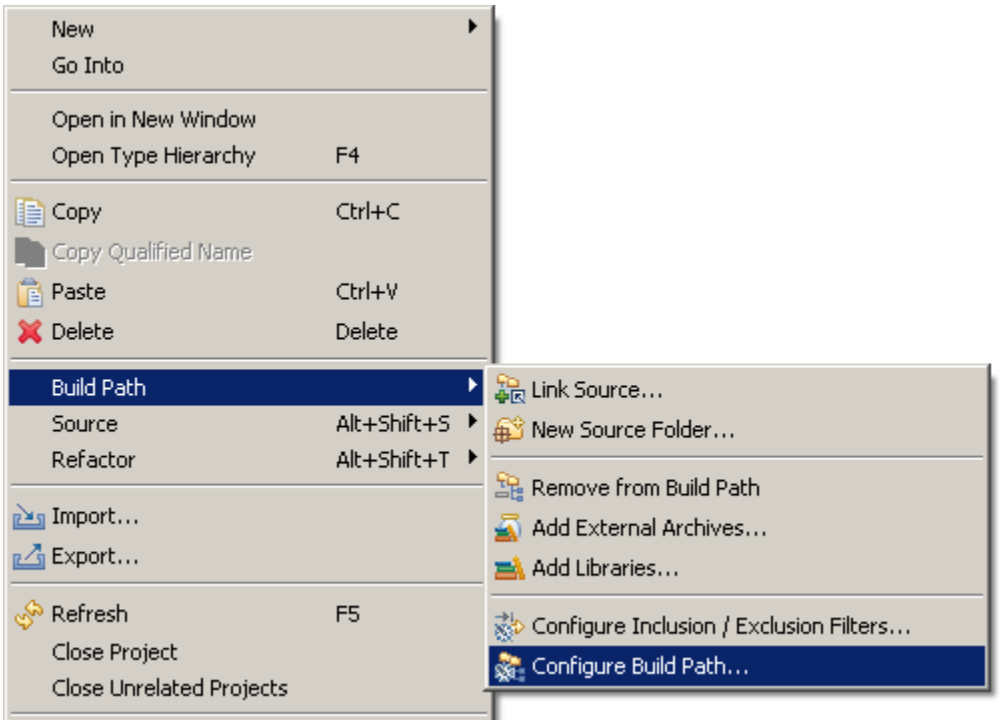


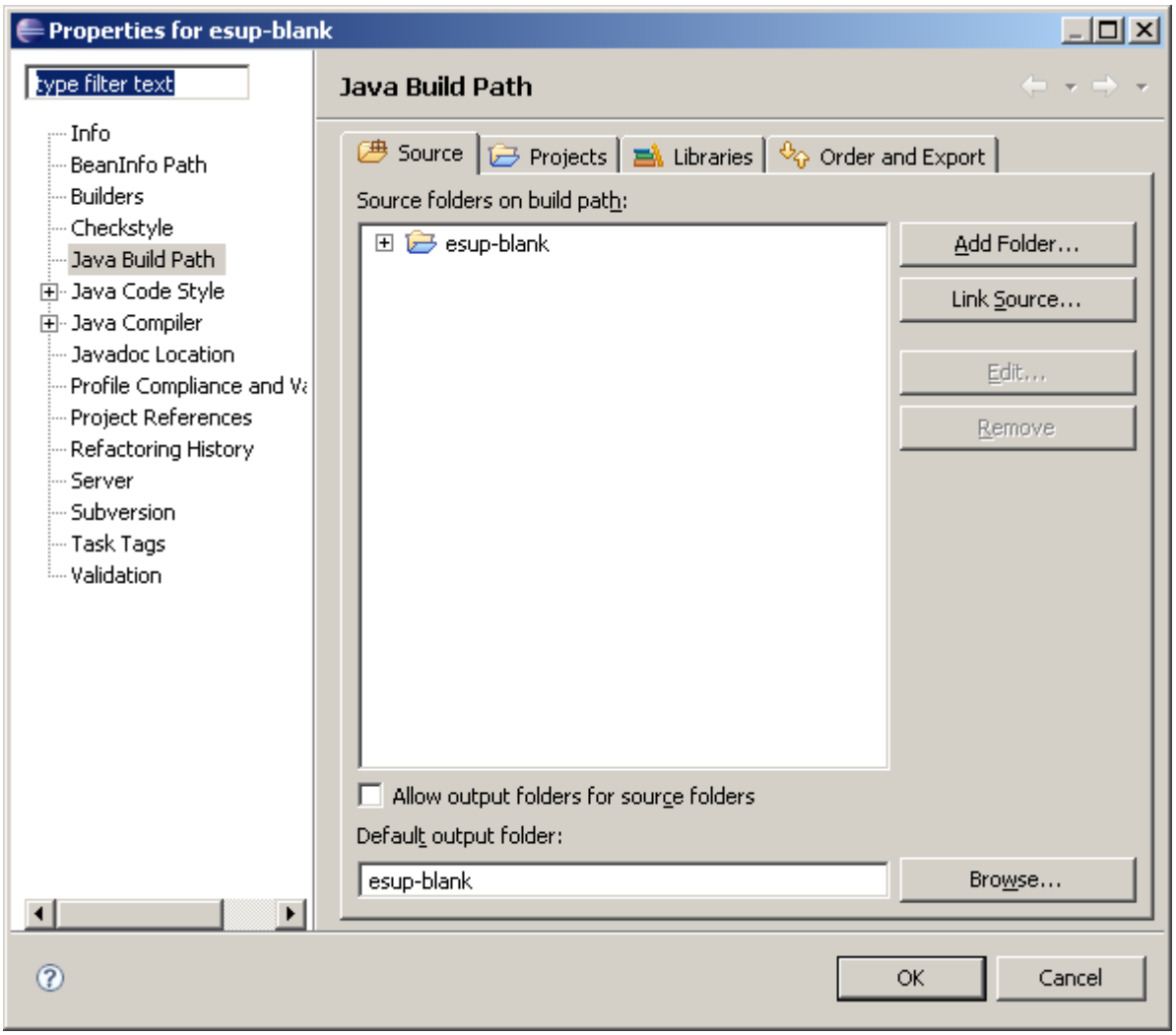


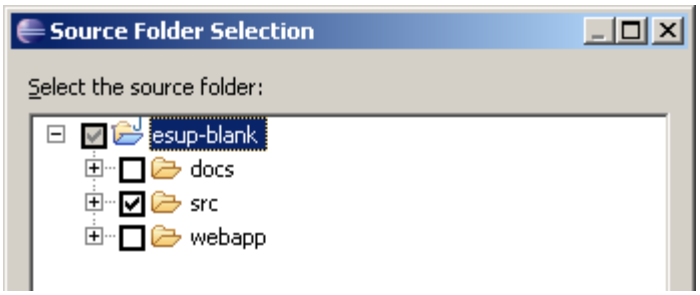
Configuration du projet

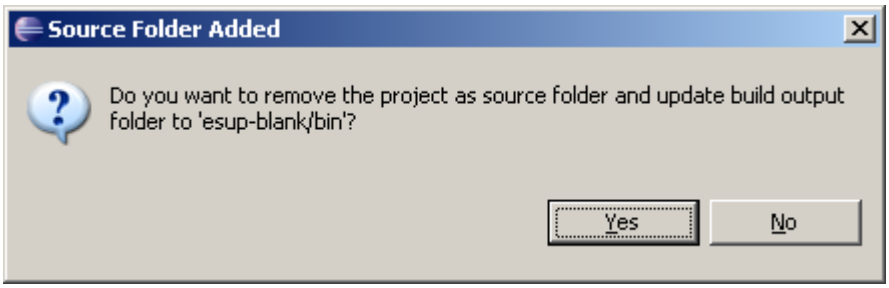
Exercice 10 : Configurer le projet *esup-blank*

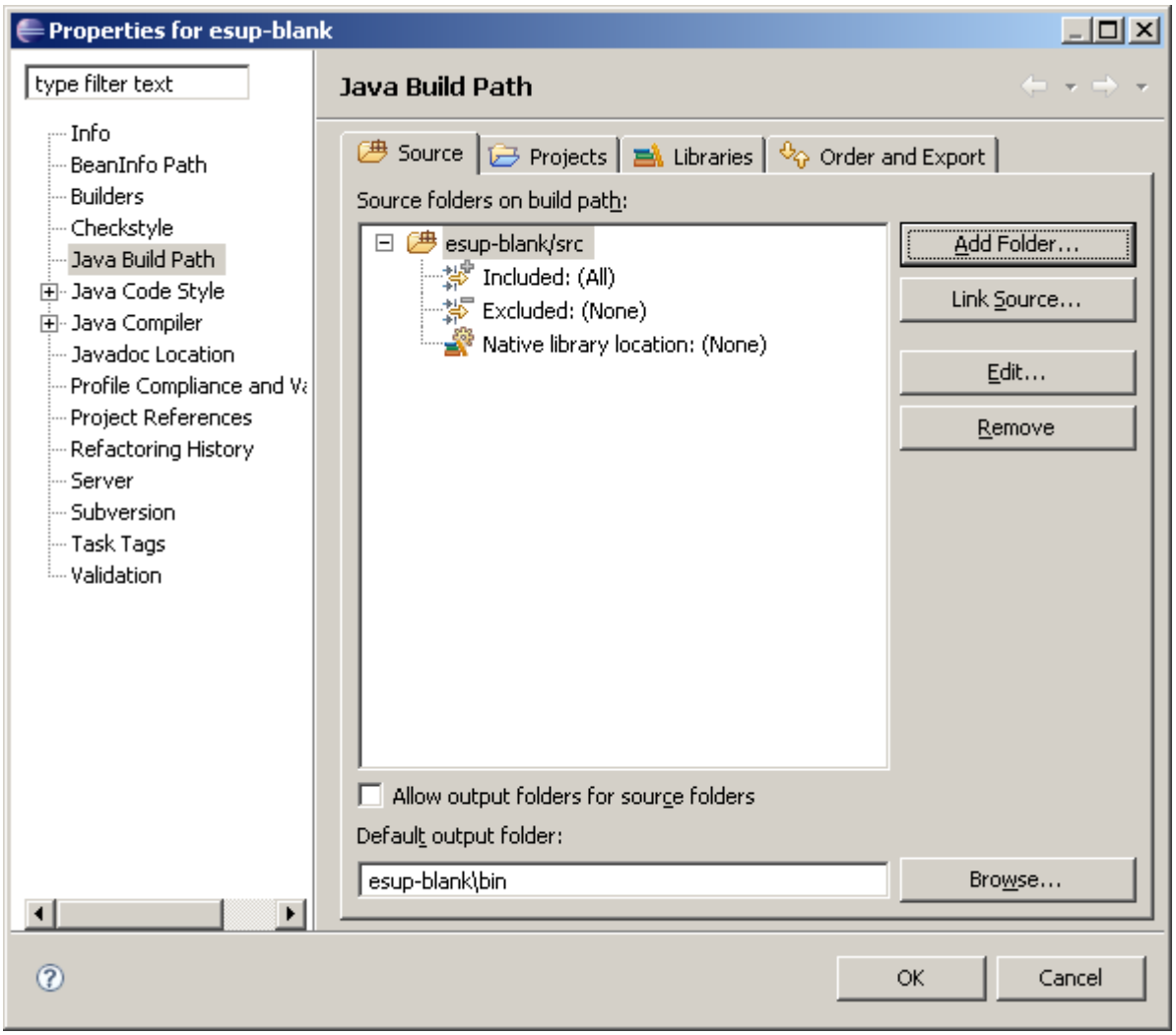
Configurer le projet *esup-blank* comme indiqué ci-dessous.



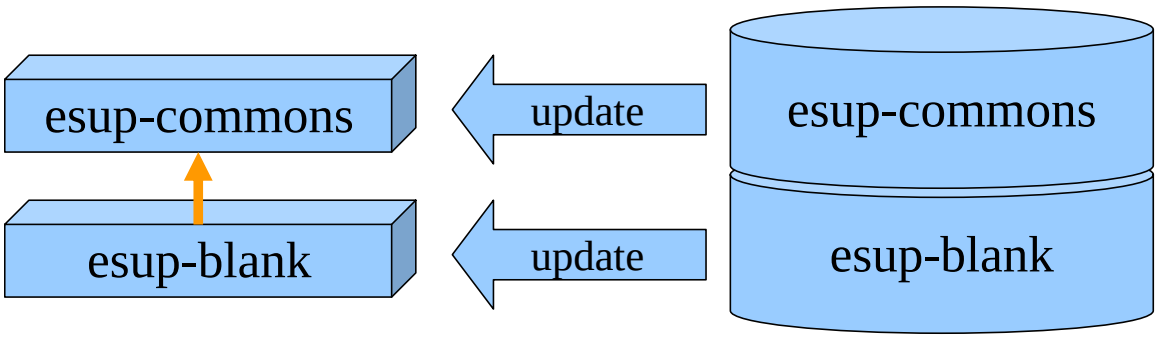


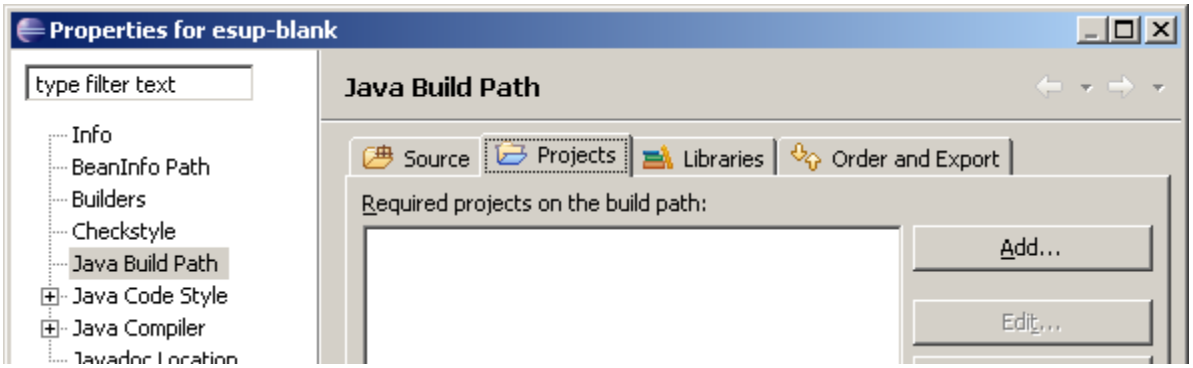


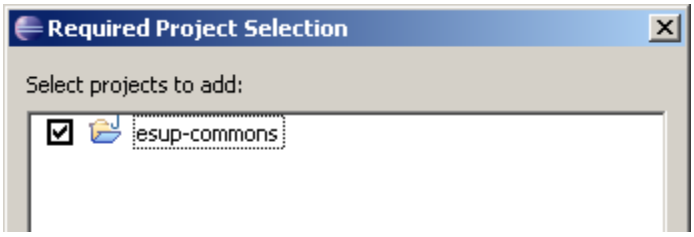


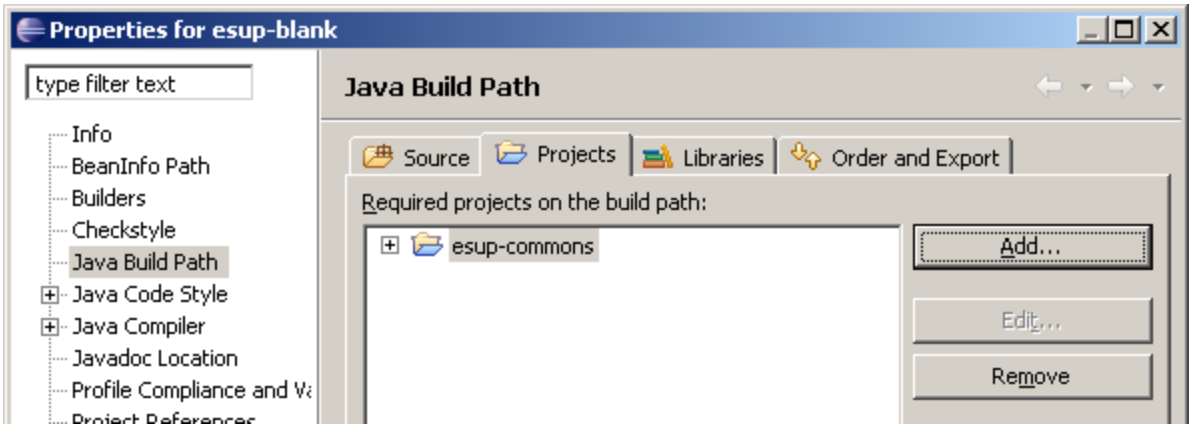


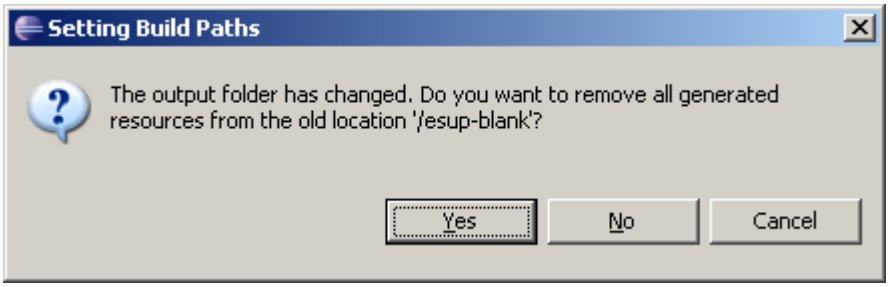
Rendre le projet esup-blank dépendant du projet *esup-commons*.



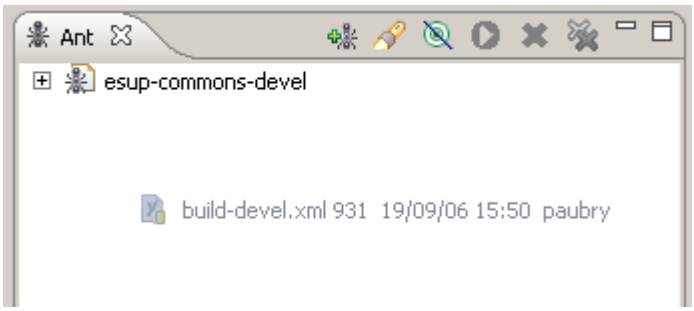






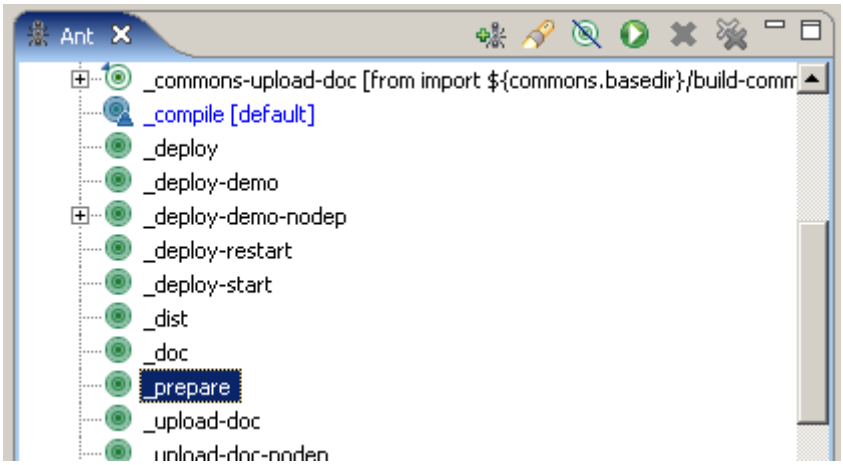


Faire glisser le fichier build-devel.xml dans la vue ant :



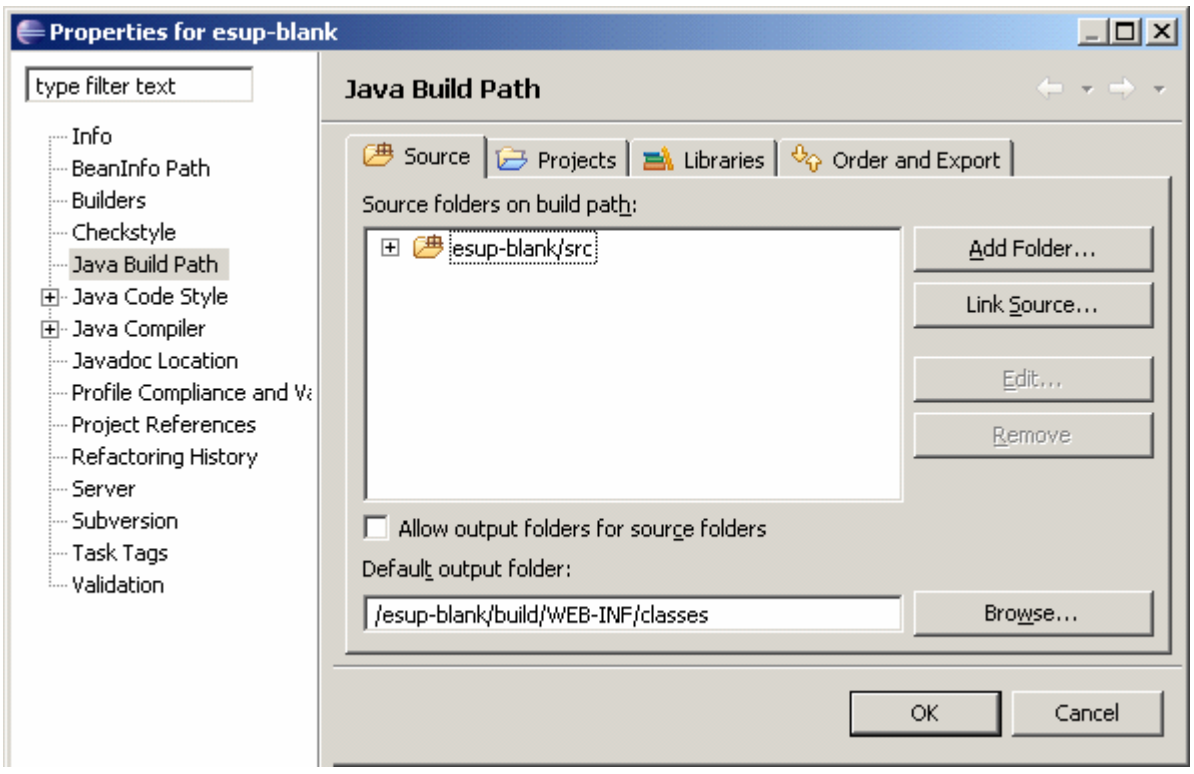
Créer les fichiers **`build.properties`** et **`build-devel.properties`** à partir des fichiers d'exemple fournis **`build-quick-start-example.properties`** et **`build-devel-example.properties`** (la tâche **`_prepare`** ne fonctionnera pas sans ces deux fichiers), puis lancer la tâche **`_prepare`** pour créer le répertoire **`/build/WEB-INF/classes`**.

Note : on n'indiquera seulement **`quick-start=true`** dans le fichier **`build.properties`**, des détails sur le déploiement en *quick-start* sont donnés plus loin dans ce document.



Actualiser le projet et indiquer le répertoire **`/build/WEB-INF/classes`** comme répertoire de sortie du projet.

Actualiser la vue *ant* de *Eclipse* pour supprimer d'éventuelles erreurs.



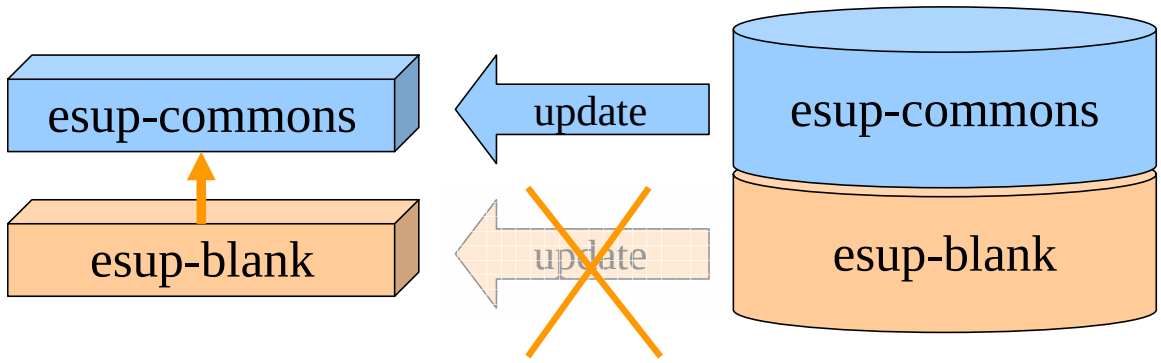
Si vous êtes un développeur du projet *esup-commons* (et donc du projet *esup-blank*), vous pouvez commencer à contribuer.

Activer *Checkstyle* pour le projet *esup-blank* comme fait précédemment pour le projet *esup-commons*.

Si vous avez en projet le développement d'une nouvelle application à partir de *esup-commons*, vous pouvez maintenant :

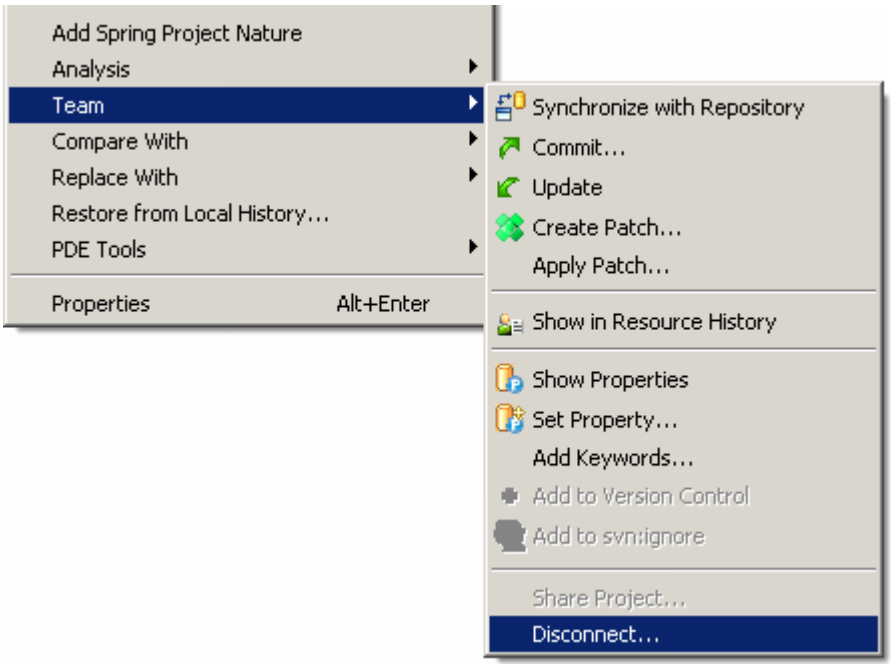
1. Déconnecter *esup-blank* du dépôt SVN,
2. Le renommer,
3. Le reconnecter à votre propre dépôt SVN.

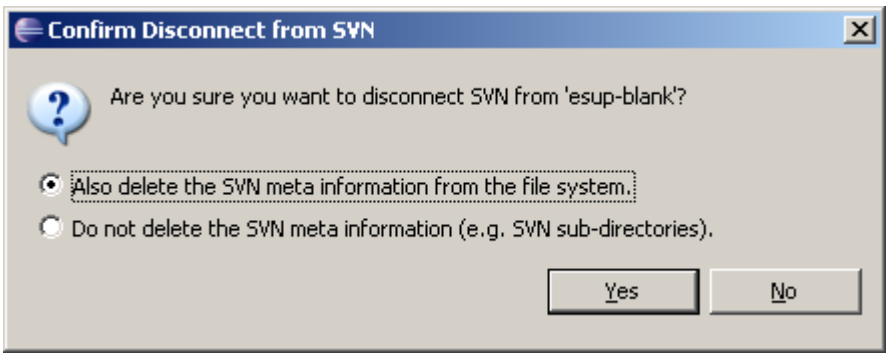
Déconnecter esup-blank du depot SVN

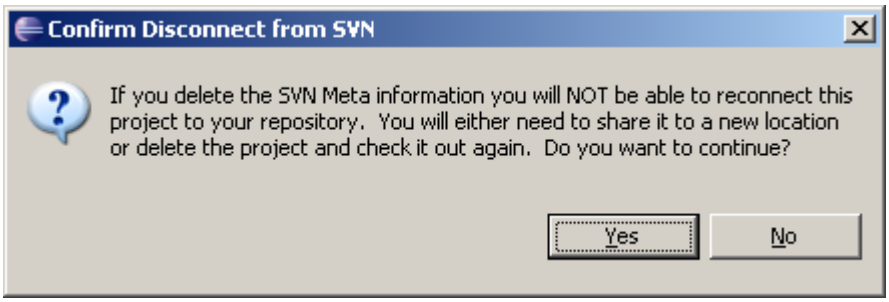


Exercice 11 : Déconnecter le projet *esup-blank* du dépôt SVN

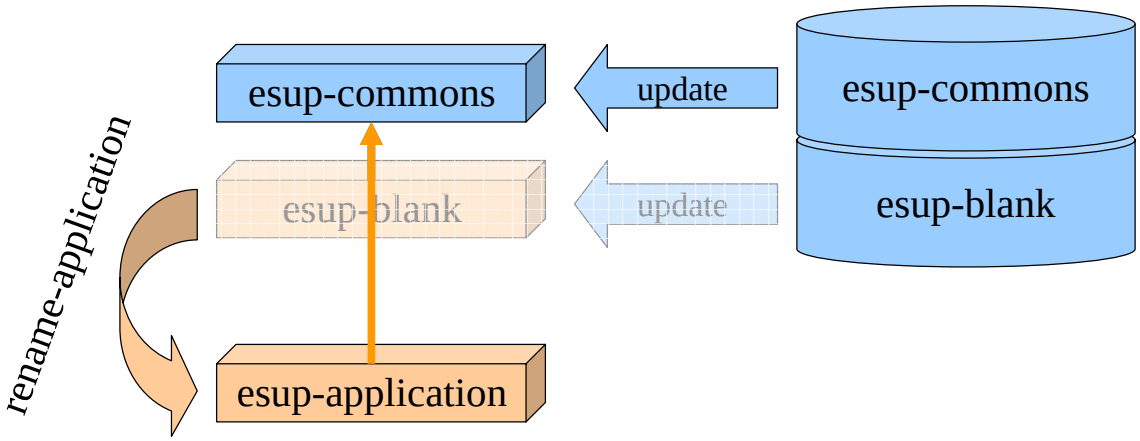
Déconnecte le projet *esup-blank du dépôt SVN* comme indiqué ci-dessous.







Renommer le projet



Exercice 12 : Renommer le projet *esup-blank*

Renommer le projet *esup-blank* comme indiqué ci-dessous.

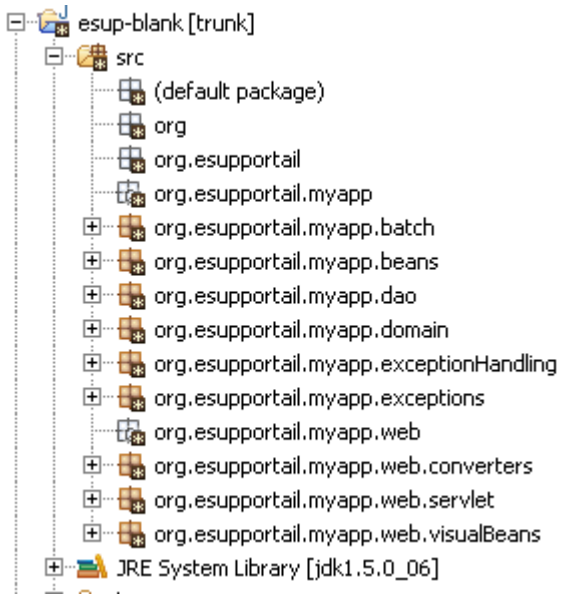
Copier le fichier **`renameApplication-example.properties`** en **`renameApplication.properties`**.

Éditer le fichier **`renameApplication.properties`** :

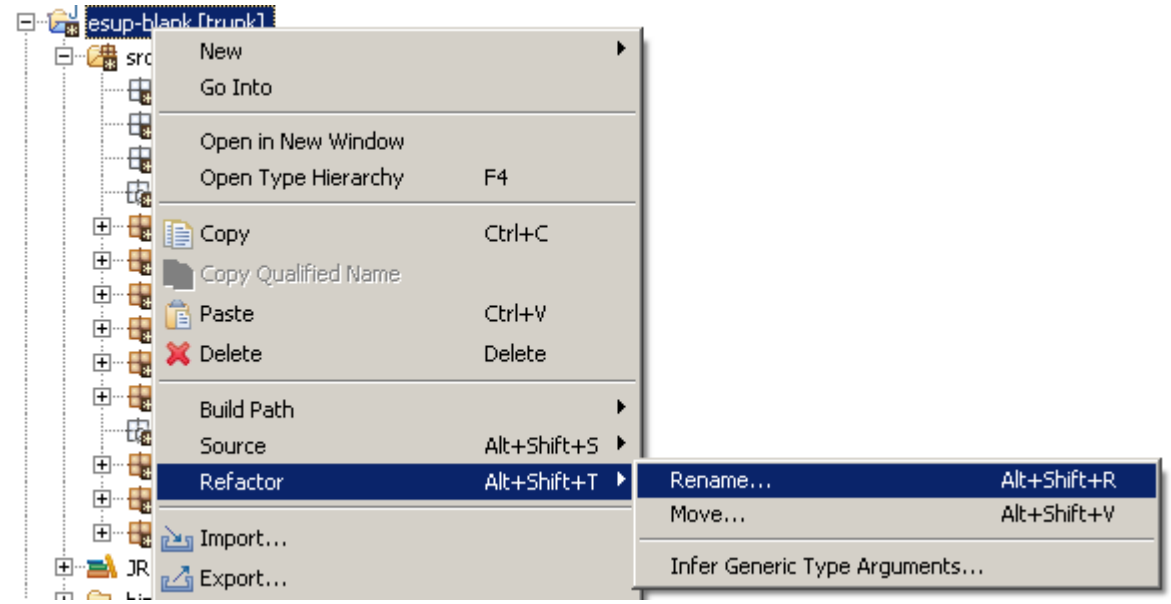
```
org/esupportail/blank=org/esupportail/myapp
org.esupportail.blank=org.esupportail.myapp
edu/domain/blank=edu/domain/myapp
ESUP-Portail\ Blank\ Application=ESUP-Portail My Application
project\ name\="esup-blank=project name="esup-myapp
logo_blank=logo_myapp
esup-blank=esup-myapp
```

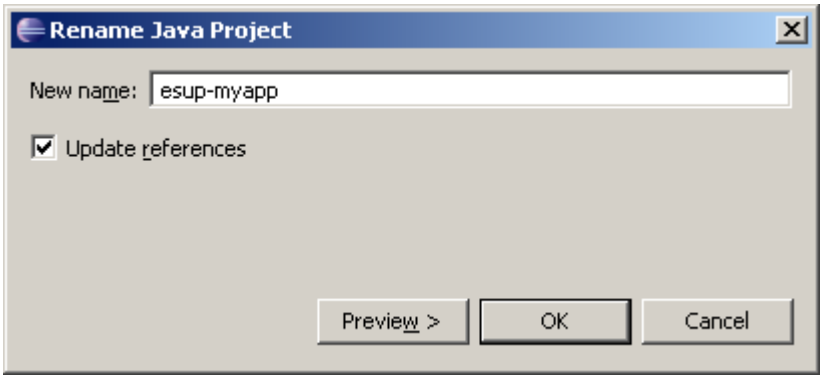
Lancer la tâche **`_rename-application`**, puis rafraîchir le projet.

La tâche **`_rename-application`** ne doit être lancée qu'une seule fois. Soyez sûr de vos modifications dans le fichier **`renameApplication.properties`** avant de lancer la tâche !



Renommer le projet Eclipse :

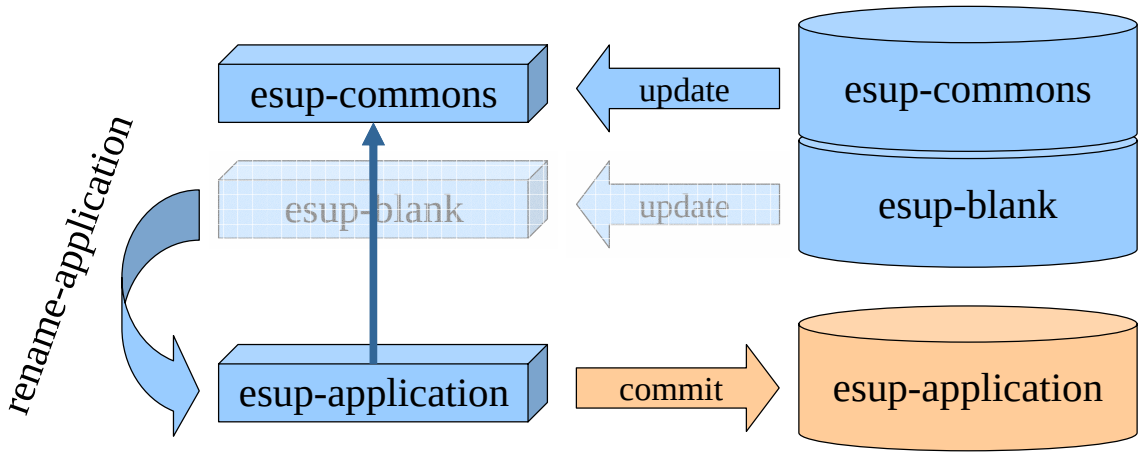




Finalement renommer le répertoire de base du projet **`c:/devel/esup-blank`** (par exemple **`c:/devel/esup-myapp`**).

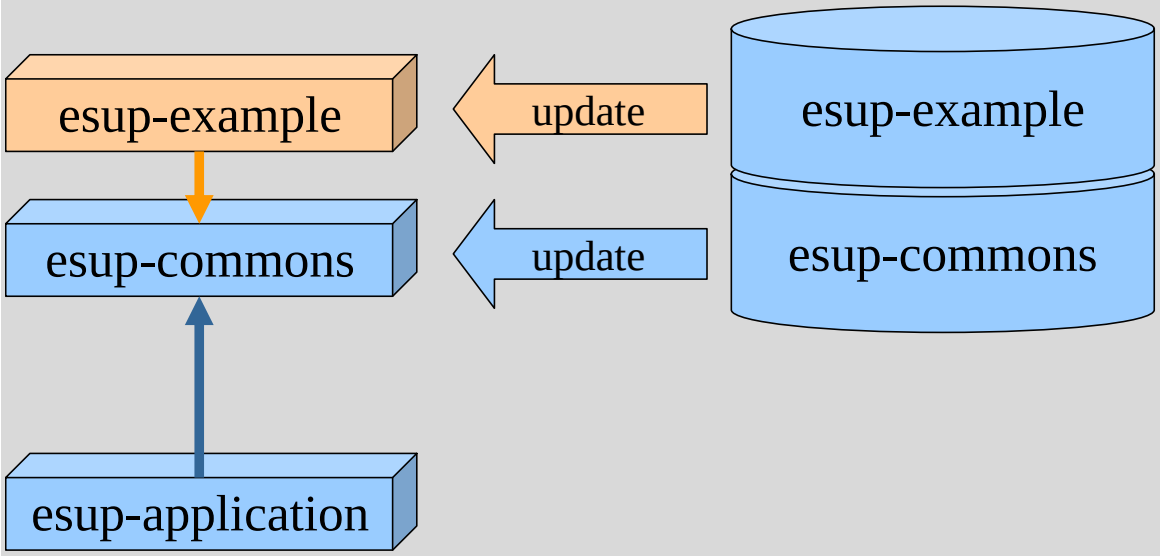
Note : le *refactoring* d'*Eclipse* se charge de cette opération si l'on utilise le *workspace* par défaut.

Le projet peut maintenant être connecté à un autre dépôt *SVN* et le développement proprement dit peut commencer ;-) :



Exercice 13 : créer et configurer le projet *esup-example*

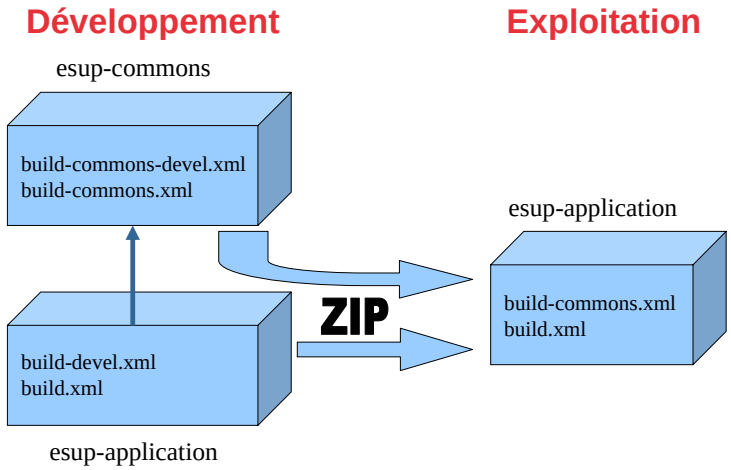
Créer et configurer le projet *esup-example* en suivant la même démarche. Le projet *esup-example* pourra être utilisé tout au long de la formation pour voir quelles sont les fonctionnalités de *esup-commons* et comment elles marchent.



3.6. Du développement à l'exploitation

Comme vu précédemment, le développement d'une application bâtie sur *esup-commons* (sous *Eclipse*) se base sur deux projets *Eclipse*, *esup-commons* et (typiquement) *esup-application*.

Le développeur, lorsqu'il veut exporter son application, en fournit un paquetage compressé (la fabrication de ce paquetage compressé est expliquée en détail dans un chapitre ultérieur) :



Une fois décompressé, ce paquetage se présente (contrairement au mode développement) sous la forme d'une hiérarchie de fichiers unique : les fichiers de la hiérarchie esup-commons sont intégrés avec ceux de l'application pour ne former qu'une seule hiérarchie de fichiers.

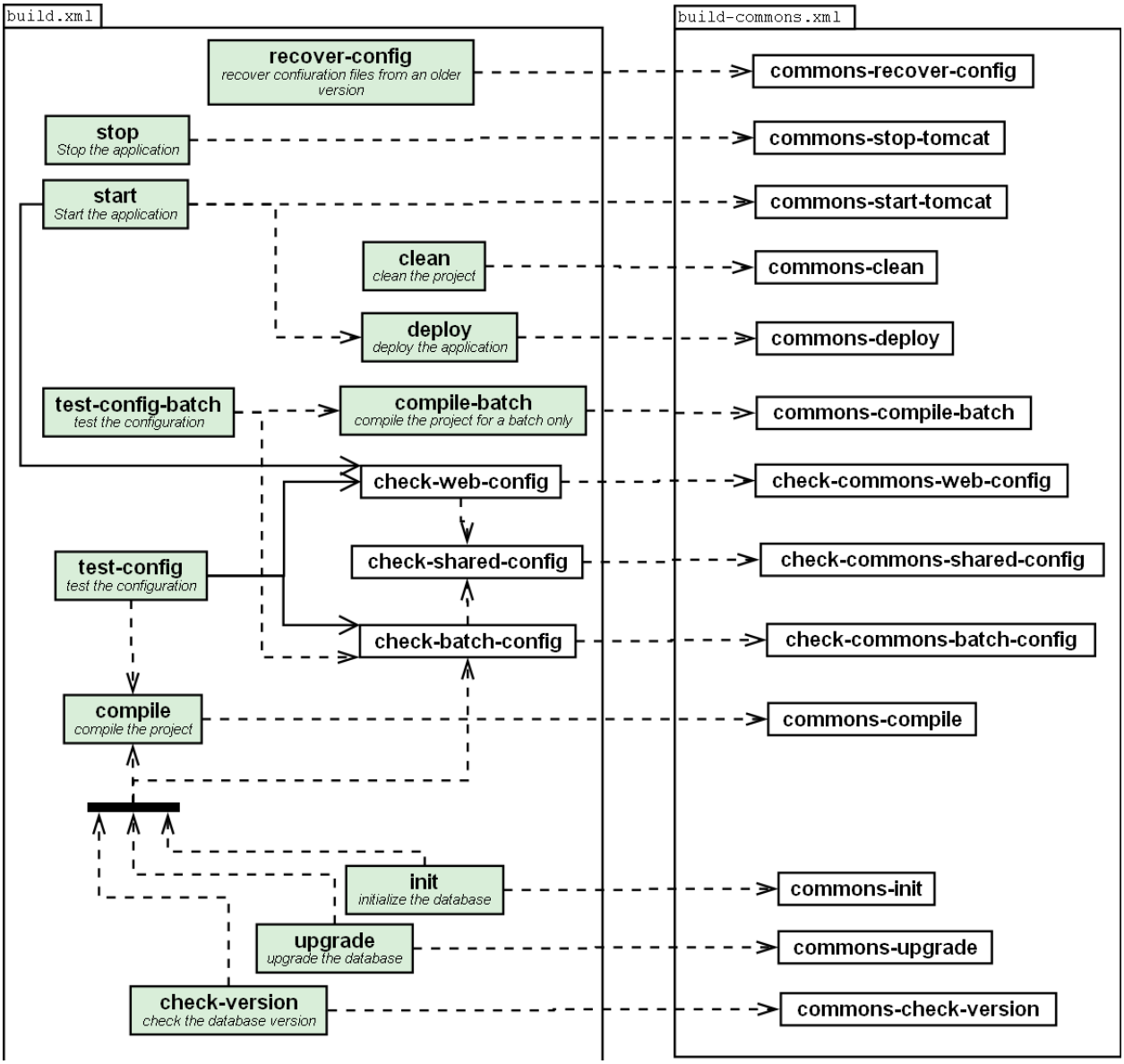
En mode développement (comme vu pendant le processus d'installation), le fichier de tâches *ant* à utiliser est le fichier **`build-devel.xml`**.

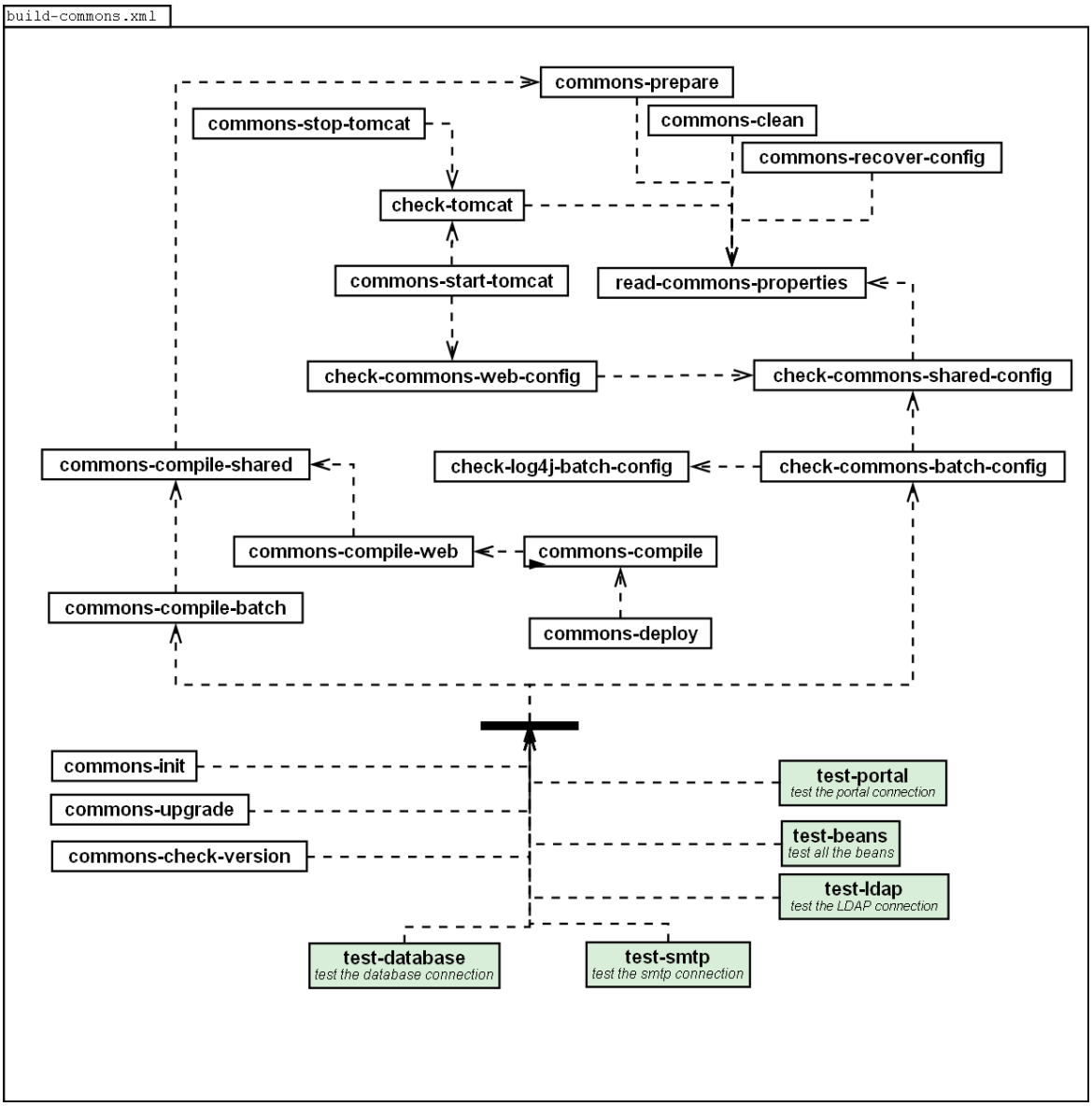
En mode exploitation, le fichier de tâches *ant* à utiliser est le fichier **`build.xml`**.

Les tâches ant en mode exploitation

En mode déploiement, le fichier **`build-devel.xml`** n'est pas disponible (il n'est pas inclus dans les fichiers distribués). Il faut donc dans ce cas utiliser les tâches du fichier **`build.xml`**.

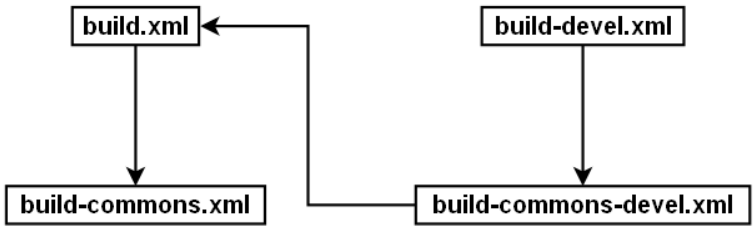
Les schémas ci-dessous donnent un aperçu des tâches *ant* disponibles :





Les tâches ant en mode développement

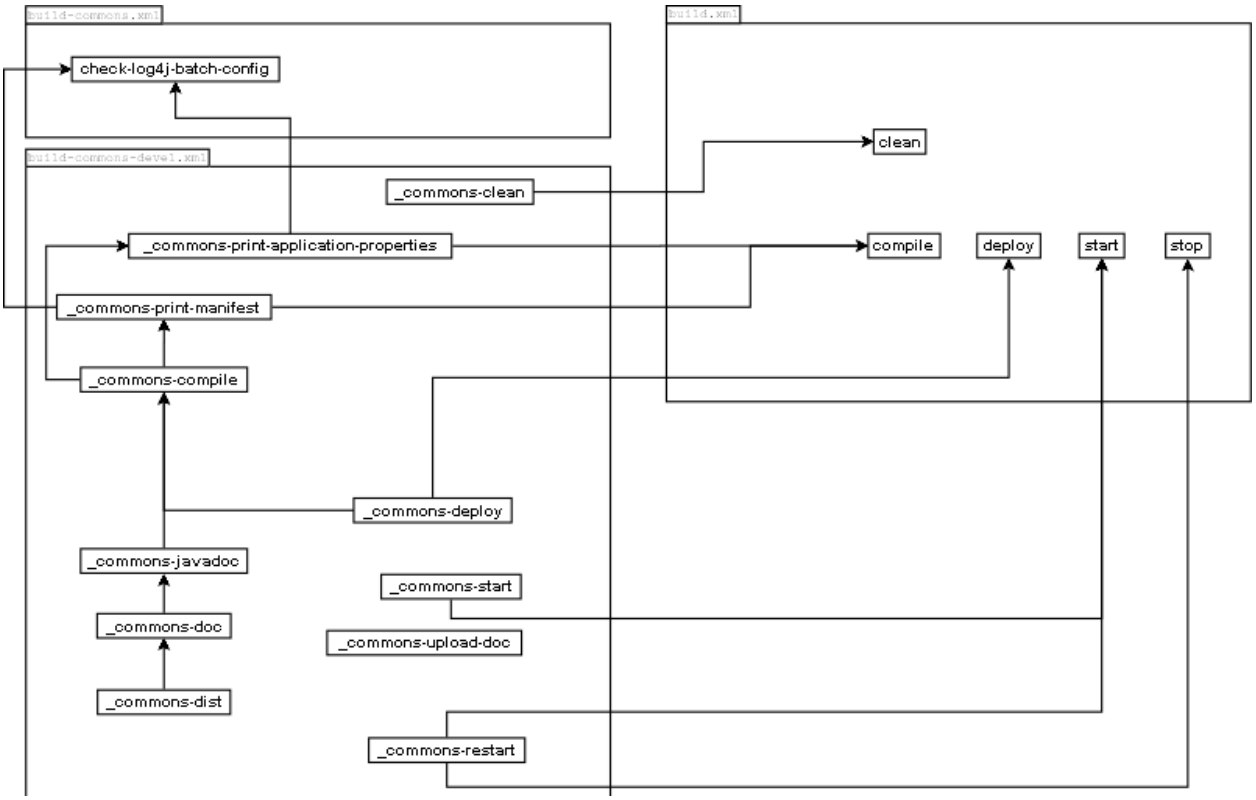
En mode développement, il faut utiliser les tâches fournies par le fichier **`build-devel.xml`**, qui inclut le fichier **`build.xml`** et donc donne accès à toutes ses tâches. Le schéma d'inclusion des fichiers de tâches ant est le suivant :



Ce schéma montre qu'une tâche mise à disposition d'un exploitant dans le fichier **`build.xml`** sera aussi accessible depuis le fichier **`build-devel.xml`** qui est le seul qui a besoin d'être manipulé par le développeur depuis l'environnement Eclipse.

Il est important de noter que les fichiers **`build-commons.xml`** et **`build-commons-devel.xml`**, bien que présents dans le répertoire du projet, ne doivent jamais être modifiés par les développeurs. En effet, ils sont recopiés à partir du projet *esup-commons* avant l'exécution de toute tâche, et toute modification locale est perdue.

Le diagramme des tâches de **`build-commons-devel.xml`** est le suivant :



4. Organisation des fichiers

Comme vu précédemment, du point de vue du développeur, une application est composée de deux projets *Eclipse* :

- Le projet *esup-commons*, qui rassemble les choses communes à toutes les applications basées sur *esup-commons*,
- Le projet de l'application.

Du point de vue de l'exploitant, une application est composée d'une hiérarchie unique de fichiers, issue de la décompression d'une archive.

L'organisation de fichiers décrite dans cette partie est celle de la vue administrateur ; le développeur (sous *Eclipse*) trouvera les fichiers dans un des deux projets *Eclipse*.

4.1. `/` : les fichiers de développement et de déploiement

Le fichier **`/build.properties`** définit des propriétés utilisées par le fichier *ant* **`build.xml`**. Il définit en particulier la manière dont est déployée l'application.

Le fichier **`/build.xml`** contient les tâches *ant* de l'application. Il inclut le fichier **`/build-commons.xml`**, qui ne doit pas être modifié même par les développeurs.

Le fichier **`/build-devel.xml`** contient les tâches *ant* de l'application réservées aux développeurs (ce fichier n'est pas distribué). Il inclut le fichier **`/build-commons-devel.xml`**, qui ne doit pas être modifié même par les développeurs.

Le fichier **`/version.properties`** est généré automatiquement par les tâches *ant*, il ne doit pas être modifié.

4.2. `/build` : la compilation

Le répertoire **`/build`** est utilisé pour la compilation de l'application, les fichiers s'y trouvant ne doivent pas être modifiés à la main.

4.3. `/deploy` : le déploiement

Le répertoire **`/deploy`** est utilisé pour le déploiement de l'application en mode *quick-start*, les fichiers s'y trouvant ne doivent pas être modifiés à la main.

4.4. `/docs` : la documentation

Toute la documentation du projet est située dans le répertoire **`docs/`**, le sous-répertoire **`docs/api`** contient le *Javadoc* généré.

4.5. `/properties` : les fichiers de configuration

Les fichiers de configuration sont en général fournis sous forme de fichiers d'exemple (**`xxx-example.xml`**). En mode développement, ces fichiers d'exemple se trouvent soit dans la hiérarchie du projet *esup-commons* (s'ils ne sont pas particulier à une application), soit dans le projet de votre application (si la configuration par défaut est particulière).

Important : la bonne gestion des fichiers de configuration. Les exploitants peuvent renommer ces fichiers (en **`xxx.xml`**), mais les développeurs doivent copier les fichiers d'exemple sous leur nom final (en laissant les fichiers d'exemple en place car ceux-ci sont liés au dépôt SVN).

Note : Il est possible, pour un développeur, de préciser quels fichiers de configuration seront livrés sous forme d'exemple ou directement utilisables sans à être renommés.

Le fichier **applicationContext.xml** importe tous les fichiers de configuration *Spring*. Il peut également être utilisé pour définir des *beans*.

/properties/auth : l'authentification

Le fichier de configuration *Spring* **auth-example.xml** définit le *bean* **authenticationService**, qui sert à l'application à récupérer l'identifiant de l'utilisateur connecté. Il doit être copié en **auth.xml**.

/properties/cache : le cache

Le fichier de configuration *Spring* **cache-example.xml** définit le *bean* **cacheManager**, qui sert à l'application à s'appuyer sur un gestionnaire de cache. Il doit être copié en **cache.xml**.

Le fichier de configuration *EhCache* **ehcache-example.xml** définit la configuration de la bibliothèque *EhCache*.

/properties/dao : l'accès aux données

Le fichier de configuration *Spring* **dao-example.xml** définit la manière dont l'application récupère les données de la base de données, par exemple avec *Hibernate*. Il doit être copié en **dao.xml**.

Le fichier de configuration *Hibernate* **hibernate/hibernate.cfg-example.xml** définit la manière dont *Hibernate* se connecte à la base de données. Il doit être copié en **hibernate.cfg.xml**. Il est référencé par le *bean* **abstractHibernateSessionFactory** de **dao.xml**.

Les fichiers de configuration *Hibernate* **hibernate/mapping/*.hbm.xml** décrivent le *mapping* entre les classes Java et les tables de la base de données. Ils sont également référencés par le *bean* **abstractHibernateSessionFactory** de **dao.xml**. Il n'est en général pas nécessaire pour les administrateurs de modifier ces fichiers, ils ne sont pas fournis sous forme d'exemples.

/properties/deepLinking : les liens directs

Le fichier de configuration *Spring* **deepLinking-example.xml** définit le *bean* **deepLinkingRedirector**, qui indique comment l'application gère les liens (hypertextes) directs. Il doit être copié en **deepLinking.xml**.

/properties/exceptionHandling : la gestion des exceptions

Le fichier de configuration *Spring* **exceptionHandling-example.xml** définit le *bean* **exceptionServiceFactory**, qui indique comment l'application gère les exceptions. Il doit être copié en **exceptionHandling.xml**.

/properties/i18n : la gestion de l'internationalisation

Le fichier de configuration *Spring* **i18n-example.xml** définit le *bean* **i18nService**, qui indique comment l'application récupère les chaînes de caractères utilisés dans l'application. Il doit être copié en **i18n.xml**.

Les fichiers **bundles/*_*.properties** contiennent les chaînes de caractères proprement dites.

/properties/init : l'initialisation et la mise à jour

Le fichier de configuration *Spring* **init-example.xml** définit le *bean* **versionningService**, qui indique la manière dont est initialisée la base de données. On y trouvera par exemple l'uid du premier administrateur de l'application qui sera créé automatiquement en même temps que la base de données. Il doit être copié en **init.xml**.

`/properties/jsf` : le MVC et la présentation

Le fichier **`commons-render-kit.xml`** définit les classes de rendu de la *taglib* de *esup-commons*.

Le fichier **`fck-faces-config.xml`** définit l'interface de *FckEditor* pour *JSF*.

Le fichier **`application.xml`** définit la résolution de variables sous *JSF* ainsi que les langues disponibles.

Le fichier **`navigation-rules.xml`** définit les règles de navigation entre les pages de l'application.

`/properties/ldap` : l'accès à l'annuaire LDAP

Le fichier de configuration *Spring* **`ldap-example.xml`** définit le *bean* **`ldapService`**, qui indique comment sont faits les accès à l'annuaire LDAP. Il doit être copié en **`ldap.xml`**.

`/properties/logging` : les traces de l'application

Le fichier **`log4j-example.properties`** configure *log4j* (la bibliothèque utilisée pour les traces) en mode web (lancement par la tâche *ant* **`start`**). Le fichier **`log4j-batch-example.properties`** est utilisé en mode *batch*, depuis une autre tâche *ant*. Ils doivent être copiés respectivement en **`lo4j.properties`** et **`log4j-batch.properties`**.

`/properties/misc` : des choses qu'on a pas su mettre autre part ;-)

Le fichier de configuration *Spring* **`application.xml`** définit le *bean* **`applicationService`**, qui indique à l'application son numéro de version, copyright, … Ce fichier n'est pas fourni sous forme d'exemple, il ne doit normalement pas être modifié par les exploitants.

Le fichier de configuration *Spring* **`abstractBeans.xml`** définit des *beans* abstraits utilisés par héritage (notamment dans **`/properties/web/controllers.xml`**). Ce fichier n'est pas fourni sous forme d'exemple, il ne doit normalement pas être modifié par les exploitants.

`/properties/portal` : l'accès aux informations du portail

Le fichier de configuration *Spring* **`portal-example.xml`** définit le *bean* **`portalService`**, qui indique à l'application comment récupérer les informations du portail (groupes, attributs utilisateur, …). Il doit être copié en **`portal.xml`**.

`/properties/smtp` : l'envoi de courriers électroniques

Le fichier de configuration *Spring* **`smtp-example.xml`** définit le *bean* **`smtpService`**, qui indique à l'application comment envoyer les courriers électroniques. Il doit être copié en **`smtp.xml`**.

`/properties/tags` : le rendu des tags esup-commons

Le fichier de configuration *Spring* **`tags-example.xml`** définit le *bean* **`tagsConfigurator`**, qui configure dynamiquement les balises de la *taglib esup-commons*. Il doit être copié en **`tags.xml`**.

`/properties/urlGeneration` : la génération des liens hypertextes

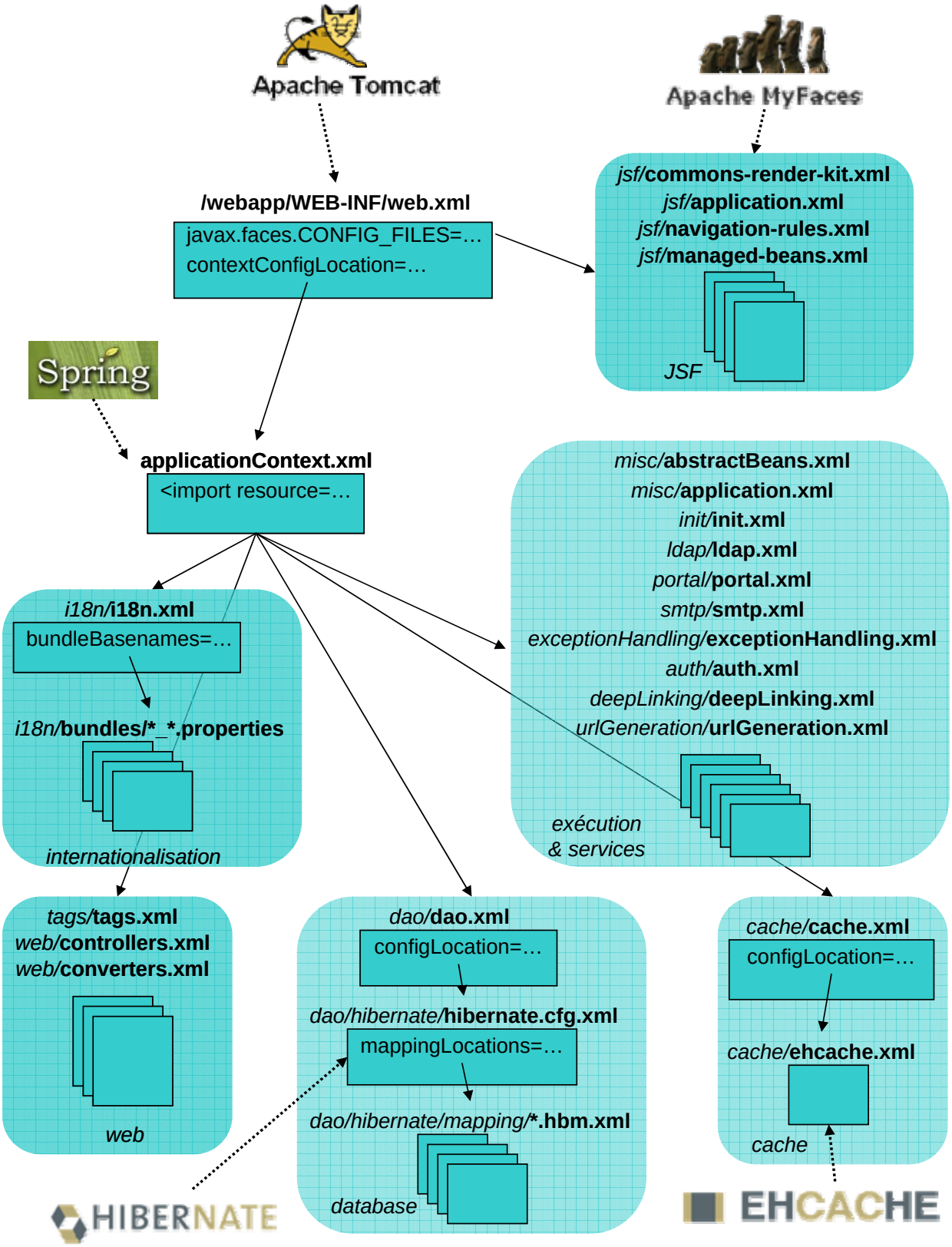
Le fichier de configuration *Spring* **`urlGeneration-example.xml`** définit le *bean* **`urlGenerator`**, qui génère les liens hypertextes vers l'application (avec prise en compte de l'authentification, des paramètres de liens directs, …). Il doit être copié en **`urlGeneration.xml`**.

`/properties/web` : l'interface utilisateur

Le fichier de configuration *Spring* **`controlers.xml`** définit les contrôleurs de l'application, qui réagissent aux actions de l'utilisateur.

Le fichier de configuration *Spring* **`converters.xml`** définit les convertisseurs de l'application, qui convertissent des objets en chaînes (vice-versa) lors des interactions utilisateur.

Articulation des fichiers de configuration

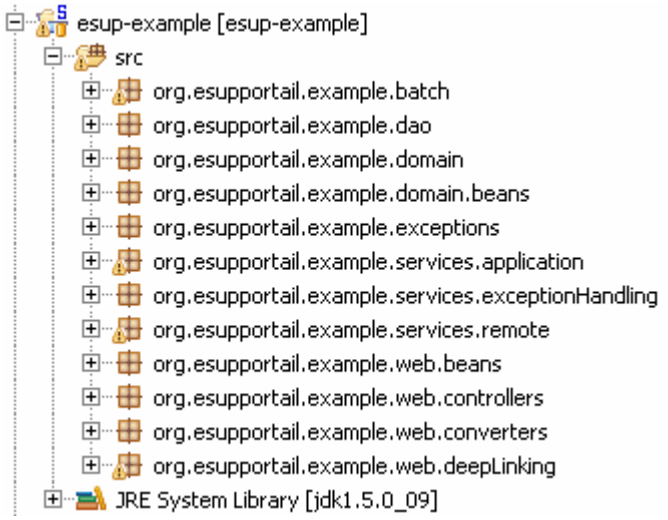


4.6. `/src` : les sources

Tous les sources de l'application doivent être situés dans le répertoire **`/src`**.

Pour faciliter la maintenance des applications basées sur esup-commons, il est d'usage d'adopter les mêmes conventions de nommage des *packages Java* :

Voici à titre d'exemple la hiérarchie des *packages* de l'application *esup-example* :



4.7. `/utils` : les utilitaires

Le répertoire **`/utils`** est utilisé pour stocker toutes sortes de fichiers, autres que des sources et des fichiers de configuration, et qui ne sont pas utiles ni au déploiement ni au fonctionnement de l'application.

On y trouvera, par exemple :

- Des librairies utiles à la compilation (**`/utils/ant/ant-contrib-*.jar`**),
- Le certificat de l'AC racine du CRU (**`/utils/cas/cru-root.keystore`**),
- Des fichiers d'aide au déploiement (**`/utils/uPortal/app-portlet-[chanpub|fragment].xml`**),
- La configuration de *checkstyle* (**`/utils/checkstyle/checkstyle.xml`**),
- Des librairies utiles à la compilation comme *Pluto*, *JSP*, *servlet*, … (**`/utils/lib/*.jar`**),
- Des fichiers utiles au déploiement en *quick-start* (**`/utils/tomcat/*`**).

4.8. `/webapp` : l'application web et les bibliothèques

`/webapp/media` : les fichiers statiques

On trouvera dans ce répertoire tous les fichiers délivrés de manière statique par l'application web aux clients :

- Les images,
- Les feuilles de style (**`*.css`**),
- …

L'intérêt de ce regroupement est de pouvoir *shunter Tomcat* par un frontal *Apache*, plus efficace pour le délivrement statique.

`/webapp/META-INF` : le manifest

Le fichier **`/webapp/META-INF/MANIFEST.MF`** est produit automatiquement par la tâche *ant* **`_dist`**.

`/webapp/stylesheets` : les pages JSF

Toutes les pages *JSF* doivent être situées à cet endroit pour pouvoir être protégées d'un accès direct de manière globale.

`/webapp/WEB-INF` : la configuration de l'application web

Le fichier **`portlet-example.xml`** indique à *Pluto* comment configurer la *portlet*, il n'est utilisé qu'en mode *portlet*. Il doit être copié en **`portlet.xml`**.

Les fichiers **`web-portlet-example.xml`** (resp. **`web-servlet-example.xml`**) est un exemple de configuration du contexte *Tomcat* associé à l'application. En mode *portlet* (resp. *servlet*), il doit être copié en **`web.xml`**.

Les fichiers **`esup-commons.tld`** et **`fck-faces.tld`** sont les fichiers de définition des *taglibs* de *esup-commons* et *FckEditor*.

`/webapp/WEB-INF/lib` : les bibliothèques de l'application

Même les applications en mode *batch* seulement doivent utiliser **`/webapp/WEB-INF/lib`** pour déposer leurs bibliothèques, même si dans ce cas le nommage n'est pas très approprié.

4.9. `/website` : la construction du site web

Le répertoire **`/website`** est utilisé de manière temporaire par la tâche *ant* **`doc`** pour générer la documentation du projet avant de la transférer sur le site web du projet, les fichiers s'y trouvant ne doivent pas être modifiés même par les développeurs.

5. Les *beans* Spring

Ce chapitre n'a pas la prétention d'être une formation à *Spring* (Pour plus d'informations vous pouvez vous reporter à une présentation de *Spring* en Français comme celle de Thierry Templier : http://www.springframework.org/node/155). Ne sont abordés ici que quelques éléments qui permettent de mieux comprendre certains éléments des fichiers de configuration de *esup-commons*.

Tout au long de ce chapitre nous allons nous appuyer sur un exemple (configuration du gestionnaire d'exceptions) :

```
<bean
  id="exceptionServiceFactory"
  class="org.esupportail.formation.services.exceptionHandling.CachingE
mailExceptionServiceFactoryImpl"
  parent="abstractApplicationAwareBean">
  <property
      name="doNotSendExceptionReportsToDevelopers"
      value="false"/>
  <property name="smtpService" ref="smtpService"/>
  <property name="authenticationService" ref="authenticationService"/>
  <property name="recipientEmail" value="webmaster@domain.edu"/>
  <property name="cacheManager" ref="cacheManager"/>
  <property name="cacheName" value=""/>
</bean>
```

5.1. Les fichiers de configuration

Spring permet de créer des objets (appelés alors *beans*) en les déclarant dans un fichier de configuration *XML*.

Le fichier de configuration principal (**`/properties/applicationContext.xml`**) est déclaré dans le **`web.xml`** sous forme d'un paramètre de l'application :

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:/properties/applicationContext.xml
  </param-value>
</context-param>
```

Dans *esup-commons* ce fichier de configuration principal contient seulement des inclusions de fichiers de configurations spécialisés par domaine, par exemple :

```
<import resource="exceptionHandling/exceptionHandling.xml" />
```

Il est possible, suivant les besoins de votre application, de supprimer ou d'ajouter des fichiers de configuration spécialisés.

5.2. L'injection de données

Une des caractéristiques de base de *Spring* est de permettre l'injection de données.

Note : Cette fonctionnalité est aussi possible avec *JSF*, qui est également utilisé dans *esup-commons*, mais *JSF* est moins puissant que *Spring* dans ce domaine. *esup-commons* n'utilise donc pas l'injection de données de *JSF*.

L'injection de données permet de renseigner des attributs d'un *bean* via un fichier de configuration. Le *bean* doit disposer d'un setter pour l'accès à ces attributs. Voyons quelques exemples.

Injection d'une chaîne de caractères

```
<property name="recipientEmail" value="webmaster@domain.edu"/>
```

Dans ce cas, la méthode **setRecipentEmail()** sera appelée avec la valeur **webmaster@domain.edu**.

Injection d'un autre bean

```
<property name="authenticationService" ref="authenticationService"/>
```

Note : On voit ici un autre aspect important de *Spring* qui est l'utilisation quasi systématique des interfaces. La classe **CachingEmailExceptionServiceFactoryImpl** (qui correspond au *bean* **exceptionServiceFactory** et contient la définition de cette propriété **authenticationService**) a un attribut **authenticationService** de type **AuthenticationService**. **AuthenticationService** est une interface. Le *bean* **authenticationService** doit être d'une classe qui implémente cette interface. Ceci permet d'avoir plusieurs implémentations possibles de cette interface et de choisir, simplement en modifiant un fichier de configuration, laquelle on utilise. Cette approche est particulièrement intéressante. Elle permet, par exemple, de très facilement tester une couche de l'application en branchant des implémentations de tests des autres couches avec lesquelles le *bean* doit interagir.

Injection d'une liste

```
<property name="servers">
  <list>
    <ref bean="smtpServer1" />
    <ref bean="smtpServer2" />
  </list>
</property>
```

5.3. L'héritage de configuration

Spring n'oblige pas à saisir, dans toutes les définitions de *beans*, les mêmes propriétés. Pour cela, il est possible d'utiliser le mot-clé **parent**.

Le « parent » à un attribut **abstract="true"** car il ne doit pas être créé en mémoire par *Spring*. Cette notation permet de se rapprocher de l'héritage *Java* qui est beaucoup utilisé dans *esup-commons*.

Exemple d'un *bean* « parent » ayant aussi lui-même un « parent » :

```
<bean
  id="abstractApplicationAwareBean"
  parent="abstractI18nAwareBean"
  abstract="true">
  <property name="applicationService" ref="applicationService" />
</bean>
```

Note : L'attribut **scope** (voir ci-après) n'est pas héritable, il est donc inutile de le préciser pour un *bean* abstrait.

5.4. Vérification des beans

Les *beans* manipulés par *Spring* n'ont pas, par défaut, de dépendances particulières avec *Spring*.

Il est, par contre, possible de sciemment introduire une dépendance avec *Spring* pour obtenir des services supplémentaires.

Faire que votre *bean* implémente l'interface **InitializingBean** de *Spring* en fait partie. Cette interface vous oblige à implémenter une méthode **afterPropertiesSet** qui sera appelée par

Spring juste après l'initialisation du *bean*. Cette méthode vous permet de vérifier que toutes les propriétés sont biens initialisées. Si ce n'est pas le cas, vous pouvez, par exemple, lever une exception ou affecter une valeur par défaut.

On trouvera par exemple :

```
public void afterPropertiesSet() {
  super.afterPropertiesSet();
  if (!StringUtils.hasText(exceptionView)) {
    exceptionView = DEFAULT_SERVLET_EXCEPTION_VIEW;
    logger.info(getClass() + ": no exceptionView set, using default ["
      + exceptionView + "]");
  }
}
```

5.5. Portée des beans

Spring (à partir de 2.0) offre une notion de portée (**scope**).

Par défaut un *bean* est de portée **singleton**. *Spring* crée une seule instance de ce *bean* pour toute la durée d'exécution de l'application.

Il existe aussi des portées **session** et **request** qui, respectivement, permettent d'avoir une instance du *bean* par session utilisateur (au sens d'une application web) ou par requête (au sens HTTP). Cette notion est particulièrement intéressante pour les contrôleurs web d'une application.

Les contrôleurs des applications web sont en général des *beans* de portée **session**, comme par exemple :

```
<bean id="administratorsController"
  class="[...].formation.web.controllers.AdministratorsController"
  parent="abstractContextAwareController"
  scope="session" />
```

Un *bean* de scope **request** peut faire référence, via ses propriétés, à un *bean* de scope **session** ou **singleton**. De même, un *bean* de scope **session** peut faire référence à un *bean* de scope **singleton**. La réciproque n'a pas de sens (et provoque une exception).

5.6. Récupération des beans

Lorsque l'on veut récupérer un *bean* à partir de son nom, il faut obligatoirement utiliser la classe **BeanUtils** fournie par *esup-commons*.

En mode batch par exemple, on utilisera :

```
DomainService domainService =
       (DomainService) BeanUtils.getBean("domainService") ;
```

La classe **BeanUtils** de *esup-commons* utilise toujours la même *beanFactory* (statique), et il ne faut pas créer une nouvelle *beanFactory* « à la main » (par exemple à partir du fichier de configuration **/properties/applicationContext.xml**.

6. Déploiement en *quick-start*

Un point fort d'*esup-commons* est de pouvoir déployer les applications en *portlet* ou en *servlet*, tout en utilisant strictement le même code, et c'est dans cet objectif que *JSF* a été choisi par rapport à *Spring* car son *MVC* est indépendant du mode de déploiement.

Le déploiement en *quick-start* a été mis au point pour faciliter le déploiement des applications, en particulier pour les administrateurs n'ayant pas de connaissance *uPortal* ou même *Tomcat*, **c'est la manière la plus simple d'installer une application bâtie sur esup-commons**.

Techniquement, il s'agit d'un déploiement *servlet* simplifié.

De manière générale, puisqu'il est possible avec *esup-commons* de distribuer très simplement un *quick-start*, il est conseillé :

- Pour les développeurs, de distribuer leur application sous forme d'un *quick-start* (en plus de la version classique *servlet*/*portlet*),
- Pour les exploitants, de commencer par déployer l'application sous forme d'un *quick-start*, ou au moins d'une *servlet*, pour bien séparer les problèmes de l'application de ceux liés au déploiement des *portlets*.

La technique de fabrication d'un *quick-start* est discutée dans un chapitre dédié.

Note : il a été volontairement choisi par les formateurs de repousser les déploiements en *servlet* et *portlet* ultérieurement, et de se baser sur un déploiement en *quick-start* pour le test de la plupart des fonctionnalités de *esup-commons*.

6.1. Le fichier `build.properties`

Pour déployer une application sous forme de *quick-start*, il suffit d'indiquer que le déploiement se fait sous forme de *quick-start*, en positionnant la propriété **`${quick-start}`** à **`true`** (**`false`** par défaut).

On peut ensuite, de manière optionnelle, indiquer :

- Le port sur lequel tournera l'application avec la propriété **`${tomcat.port}`**, par défaut **`8080`**,
- Le port sur lequel on arrête l'application avec la propriété **`${tomcat.shutdown-port}`**, par défaut **`8009`** (cela permet de faire tourner plusieurs instances de *Tomcat* sur la même machine),
- Le nom du serveur avec la propriété **`${tomcat.host}`**, par défaut **`localhost`**,
- L'emplacement du *keystore* contenant l'AC racine du serveur *CAS* (en cas d'authentification *CAS*) avec la propriété **`${tomcat.keystore}`**.

Un **`build.properties`** minimal sera de la forme suivante :

```
quick-start=true
```

Il permettra de faire tourner l'application, qui sera accessible sur **`http://localhost:8080`**.

Dans le cas d'un déploiement en *quick-start*, le répertoire de déploiement est **`/deploy`**, et *Tomcat* est installé dans le répertoire **`/apache-tomcat-x.y.z`**.

Exercice 14 : Démarrer et tester l'application

Démarrer l'application à l'aide de la tâche *ant* **`_start`** et tester en accédant à **`http://localhost:8080`**.

Note : la tâche *ant* **`_restart`** peut ensuite être utilisée pour redémarrer l'application, en appelant successivement les tâches **`_stop`**, **`_deploy`** puis **`_start`**.

Il arrive quelque fois que la tâche **`_stop`** (et donc la tâche **`_restart`**) échoue en ne réussissant pas à tuer le processus *Java* de l'application, ce qui résulte en une erreur du type :

```
java.net.BindException: Address already in use: JVM_Bind:8080
```

La seule solution est alors de tuer « à la main » les processus *Java* (de *Tomcat*) à l'aide du gestionnaire de tâches.

6.2. La gestion des logs

Cf 28.

6.3. Les feuilles de style (CSS)

Cf Annexe B.

7. JSF : Java Server Faces

7.1. Généralités

Exemple de page

```
<%@include file="_include.jsp"%>
<e:page
    stringsVar="msgs"
    menuItem="preferences"
    locale="#{sessionController.locale}"
    authorized="#{preferencesController.pageAuthorized}">
  <%@include file="_navigation.jsp"%>
  <h:form id="preferencesForm">
    <e:section value="#{msgs['PREFERENCES.TITLE']}" />
    <e:messages />
    <e:panelGrid columns="2">
      <e:outputLabel
          for="locale"
          value="#{msgs['PREFERENCES.TEXT.LANGUAGE']}" />
      <h:panelGroup>
        <e:selectOneMenu
            id="locale"
            onchange="submit();"
            value="#{preferencesController.locale}"
            converter="#{localeConverter}" >
          <f:selectItems
              value="#{preferencesController.localeItems}" />
        </e:selectOneMenu>
        <e:commandButton
            value="#{msgs['_.BUTTON.CHANGE']}"
            id="localeChangeButton" />
      </h:panelGroup>
    </e:panelGrid>
  </h:form>
  <script type="text/javascript">
    hideButton("preferencesForm:localeChangeButton");
  </script>
</e:page>
```

Dans la page ci-dessus, la balise *JSP* **<%@include file="_include.jsp"%>** permet d'inclure toutes les librairies nécessaires (voir plus loin).

La balise *JSF* **<e:page>** se charge de la mise en forme globale, que l'on soit en *servlet* ou en *portlet*.

La balise *JSP* **<%@include file="_navigation.jsp"%>** permet d'inclure la barre de navigation, commune à toutes les pages.

La balise *JSF* **<e:section>** affiche le titre de la page.

La balise *JSF* **<h:form>** encadre un formulaire.

La balise *JSF* **<e:outputLabel>** affiche le label d'un formulaire, celui qui possède l'identifiant **locale**.

La balise *JSF* **<e:selectOneMenu>** propose un choix simple de valeurs dans une boite déroulante.

La balise *JSF* **<f:selectItems>** renseigne la balise précédente sur les choix possibles, fournis par la méthode **getLocaleItems()** du *bean* **preferencesController**.

La balise *JSF* **<e:commandButton>** est un bouton de soumission du formulaire. Lors de la soumission du formulaire, la méthode **setLocale()** du *bean* **preferencesController** est appelée avec la valeur sélectionnée dans la boite déroulante.

Syntaxe EL

JSF dispose d'un *Expressions Language* (*EL*) qui lui est propre. Syntaxiquement différent de l'*EL* de *JSP* 2.0. Il est utilisable dans les attributs des *taglibs JSF*. Il permet d'accéder en lecture ou en écriture à des propriétés du contrôleur (dans la mesure où celui-ci implémente des getters/setters sur ses propriétés). Il permet aussi d'invoquer une action du contrôleur (méthode sans paramètres qui renvoie une chaîne).

La syntaxe de base est **#{beanName.propertyName}** ou **#{beanName.methodName}**.

Il est possible d'accéder à des objets récursivement, par exemple :

- **#{homeController.context.name}**.

Il est possible d'accéder à des éléments de tableaux ou de **Map**, par exemple :

- **#{tableau[1]}**,
- **#{hash.key}**,
- **#{hash["key"]}**,
- **#{hash[keyvar]}** (dans ce dernier exemple **keyvar** est évalué avant de retrouver l'entrée dans la table **hash**)

Certains tableaux associatifs sont définis par défaut : **param**, **header**, **cookie**, etc.

On peut également utiliser quelques opérateurs logiques (**and**, **or**, **not**, **empty**, …) ainsi que la plupart des opérateurs mathématiques.

Navigation entre les pages

La navigation entre les pages de l'application est faite en *JSF* à l'aide de règles de navigation écrites en *XML*, définies dans *esup-commons* par convention dans le fichier **/properties/jsf/navigation-rules.xml**.

Cette approche permet de bien dissocier les contrôleurs de la navigation en elle-même. Quand on appelle une méthode d'un contrôleur, par exemple au moment de la validation d'un formulaire, cette méthode (*callback*) renvoie une simple chaîne de caractères qui sera utilisée par *JSF* pour trouver la règle de navigation correspondante dans le fichier **/properties/jsf/navigation-rules.xml**

Voici ci-dessous un exemple de règle de navigation :

```
<navigation-rule>
  <display-name>administrators</display-name>
  <from-view-id>/stylesheets/administrators.jsp</from-view-id>
  <navigation-case>
    <from-outcome>addAdmin</from-outcome>
    <to-view-id>/stylesheets/administratorAdd.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>deleteAdmin</from-outcome>
    <to-view-id>
      /stylesheets/administratorDelete.jsp
    </to-view-id>
  </navigation-case>
</navigation-rule>
```

La balise **from-view-id** est facultative. Elle donne la vue (page) à partir de laquelle les règles de navigation vont s'appliquer. Si elle n'est pas présente, la règle est potentiellement applicable depuis toutes les pages de l'application.

Note : La vue est techniquement une page *JSP* mais l'utilisateur accède à des vues dont l'extension est **.faces** (paramétrage du fichier **web.xml**).

Le noeud **navigation-case** permet de définir une règle de navigation :

- **from-outcome** est l'action de l'utilisateur. Cette action peut être définie « en dur » dans la page *JSP* (par ex. **<h:commandButton value="suivant" action="next"/>**) ou bien être le résultat de l'exécution d'une méthode d'un bean contrôleur de l'application (par ex. **<h:commandButton value="suivant" action="#{controller.value}"/>**).
- **to-view-id** donne la page de destination.
- **<redirect/>** peut être utilisé pour indiquer que l'affichage de la page de destination doit être fait sous forme d'une redirection *HTTP* (code *HTTP* 302). Cette balise doit être utilisée pour protéger l'utilisateur des effets de soumission multiple des formulaires à l'aide des boutons « Page précédente » et « Page suivante » des navigateurs.

Il est parfois complexe (et peu confortable) d'éditer les règles de navigation « à la main ». On peut alors utiliser des outils graphiques, tel *Exadel*.

Protection des pages JSP

Les pages *JSP* ne devraient pas être accessibles directement. En effet, pour accéder aux vues on utilise l'extension **.faces** pour activer le filtre *MyFaces*. Une solution pour interdire l'accès aux *JSP* consiste à définir une contrainte de sécurité dans le **/webapps/WEB-INF/web.xml**

On utilisera par exemple :

```
<security-constraint>
  <display-name>Protect the raw JSP pages</display-name>
  <web-resource-collection>
    <web-resource-name>Raw JSF JSP Pages</web-resource-name>
    <url-pattern>/stylesheets/administrators.jsp</url-pattern>
    <url-pattern>/stylesheets/administratorsAdd.jsp</url-pattern>
    <url-pattern>/stylesheets/administratorsDelete.jsp</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description>No roles, so no direct access</description>
  </auth-constraint>
</security-constraint>
```

Bibliothèques utilisées

esup-commons utilise très communément les deux *tablig*s prévues par la spécification *JSF* (**jsf/core** et **jsf/html**) ainsi que les bibliothèques *JSF esup-commons* (cf. ci-dessous) et *Tomahawk* (cf. http://myfaces.apache.org/tomahawk/).

Afin de ne pas avoir à redéfinir ces bibliothèques dans toutes les pages de votre application, *esup-commons* est livré avec le fichier **_include.jsp** à inclure dans l'entête de vos pages JSP, de cette manière :

```
<%@include file="_include.jsp"%>
```

Le contenu de ce fichier est le suivant :

```
<%@ page
    language="java"
    contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>
<%@ taglib uri="http://commons.esup-portail.org" prefix="e"%>
```

Les tags commençant par **<f:** et **<h:** sont donc ceux de *MyFaces* (l'implémentation *JSF* de *Apache*), ceux commençant par **<t:** sont ceux de *Tomahawk* (les extensions de *MyFaces*) et ceux commençant par **<e:** ceux de *esup-commons*.

Page d'accueil de l'application

La page d'accueil de votre application se configure différemment suivant que votre application sera utilisée en mode *servlet* ou en mode *portlet*.

En mode servlet

En mode *servlet*, *esup-commons* propose par défaut une page **`index.jsp`** qui fait une redirection vers **`stylesheets/welcome.faces`**

Il est possible de surcharger cette page dans votre application.

En mode portlet

En mode *portlet* on utilise le paramètre **`default-view`** de la *portlet* **`org.esupportail.commons.web.portlet.FacesPortlet`** dont la définition est présente dans le fichier **`/webapps/WEB-INF/portlet.xml`**

Exemple :

```
<portlet>
  <portlet-name>esup-formation</portlet-name>
  <portlet-class>
    org.esupportail.commons.web.portlet.FacesPortlet
  </portlet-class>
  <init-param>
    <name>default-view</name>
    <value>/stylesheets/welcome.jsp</value>
  </init-param>
  ...
</portlet>
```

7.2. Le taglib esup-commons

esup-commons offre un *taglib*, une librairie de balises *JSF*. La plupart des balises de cette librairie sont configurées dynamiquement par le *bean* **`tagsConfigurator`**.

Le plus souvent, elles héritent de balises existantes pour les améliorer. Lorsqu'une balise est présente dans le *taglib esup-commons* et dans une autre librairie (*MyFaces*, *Tomahawk*, …), il faut alors utiliser la balise de *esup-commons*.

Les balises

Présentation des pages (e:page)

La balise **`e:page`** doit encapsuler toutes les autres balises de la page.

```
<%@ page language="java" %>
< %@ taglib uri="http://commons.esup-portail.org" prefix="e" %>
<e:page
    locale="#{sessionController.locale}"
    stringVars="msgs"
    menuItem="welcome" >
    ...
</e:page>
```

Le titre, les fichiers de script et les feuilles de style (en mode *servlet* uniquement) sont définies par le *bean* **`tagsConfigurator`**.

La balise charge automatiquement les chaînes spécifiées par le *bean* **`i18nService`** dans une variable de requête, (**`msgs`** dans notre exemple).

Note : l'internationalisation des applications sera détaillée ultérieurement.

Navigation (e:menu, e:menuItem, e:emptyMenu)

La navigation peut être spécifiée de la manière suivante avec *esup-commons* :

```
<e:menu>
  <e:menuItem
    id="welcome"
    value="#{msgs['NAVIGATION.TEXT.WELCOME']}"
    action="navigationWelcome"
    accesskey="#{msgs['NAVIGATION.ACCESSKEY.WELCOME']}"
    rendered="#{welcomeController.pageAuthorized}"
    />
  <e:menuItem
    id="filterRules"
    action="navigationAbout"
    value="#{msgs['NAVIGATION.TEXT.ABOUT']}"
    accesskey="#{msgs['NAVIGATION.ACCESSKEY.ABOUT']}"
    rendered="#{aboutBean.pageAuthorized}"
    />
</e:menu>
```

Les pages qui n'utilisent pas de barre de navigation devraient utiliser la balise **`<e:emptyMenu/>`** pour un rendu homogène.

Messages (e:message, e:messages)

Les balises **`e:message`** et **`e:messages`** doivent être utilisées pour bénéficier des classes *CSS* (*Cascading Style Sheets*) positionnées par le *bean* **`tagsConfigurator`**.

Formatage de caractères (e:text, e:bold, e:italic)

Ces balises peuvent être utilisées avec des paramètres, par exemple :

```
<e:text value="L''utilisateur est : ''{0}''" >
  <f:param value="#{controller.user.id }" />
</e:text>
<e:text value="L'utilisateur est beau." />
```


Comme on le voit ci-dessus, les apostrophes doivent être échappées quand il y a des paramètres, et seulement dans ce cas.

Formatage de pararaphes (e:paragraph, e:section, e:subsection)

Ces balises acceptent également des paramètres et les apostrophes doivent être échappées quand et seulement quand il y a des paramètres.

```
<e:section value="Mon titre " />
<e:subSection value="Mon sous-titre " />
<e:paragraph value="Mon paragraphe" />
```

Les balises *HTML* et les styles *CSS* utilisés pour rendre ces balises *JSF* sont configurables par le *bean* **`tagsConfigurator`**.

Entrées de formulaire (e:commandButton, e:inputText, e:inputTextArea, e:selectBooleanCheckbox, e:selectManyCheckbox, e:selectOneMenu)

Ces balises doivent être utilisées pour bénéficier des classes *CSS* positionnées par le *bean* **`tagsConfigurator`**.


Aucune balise **`e:commandLink`** n'est fournie car ces balises ne sont pas accessibles (*WAI*).

Tables (`e:dataTable`, `e:panelgrid`)

Ces balises doivent être utilisées pour bénéficier des classes *CSS* positionnées par le *bean* **`tagsConfigurator`**. Elle ont toutes deux un attribut supplémentaire **`alternateColors`** qui alterne les couleurs des lignes des tables (quand il est positionné à **`true`**).

Autres (`e:li`, `e:ul`, `e:outputLabel`)

Ces balises doivent être utilisées pour bénéficier des classes *CSS* positionnées par le *bean* **`tagsConfigurator`**.

Configuration dynamique des balises

Le fichier **`/properties/tags/tags.xml`** doit déclarer un *bean* nommé **`tagsConfigurator`**, qui doit implémenter l'interface **`TagsConfigurator`**.

Les valeurs par défaut de ce *bean* suivent les recommandations de http://www.ja-sig.org/wiki/display/UPC/JSR-168+PLT.C+CSS+Style+Definitions+section.

On se réfèrera au fichier d'exemple **`/properties/tags/tags-example.xml`** pour plus de détails.

7.3. Écriture des formulaires

L'écriture des formulaires *JSF* ne déroutera pas l'habitué des formulaires *JSP*. On utilisera par exemple :

```
<h:form id="administratorAddForm">
  <e:messages />
  <e:outputLabel
      for="ldapUid"
      value="#{msgs['ADMINISTRATOR_ADD.TEXT.PROMPT']}" />
  <e:inputText
      id="ldapUid"
      value="#{administratorsController.ldapUid}"
      required="true" />
  <e:message for="ldapUid" />
  <e:commandButton
      value="#{msgs['ADMINISTRATOR_ADD.BUTTON.ADD_ADMIN']}"
      action="#{administratorsController.addAdmin}" />
  <e:commandButton
      value="#{msgs['_.BUTTON.CANCEL']}"
      action="cancel"
      immediate="true" />
</h:form>
```

L'attribut **`value`** de la balise **`<e:inputText>`** contient une référence vers un attribut du contrôleur. Cet attribut sera mis à jour lors de la validation du formulaire.

L'attribut **`action`** du bouton **`<e:commandButton>`** contient une référence vers la *callback* (méthode) du contrôleur. C'est elle qui sera appelée lors de l'appui sur le bouton et dont le résultat sera utilisé par les règles de navigation pour connaître la page que l'application doit afficher en retour.

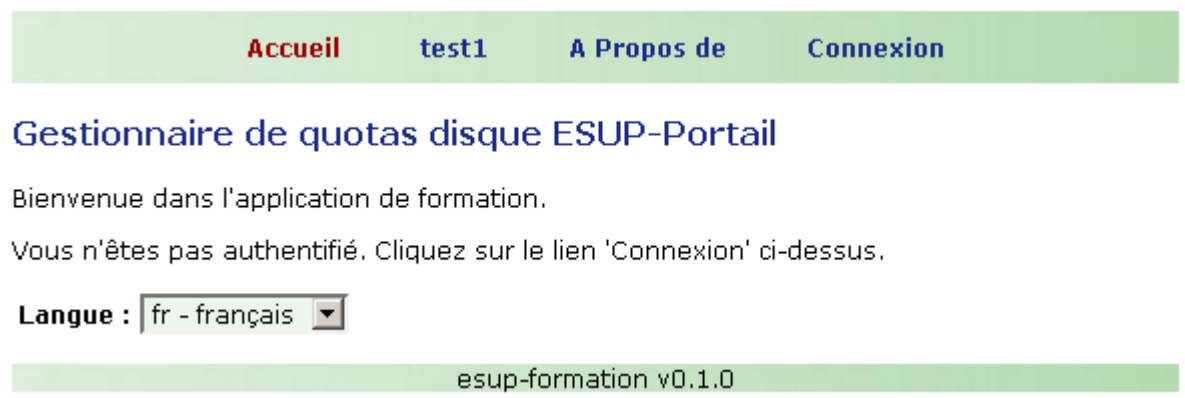
Le deuxième bouton a un attribut **`immediate="true"`**. Dans ce cas, les attributs du contrôleur relatifs aux balises **`<e:inputText>`** ne seront pas mis à jour et les éventuelles vérifications de forme ou de contenu ne seront pas exécutées. Ceci est particulièrement utile sur un bouton d'annulation comme c'est le cas ici.

Les balises **`<e:messages>`** et **`<e:message>`** sont traitées dans un paragraphe à suivre.

Exercice 15 : Ajouter une entrée dans la barre de navigation

Ajouter une entrée **`Test1`** dans la barre de navigation (**`_navigationItems.jsp`**) qui sera toujours affichée (pas d'attribut **`rendered`**) et dont l'action sera **`navigationTest1`** (en dur). Ajouter une règle de navigation pour que la sélection de cet item envoie sur la vue **`/stylesheets/test1.jsp`**. Tester.

Note : l'envoi vers **`test1.jsp`** ne doit pas marcher tant que la vue n'a pas été créée.



Exercice 16 : Ajouter une page *JSF*

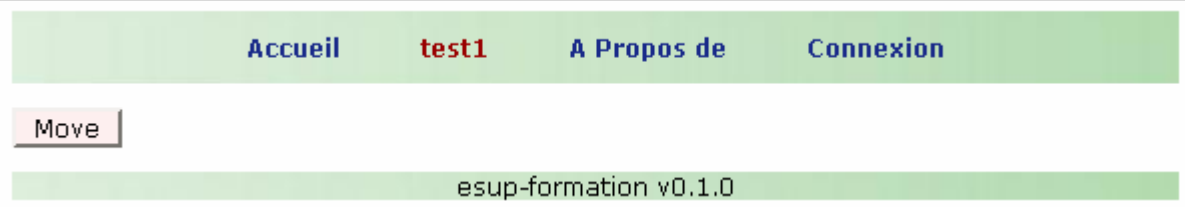
Créer la page **`test1.jsp`**, (la créer à partir de **`about.jsp`**, en ne gardant que la barre de navigation) et tester (elle doit s'afficher).

Note : la page doit s'afficher maintenant qu'elle existe.

Exercice 17 : Créer une règle de navigation

Ajouter un formulaire avec un bouton **`Move`** sur la page, dont l'action **`gotoWelcome`** envoie sur la page **`welcome.jsp`**.

Note : il faut ajouter une règle de navigation.



Exercice 18 : Créer un contrôleur

Ajouter une classe contrôleur **`Test1Controller`** (la créer à partir de **`AboutController`**), ajouter à cette classe une méthode **`callback()`** qui renvoie la chaîne **`gotoWelcome`**, et appeler cette méthode en réaction au bouton de la page **`test1.jsp`**. Tester.

Note : ne pas oublier de déclarer le bean **`test1Controller`** dans **`/properties/web/controllers.xml`**.

Messages d'erreur

Dans une page *JSP* les balises **`<e:messages>`** et **`<e:message>`** permettent d'afficher des messages d'erreur à l'utilisateur.

La balise **`<e:messages>`** permet d'afficher l'ensembles des messages d'erreurs.

La balise **<e:message>** permet d'afficher les messages d'erreurs relatifs à une balise **<e:inputText>** particulière. Elle dispose, à cet effet, d'un attribut **for** qui lui permet de faire le lien avec un attribut **id** d'une balise **<e:inputText>** donnée. Ce mécanisme est notamment utilisé quand l'attribut **required** de la balise **<e:inputText>** est positionné à **true** ou bien quand un validateur (cf. ci-dessous) lui est associé.

Il est aussi possible, dans une *callback* d'un contrôleur, de remonter des messages d'erreurs vers la vue. Ces messages seront généralement rendus grâce à la balise **<e:messages>**. Typiquement, la *callback* renverra **null**, c'est-à-dire que la page affichée à l'utilisateur restera la même (celle où s'est produite l'erreur de saisie). Dans l'exemple suivant, on affiche un message si l'utilisateur correspondant à l'identifiant donné est déjà administrateur :

```
User user = getDomainService().getUser(ldapUid);
if (user.getAdmin()) {
  addErrorMessage(
      "form:uid",
      "ADMINISTRATORS.MESSAGE.USER_ALREADY_ADMINISTRATOR",
      ldapUid);
  return null;
}
```

Le premier paramètre de la méthode **addErrorMessage** vaut **"uid"**. Le message ajouté sera alors affiché par la balise **<e:message for="uid">**. Il est également possible de spécifier un premier paramètre **null** ; le message d'erreur sera global, c'est-à-dire affiché par la balise **<e:messages>**.

Exercice 19 : Afficher un message sur une page *JSF*

Ajouter une boite de saisie (**id="myInput"**) au dessus du bouton dans le formulaire, lier le contenu de la boite à une propriété **myInput** de **test1Controller**, et modifier la méthode **callback()** pour qu'elle ne renvoie vers la page d'accueil que si **myInput** a au moins deux caractères, sinon reste sur la vue **test1.jsp** en affichant le message d'erreur **TEST1.MESSAGE.SHORT** (on pourra utiliser le code **addErrorMessage("form:myInput", "TEST1.MESSAGE.SHORT")**).

Note : il faudra aller déclarer le message d'erreur dans les fichiers **/properties/i18n/bundles/Messages_*.properties**).

Validation des formulaires

Utiliser les validateurs prédéfinis

Il existe des validateurs par défaut dans *JSF* (**validateLength**, **validateLongRange** et **validateDoubleRange**), par exemple :

```
<e:inputText id="age" value="#{testController.age}">
  <f:validateLongRange minimum="18"/>
</e:inputText>
<e:message for="age"/>
```

On peut trouver d'autres validateurs que ceux fournis par défaut, *Tomahawk* propose par exemple **validateCreditCard**, **validateUrl**, **validateEmail**, **validateEqual** et **validateRegExpr**.

Exercice 20 : Utiliser un validateur prédéfini

Ajouter un validateur prédéfini à la boite de saisie et supprimer le code correspondant de la méthode **`callback()`** (utiliser le validateur **`validateLength`**)

Validateurs personnalisés

Il est aussi possible d'écrire ses propres validateurs. Leur mise en œuvre est relativement simple, par exemple :

```
<e:inputText
    id="age"
    value="#{testController.age}"
    validator="#{bean.validateAge}">
</e:inputText>
```

La méthode **`validateAge`** de **`bean`** ressemblera à :

```
public void validateAge(
    FacesContext context,
    UIComponent componentToValidate,
    Object value)
throws ValidatorException {
  if (...) {
    throw new ValidatorException(
       getFacesErrorMessage("MESSAGE.ALWAYS_ERROR")) ;
  }
}
```

Exercice 21 : Écrire un validateur

Ajouter un validateur personnalisé **`validateMyInput()`** à la boite de saisie qui se charge de lancer une exception à la place de la méthode **`callback()`**.

Mise à jour de propriétés par les formulaires (`updateActionlistener`)

`updateActionListener` est une balise de la librairie *JSF Tomahawk* qui est particulièrement utile. C'est un *listener* qui est associé à une balise permettant une action (bouton, lien) qui, au moment où ce dernier est activé, va lire le contenu de son attribut **`value`** pour l'assigner à la référence contenue dans son attribut **`property`**.

On utilisera par exemple :

```
<e:commandButton
      action="deleteUser"
      value="#{msgs['BUTTON.DELETE']}">
  <t:updateActionListener
      value="#{user}"
      property="#{controller.userToDelete}" />
</e:commandButton>
```

Ici, l'action **`deleteUser`** va diriger l'utilisateur vers une autre page (typiquement une page de demande de confirmation avant d'effacer l'administrateur), via le fichier de règles de navigation. La présence de la balise **`updateActionListener`** fait que le moteur *JSF* aura, avant d'effectuer le changement de page, positionné la valeur de **`#{user}`** dans l'attribut **`userToDelete`** du contrôleur **`controller`**. La page de confirmation de l'effacement pourra alors faire appel à ce contrôleur pour générer un contenu en fonction de la valeur de cet attribut.

Il est à noter que, dans cet exemple *JSF*, **`user`** est un objet de type complexe et pas simplement une chaîne de caractères comme on peut en avoir l'habitude en développement web classique.

Exercice 22 : Utiliser un **`updateActionListener`**

Faire une vue **`test2.jsp`** qui s'appuie sur un contrôleur **`test2Controller`** qui possède une propriété chaîne de caractères **`value`**. Cette vue **`test2.jsp`** ne fait qu'afficher la valeur de value. Rajouter un bouton **`SetTest2Value`** à la vue **`test1.jsp`** et faire en sorte que l'appuie de ce bouton mettre dans **`test2Controller.value`** la valeur de **`myInput`**.

Note : rajouter au passage une entrée dans la barre de navigation pour aller sur la vue **`test2.jsp`** depuis n'importe quelle page de l'application.


Les opérations sont faites dans cet ordre en *JSF* :

1. Appel des validateurs,
2. Affectation des valeurs des entrées de formulaires,
3. Appel des *updateActionListeners*,
4. Appel de la *callback* d'action du formulaire.

Conversion des types complexes

Il existe des convertisseurs par défaut dans *JSF* (**`DateTimeConverter`** et **`NumberConverter`**). Ils permettent de transformer une date ou un nombre suivant différentes règles, par exemple :

```
<h:outputText value="#{testController.date}">
  <f:convertDateTime
      dateStyle="short"
      locale="#{sessionController.locale}"/>
</h:outputText>
```

Dans certains cas, il est aussi nécessaire de définir des convertisseurs manuellement. C'est notamment le cas pour les listes déroulantes.

Considérons que l'on veuille afficher une liste déroulante permettant à l'utilisateur de choisir un objet de type **`Locale`**.

Pour la génération du *HTML*, *JSF* a besoin d'une représentation textuelle de chaque objet de la liste. De même, il a besoin de pouvoir retrouver un objet depuis un choix de l'utilisateur qui correspond à la valeur textuelle de la balise **`<option>`** du formulaire *HTML*. C'est le rôle du convertisseur **`converter`** de faire ce travail :

```
<e:selectOneMenu
    id="locale"
    onchange="submit();"
    value="#{preferencesController.locale}"
    converter="#{localeConverter}" >
  <f:selectItems value="#{preferencesController.localeItems}" />
</e:selectOneMenu>
```

Ici, la liste déroulante est constituée d'objets de type **`Locale`**.

Le convertisseur **`localeConverter`** est défini (par convention dans *esup-commons*) dans le fichier **`/properties/web/converters.xml`** :

```
<bean
  id="localeConverter"
  class="org.esupportail.commons.web.converters.LocaleConverter">
  <description>
    A converter for Locale objects.
  </description>
</bean>
```

La classe **`LocaleConverter`** implémente l'interface **`Converter`** de JSF.

JSF et accessibilité

Un des objectifs de l'utilisation de *JSF* est, via un standard de haut niveau, de tendre vers plus d'accessibilité (*WAI*).

L'accessibilité des applications doit être une préoccupation constante des programmeurs, qui ne doivent pas hésiter à tester leurs applications en utilisant des navigateurs pauvres, tel *Lynx*.

En règle générale, on évitera absolument d'utiliser les balises **`h:commandLink`** qui à cause de l'utilisation de *Javascript* brisent toutes les règles de l'accessibilité. On préfèrera dans tous les cas des boutons (**`h:commandButton`**).

Javascript peut néanmoins être utilisé pour améliorer l'IHM des applications, en faisant attention à ce que les navigateurs ne parlant pas *Javascript* puisse quand même utiliser l'application. Nous montrons ici à titre d'exemple comment on peut soumettre un formulaire par simple changement de la valeur d'une boite déroulante :

```
<h:form id="form">
  <e:messages />
  <e:outputLabel
      for="value"
      value="#{msgs['TEXT.VALUE']}" />
  <e:selectOneMenu
      id="value"
      onchange="submit();"
      value="#{controller.value}" >
    <f:selectItems
        value="#{controller.valueItems}" />
  </e:selectOneMenu>
  <e:commandButton
      value="#{msgs['BUTTON.CHANGE']}"
      id="changeButton" />
</h:form>
<script type="text/javascript">
  hideButton("form:changeButton");
</script>
```

Le bouton **`changeButton`** est caché par du code *Javascript*.

- Les clients qui parlent *Javascript* ne le voient pas ; ce n'est pas grave puisqu'un changement de valeur de la boite de sélection déroulante soumet automatiquement le formulaire (**`onchange="submit();"`**).
- Les clients qui ne parlent pas *Javascript* voient le bouton, et peuvent donc valider le formulaire.

8. Internationalisation

L'internationalisation est native dans *esup-commons*. L'intérêt n'est pas seulement de fournir une application en plusieurs langages ; l'externalisation de toutes les chaînes de caractères, et la possibilité d'utiliser simultanément plusieurs fichiers de chaînes (les bundles) permet de simplifier la personnalisation des applications par les administrateurs.

8.1. Principes

Configuration

L'internationalisation est définie dans le fichier de configuration **`/properties/i18n/i18n.xml`**. On y trouvera par exemple :

```
<bean
    id="i18nService"
    class="[...].commons.services.i18n.BundlesCachingI18nServiceImpl"
    >
  <property name="bundleBasenames">
   <list>
    <value>properties/i18n/bundles/Commons</value>
    <value>properties/i18n/bundles/Messages</value>
    <value>properties/i18n/bundles/Custom</value>
   </list>
  </property>
 </bean>
```

La propriété **`bundleBasenames`** donne la liste des fichiers de messages à utiliser. Chaque valeur de la liste correspond à la racine du chemin du fichier de message dans le *classpath*. Par exemple, on cherchera pour la valeur **`Custom`** les fichiers **`Custom_<locale>.properties`** ou à défaut **`Custom.properties`**.

Par convention, on groupera tous les fichiers de messages dans le répertoire **`/properties/i18n/bundles`**.

Implémentations disponibles

esup-commons offre plusieurs implémentations afin de gérer l'internationalisation des applications :

- **`BundleI18nServiceImpl`** lit les chaînes de caractères depuis un unique fichier de messages.
- **`BundleCachingI18nServiceImpl`** étend **`BundleI18nServiceImpl`**. Elle permet la mise en cache des chaînes de caractères lues depuis le fichier de messages, pour des raisons de performance.
- **`BundlesI18nServiceImpl`** lit les chaînes de caractères depuis plusieurs fichiers de messages.
- **`BundlesCachingI18nServiceImpl`** étend **`BundlesI18nServiceImpl`** en apportant des fonctionnalités de cache, pour les mêmes raisons de performance.

C'est cette dernière implémentation qui est généralement conseillée car elle permet la personnalisation locale des applications par les administrateurs sans modifier les fichiers distribués par les développeurs.

8.2. Préconisations d'usage

Nommage des bundles

Les fichiers de messages sont lus dans l'ordre où ils apparaissent dans le fichier de configuration. Chaque entrée d'un fichier peut être surchargée par la même entrée définie dans le fichier suivant.

Les fichiers **`Commons_*.properties`** font partie du projet *esup-commons*, ils ne doivent pas être modifiés par les développeurs d'applications.

Si le développeur a besoin de surcharger une entrée de ce fichier, il peut le faire grâce aux fichiers **`Messages_*.properties`**. C'est aussi dans ce fichier qu'il mettra tous les messages relatifs à son application.

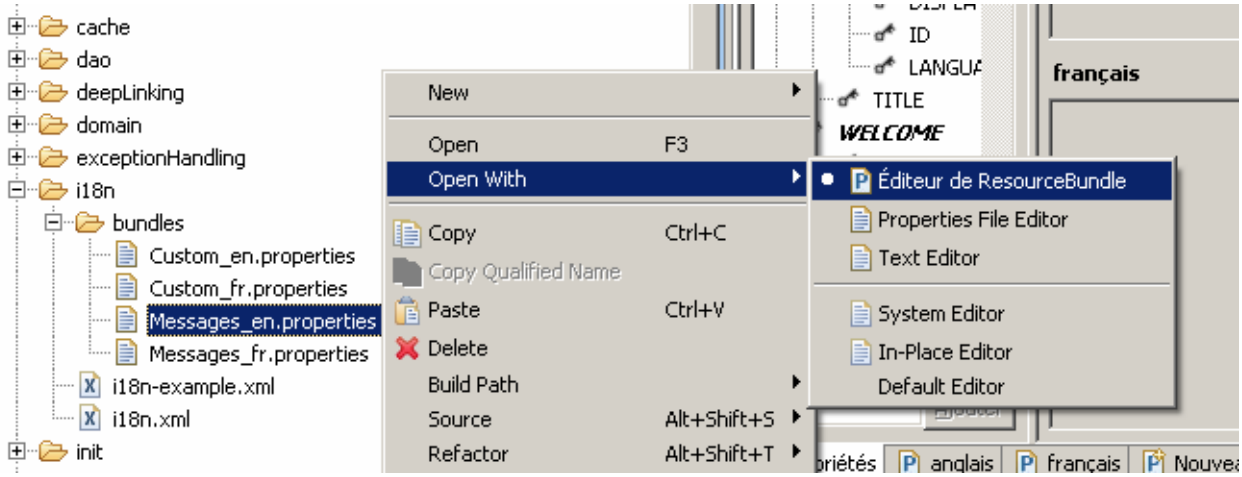
Les fichiers **`Custom_*.poperties`** permettent à l'exploitant de surcharger les messages livrés avec l'application (qu'il s'agisse de messages de *esup-commons* ou de l'application). Typiquement, ce fichier devrait être repris par la tâche *ant* **`recover-config`** lors d'une mise à jour. Pour cela, l'exploitant ajoutera une entrée correspondant à ce fichier dans le paramètre **`custom.recover.files`** de son fichier **`build.properties`**.

Exercice 23 : Surcharger un bundle

Surcharger dans **`Custom_fr.properties`** une des entrées de **`Messages_fr.properties`** et tester.

Modification des fichiers de messages

Pour modifier les fichiers de messages il est recommandé d'utiliser *Ressource Bundle Editor* (*RBE*). Il permet notamment d'éditer plusieurs fichiers, correspondants à plusieurs langues, en même temps.



Les messages enregistrés dans ces fichiers peuvent contenir des paramètres qui seront dynamiquement renseignés lors de l'exécution. Ces paramètres apparaissent dans les messages sous cette forme : **`{n}`** avec **`n`** commençant à **`0`**.

Exemple : **`L''utilisateur {0} est maintenant gestionnaire du service {1}.`**

Notes :

- On note ci-dessus l'échappement des apostrophes lorsque la chaîne contient un ou plusieurs paramètres.
- Il est important que *RBE* soit configuré de la même manière par tous les développeurs d'un même projet.

Ajout d'un langage

L'ajout d'un langage se fait dans le fichier de configuration **`/properties/jsf/application.xml`**. Il suffit ensuite d'écrire le *bundle* correspondant.

Exercice 24 : Ajouter un langage

Ajouter le langage japonais et traduire tout esup-commons ;-)

Externalisation des chaînes sans internationalisation

Comme vu en introduction de cette partie, externaliser les chaînes sans internationaliser conserve un intérêt pour la distribution des applications : faciliter la personnalisation locale des applications là où elles sont portées.

Dans ce cas, on pourra simplement utiliser un seul fichier de ressources, correspondant au langage déclaré.

8.3. Utilisation

Dans une vue (JSF)

La balise **`<e:page>`** du *taglib* de *esup-commons* propose un attribut **`stringsVar`** qui donne le nom du tableau associatif qui contiendra tous les messages. Cette balise propose aussi un attribut **`locale`** qui permet de spécifier la langue à utiliser. Cette information est généralement donnée par un contrôleur. On trouvera par exemple en entête des pages :

```
<e:page stringsVar="msgs" locale="#{controller.locale}" …
```

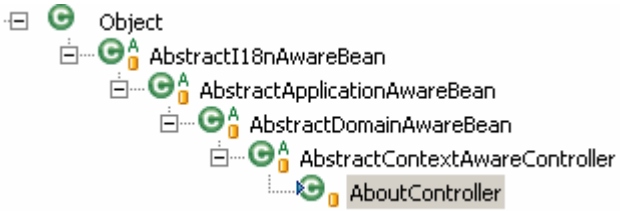
Le tableau de messages **`msgs`** est utilisable dans toute la vue, par exemple :

```
<e:text value="#{msgs['MANAGERS.TEXT.PAGES']}">
  <f:param value="#{controller.paginator.firstVisibleNumber + 1}"/>
  <f:param value="#{controller.paginator.lastVisibleNumber + 1}"/>
  <f:param value="#{controller.paginator.totalItemsCount}"/>
</e:text>
```

Les balises **`<f:param>`** permettent de passer les valeurs des paramètres à insérer dans les messages.

Depuis un contrôleur (Java)

Par héritage, toute classe (métier ou contrôleur) qui étend **`AbstractApplicationAwareBean`** bénéficie des services de **`AbstractI18nAwareBean`** et peut appeler ses méthodes d'internationalisation.



On utilise cela en particulier pour les messages (d'information ou d'erreur) lancés par les contrôleurs et affichés sur les vues *JSF* à l'aide de la balise **`<e:messages>`**. On utilisera par exemple :

```
addErrorMessage(null, "MANAGERS.MESSAGE.USER_NOT_FOUND", userId);
```

On utilisera également cette fonctionnalité à toute autre fin en accédant directement aux fichiers de messages de cette manière :

```
String msg = getString("MY.MESSAGE");
```

On peut également passer des paramètres à la méthode **`getString()`**, ainsi qu'une locale (si l'on souhaite une locale différente de la locale courante), qui est donnée par ma méthode **`getLocale()`** (c'est cette méthode qui est appelée pour passer une locale en paramètre à la balise **`<e:page>`**).

8.4. Changement des messages d'erreur par défaut

(Contribution de Danielle Martineau, Université de Rennes 1)

Pour modifier les messages d'erreurs par défaut ou les internationaliser, on peut :

- écrire un fichier **JsfMessage_fr.properties** dans le répertoire **`/properties/i18n/bundles`**
- ajouter :
  `<message-bundle>properties/i18n/bundles/JsfMessages</message-bundle>`
  au fichier **`/properties/jsf/application.xml`** pour que JSF prenne en charge ce nouveau *bundle*

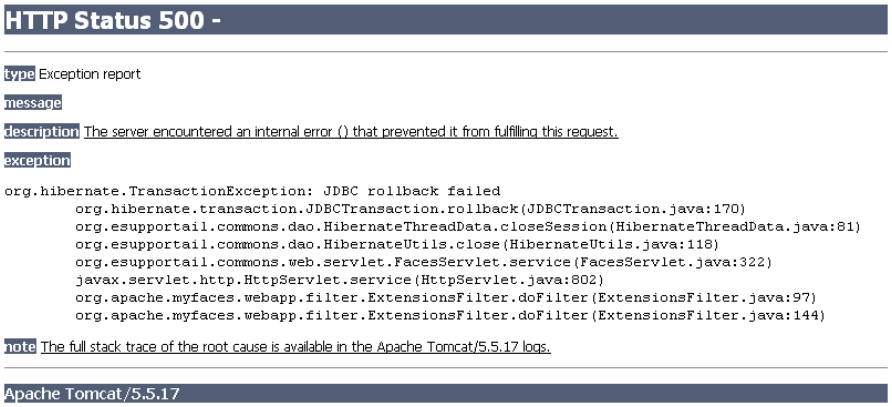
Par exemple lorsque l'on utilise les validations implicites avec l'attribut **`required="true"`** au lieu d'avoir le message « *une donnée est requise* », on pourra avoir un message personnalisé en indiquant les messages suivants dans le fichier **`JsfMessages.properties`** :

```
javax.faces.component.UIInput.REQUIRED        = Erreur de validation
javax.faces.component.UIInput.REQUIRED_detail = "et {0} alors !!!
```

9. Gestion des exceptions

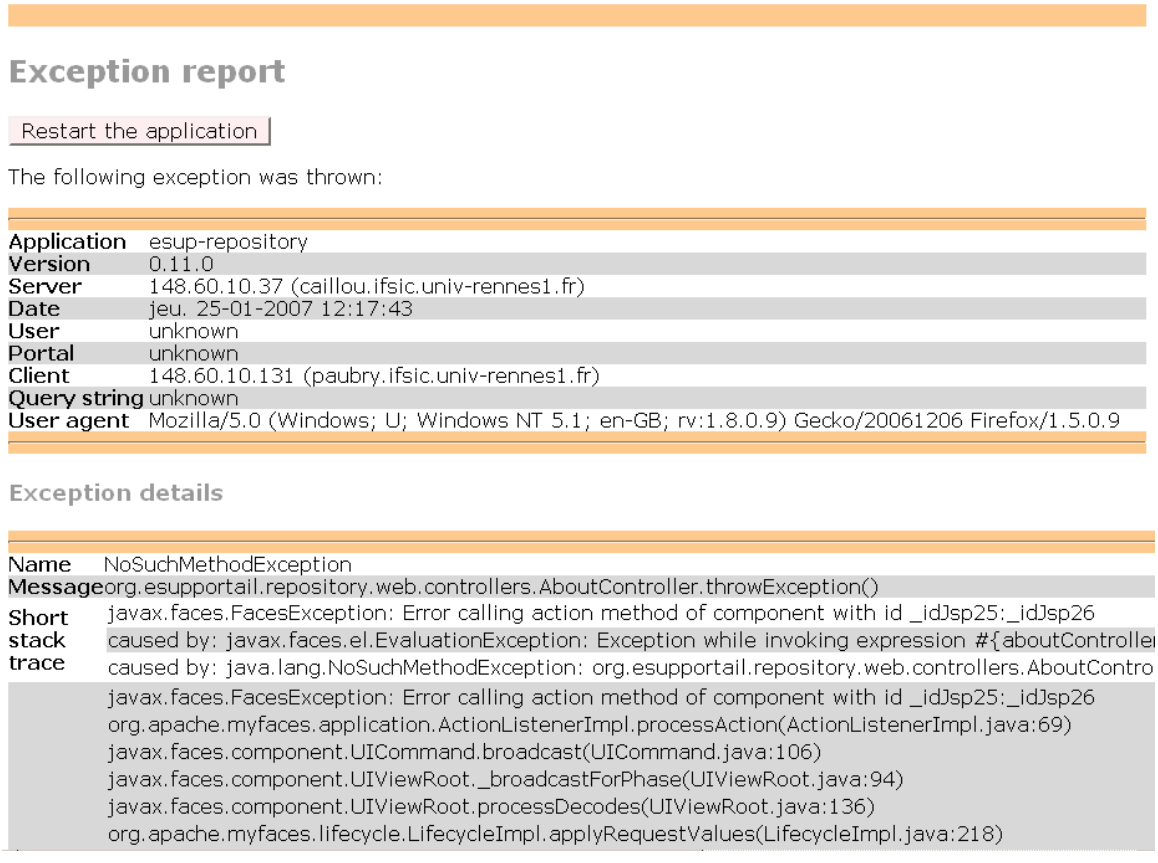
9.1. Généralités

Même dans une application mûre, il n'est de cas pour lesquels une exception ne puisse se produire. *esup-commons* offre une gestion des exceptions qui évite au développeur de recevoir le message traditionnel de *Tomcat* :

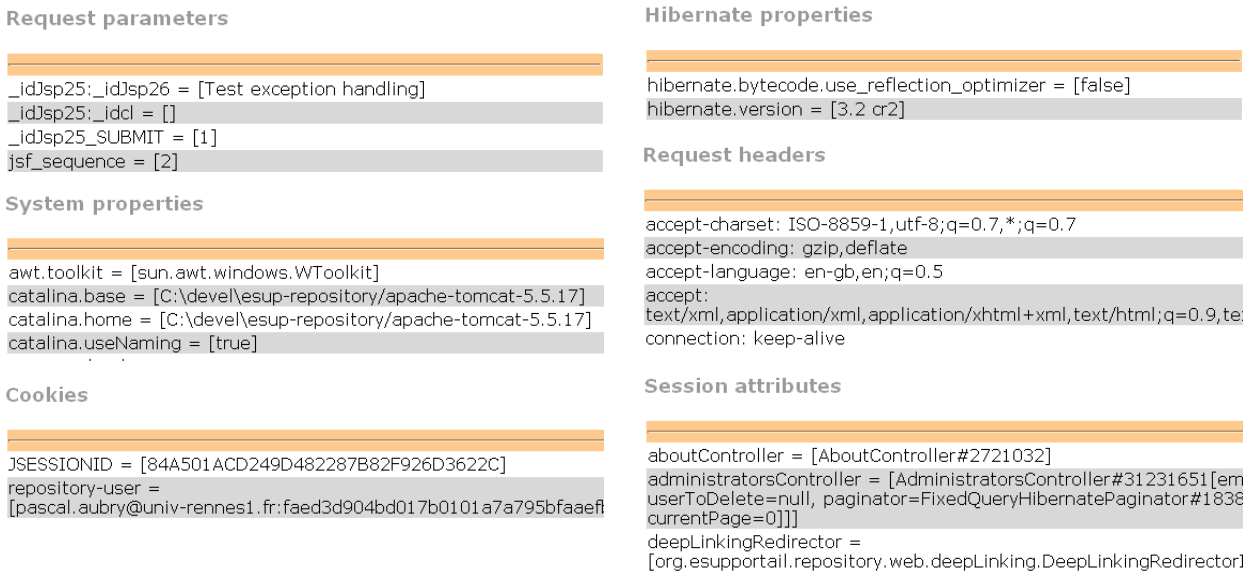


Ou encore celui de *uPortal* :

esup-commons permet de remonter les exceptions de manière propre, par exemple :



Cette gestion des exceptions est une véritable aide au développeur, qui peut ainsi connaître tout le contexte d'exécution au moment où s'est produite l'erreur :



9.2. Configuration

Le gestionnaire d'exception se configure à l'aide du fichier **`/properties/exceptionHandling/exceptionHandling-example.xml`**, dans lequel on déclare le *bean* **`exceptionServiceFactory`**, à partir duquel sont produits les services de gestion des exceptions.

Implémentations disponibles

esup-commons offre plusieurs gestionnaires d'exceptions :

- **`SafeExceptionServiceFactoryImpl`** permet seulement de loguer les exceptions. Cette implémentation n'est pas recommandée, elle est en revanche utilisée par les autres implémentations lorsqu'une exception se produit lors du traitement d'une autre exception.
- **`SimpleExceptionServiceFactoryImpl`** permet de loguer les exceptions mais présente aussi une vue à l'utilisateur avec le contenu de l'erreur. Cette implémentation peut être utilisée en production.
- **`EmailExceptionServiceFactoryImpl`** étend **`SimpleExceptionServiceImpl`** en permettant l'envoi de l'erreur par courrier électronique à une adresse donnée.
- **`CachingEmailExceptionServiceFactoryImpl`** étend **`EmailExceptionServiceFactoryImpl`** en ajoutant des fonctionnalités de cache afin de ne pas envoyer plusieurs fois la même exception et ainsi prévenir des effets de *spam* (quand par exemple la base de données est indisponible, il n'est pas nécessaire d'envoyer plusieurs fois la même exception).

Exemple

On utilisera par exemple :

```
<bean
    id="exceptionServiceFactory"
    class="[...].exceptionHandling.ExceptionServiceFactoryImpl"
    parent="abstractApplicationAwareBean"
  >
  <property
      name="doNotSendExceptionReportsToDevelopers"
      value="false"/>
  <property name="smtpService" ref="smtpService"/>
  <property name="authenticationService" ref="authenticationService"/>
  <property name="recipientEmail" value="webmaster@domain.edu"/>
  <property name="cacheManager" ref="cacheManager"/>
  <property name="logLevel" ref="WARN"/>
</bean>
```

- La propriété **doNotSendExceptionReportsToDevelopers** (facultative, **false** par défaut) permet de désactiver l'envoi des messages d'erreur aux développeurs de l'application.
- La propriété **smtpService** est le service d'envois des messages électroniques.
- La propriété **authenticationService** est le service d'authentification, qui permet d'accéder aux informations relatives à l'utilisateur connecté afin de pouvoir ajouter cette information dans les messages d'erreur. Cette propriété est facultative, en particulier en mode batch.
- La propriété **recipientEmail** est l'adresse mail où seront envoyés les messages d'erreur.
- La propriété **cachemanager** est le gestionnaire de cache.
- La propriété **logLevel** indique le niveau de log des rapports d'exception (**ERROR** par défaut).

Qui reçoit les rapports d'exception ?

La dernière implémentation (**CachingEmailExceptionServiceFactoryImpl**) n'envoie les courriers électroniques qu'à l'adresse spécifiée dans la configuration *Spring*.

Lorsque cette classe est étendue dans *esup-blank* pour, en plus d'une adresse fixée par configuration, envoyer les exceptions à une adresse fixée dans le code (celles des développeurs). Cela permet de remonter automatiquement les exceptions aux développeurs, qui peuvent ainsi suivre à distance les problèmes sur les applications déployées. Même lorsque la classe **CachingEmailExceptionServiceFactoryImpl** est étendue, les exploitants de l'application disposent de la propriété **doNotSendExceptionReportsToDevelopers** pour activer ou désactiver cette fonctionnalité.

Vue utilisée pour les rapports d'exceptions

SimpleExceptionServiceFactoryImpl (ainsi que les classes qui en héritent) présente une vue à l'utilisateur avec le contenu de l'erreur.

Cette vue peut être configurée en utilisant la propriété **exceptionView** du *bean* **exceptionServiceFactory**. Si cette propriété n'est pas définie, alors la vue **/stylesheets/exception.jsp** de *esup-commons* est utilisée par défaut.

La vue proposée par défaut, présente des détails qui ne sont certainement pas utiles à l'utilisateur. Il convient avant de mettre en production une application de personnaliser la vue **/stylesheets/exception.jsp** (en la copiant dans le projet de l'application) ou bien d'en créer une autre (par exemple **/stylesheets/customizedException.jsp**) et d'indiquer au *bean* **exceptionServiceFactory** de l'utiliser (à l'aide de la propriété **exceptionView**).

Il est possible de copier la vue **/stylesheets/exception.jsp** de *esup-commons* dans votre projet pour l'adapter et configurer votre gestionnaire d'exception pour pointer vers cette nouvelle vue.

Exercice 25 : Changer la vue des exceptions

Copier **exception.jsp** en **exception-prod.jsp** et simplifier la page pour ne garder que les informations qui vous intéressent. Configurer l'application pour utiliser cette nouvelle vue en cas d'exception.

Redémarrage de l'application.

La vue **/stylesheets/exception.jsp** présente à l'utilisateur un bouton permettant de redémarrer l'application. Ce bouton appelle la méthode **restart()** du contrôleur **exceptionController** déclaré dans le fichier de configuration **/properties/web/controllers.xml**.

Exemple de déclaration :

```
<bean id="exceptionController"
  class="org.esupportail.commons.web.controllers.ExceptionController"
  scope="session">
  <property name="resettableNames">
   <list>
    <value>sessionController</value>
    <value>administratorsController</value>
    <value>preferencesController</value>
    <value>welcomeController</value>
    <value>ldapSearchController</value>
   </list>
  </property>
 </bean>
```

La propriété **resettableNames** donne la liste des noms des contrôleurs qui seront réinitialisés par la méthode **restart()** de **exceptionController.** Cette réinitialisation est faite en appelant la méthode **reset()** des *beans* (qui doivent impérativement implémenter l'interface **Resettable**).

En conséquence, tout *bean* (contrôleur, …) qui doit être réinitialisé lors du redémarrage de l'application doivent implémenter l'interface **Resettable**.

Exercice 26 : Réinitialiser un contrôleur après une exception

Faire en sorte que sur une exception, l'attribut **value** de test2Controller soit vidé.

Utilisation de plusieurs vues d'exceptions

Il est possible depuis la version 0.17.2 d'utiliser plusieurs vues d'exceptions en fonction de l'exception levée par l'application :

```
<property name="exceptionViews" >
  <map>
    <entry
      key="org.esupportail.commons.exceptions.WebFlowException"
      value="/stylesheets/webFlowException.jsp" />
    <entry
      key="org.esupportail.commons.exceptions.ConfigException"
      value="/stylesheets/configException.jsp" />
    <entry
      key="java.lang.Exception"
      value="/stylesheets/exception.jsp" />
  </map>
</property>
```

Ne pas envoyer de courrier électronique pour certaines exceptions

Il est possible depuis la version 0.17.3 de ne pas envoyer de courrier électronique pour certaines exceptions, en ajoutant la propriété suivante :

```
<property name="noEmailExceptions" >
  <list>
    <value>
      org.esupportail.commons.exceptions.WebFlowException
    </value>
  </list>
</property>
```

10. Accès aux données

Le principal apport de *esup-commons* est de décharger le développeur de la gestion de l'accès aux données. Avec esup-commons, il n'est plus nécessaire pour lui de se soucier de l'ouverture/fermeture des sessions, ni même de savoir s'il faut valider ou non certaines transactions.

En effet, tout cela est pris en charge au niveau le plus haut (les points d'entrées de *esup-commons* que sont **`FacesServlet`**, **`FacesPortlet`**, **`XFireServlet`** et les commandes *batch*) ; le développeur peut ainsi se concentrer sur le développement de l'application elle-même.

10.1. Le modèle : one-session-per-request, one-session-per-command

On considère comme atomique les opérations suivantes :

- Une requête HTTP de l'utilisateur,
- Une commande batch de l'administrateur.

Cela veut dire que s'il se produit quoi que ce soit d'imprévu (une exception) au cours d'une de ces opérations atomiques, alors on rejette l'opération complète, sinon on la valide. On est ainsi sûr de la cohérence de la base de données.

De manière schématique, une de ces opérations, en interne dans esup-commons, se traduit par la succession des actions suivantes :

5. Commencer
6. Ouvrir ou récupérer une connexion aux bases de données
7. Ouvrir une session
8. Ouvrir une transaction
9. Faire ce qu'il y a à faire
10. Annuler la transaction (*rollback*, en cas d'exception) ou la valider (*commit*, sinon)
11. Fermer la session
12. Finir

10.2. Le fonctionnement

Les points d'entrée

Le modèle présenté ci-dessus est implémenté par les points d'entrée de esup-commons (**`FacesServlet`**, **`FacesPortlet`**, **`XFireServlet`**), en couplage avec la gestion des exceptions.

Déploiement en servlet

En déploiement *servlet*, il faut utiliser la *servlet* **`FacesServlet`** offerte par *esup-commons*, et qui gère les sessions *Hibernate* et les exceptions (cf **`/webapp/WEB-INF/web-servlet-example.xml`**) :

```
<servlet>
  <display-name>Faces Servlet</display-name>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    org.esupportail.commons.web.servlet.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

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

Déploiement en portlet

En mode *portlet*, il faut utiliser la *servlet* **PortletServlet** offerte par *esup-commons*, et qui gère également les sessions *Hibernate* et les exceptions (cf **/webapp/WEB-INF/web-portlet-example.xml**) :

```
<servlet>
  <display-name>esup-application</display-name>
  <servlet-name>esup-application</servlet-name>
  <servlet-class>
    org.esupportail.commons.web.portlet.PortletServlet
  </servlet-class>
  <init-param>
    <param-name>portlet-class</param-name>
    <param-value>
      org.esupportail.commons.web.portlet.FacesPortlet
    </param-value>
  </init-param>
  <init-param>
    <param-name>portlet-guid</param-name>
    <param-value>esup-diskquota.esup-diskquota</param-value>
  </init-param>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>esup-application</servlet-name>
  <url-pattern>/esup-application/*</url-pattern>
</servlet-mapping>
```

Web services

En mode *web service*, il faut utiliser la *servlet* **XFireServlet** offerte par *esup-commons*, qui gère également les sessions *Hibernate* et les exceptions (cf. **/webapp/WEB-INF/web-*-example.xml**) :

```
<servlet>
  <servlet-name>xfire</servlet-name>
  <servlet-class>
    org.esupportail.commons.web.servlet.XFireServlet
  </servlet-class>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>xfire</servlet-name>
  <url-pattern>/xfire/*</url-pattern>
</servlet-mapping>
```

Cela implique que les opérations effectuées soient unitaires, au sens transactionnel.

Accès batch

Les accès *batch* sont traités en détail dans un chapitre dédié.

La configuration de l'accès aux données

Les points d'entrées ci-dessus utilisent la classe statique **DatabaseUtils** qui, pour pouvoir gérer plusieurs connexions à des bases de données, s'appuie sur un « magasin de gestionnaires de bases de données ».

Ce magasin, qui recense les gestionnaires de bases de données, est un *bean* nommé **databaseManagerStore**, qui doit implémenter l'interface **DatabaseManagerStore**. Une seule implémentation est fournie de base (**BasicDatabaseManagerStoreImpl**), elle doit permettre de répondre à toutes les demandes.

On trouvera par exemple dans le fichier **/properties/dao/dao.xml** :

```
<bean
    id="databaseManagerStore"
    class="[...].services.database.BasicDatabaseManagerStoreImpl" >
  <property name="databaseManagers">
    <list>
      <ref bean="databaseManager1" />
      <ref bean="databaseManager2" />
    </list>
  </property>
</bean>
```

Le magasin recense ici deux gestionnaires de bases de données (**databaseManager1** et **databaseManager2**).

Les gestionnaires de bases de données sont chargés de l'accès à une base de données ; ce sont également des *beans*, qui doivent implémenter l'interface **DatabaseManager**. Les méthodes de cette interface sont :

- **void open(servletContext)** : ouvre la connexion à la base de données et commence une transaction ; cette méthode est utilisée en mode web.
- **void open()** : ouvre la connexion à la base de données et commence une transaction ; cette méthode est utilisée en mode *batch*.
- **void close(boolean)** : valide (*commit*) ou invalide (*rollback*) la transaction courante et ferme la connexion.
- **void test()** : teste la connexion; cette méthode est utilisée par la tâche *ant* **test-database**.
- **boolean isUpdatable()** : indique si le gestionnaire est capable de créer et mettre à jour la structure de la base de données (on pourra dans ce cas appeler les méthodes **create()** et **update()**).
- **void create()** : crée les structures de la base de données.
- **void update()** : met à jour les structures de la base de données.

Note : les utilisateurs ne s'appuyant sur aucune base de données devront utiliser l'implémentation **EmptyDatabaseManagerStoreImpl** pour le *bean* **databaseManagerStore**.

L'accès aux données depuis du code Java

Le service métier (le *bean* **domainService**) accède aux données via le service d'accès aux données (le *bean* **daoService**). Il est à la charge des implémentations de l'interface **DaoService** de récupérer les sessions courantes des bases données et d'effectuer les requêtes nécessaires.

La méthode utilisée par les implémentations *Hibernate* est détaillée dans la partie suivante.

10.3. Gestion de la structure de la base de données

Création de la structure de la base de données

La base de données est créée à l'aide de la tâche ant **init-data**. Cette tâche efface toute la base de données puis appelle la méthode **initDatabase()** du *bean* **versionningService**. C'est par exemple à cette occasion que l'on pourra créer les premiers objets de l'application.


Les données effacées par la tâche **`init-data`** sont irrécupérables !

Mise à jour de la structure de la base de données

La structure de la base de données est mise à jour à l'aide de la tâche ant **`upgrade`**. Cette tâche rajoute les champs nécessaires, s'assure que toutes les contraintes sont bien positionnées, puis appelle la méthode **`upgradeDatabase()`** du bean **`versionningService`**. C'est par exemple à cette occasion que l'on pourra initialiser les valeurs des champs de table nouvellement créés.

10.4. Accès aux données avec Hibernate

Par défaut, les applications *esup-blank* et *esup-example* sont configurées avec les gestionnaires de base de données s'appuyant sur *Hibernate*.

Les gestionnaires de bases de données

Le choix du gestionnaire de base de données Hibernate doit se faire en fonction de la maîtrise qu'a le développeur de la base de données. Deux cas se présentent :

13. **Le développeur a la totale maîtrise de la base de données**, c'est lui qui la fait évoluer en fonction des besoins de son application. Il utilisera dans ce cas l'implémentation **`UpdatableHibernateDatabaseManagerImpl`**. Cette implémentation lui permettra de faire évoluer la structure de sa base de données en modifiant son *mapping*, de manière automatique sans même toucher au code SQL.
14. **Le développeur n'a pas la maîtrise de la base de données** : il s'agit d'une base de données institutionnelle, ou bien encore d'une base maîtrisée par une autre application. Il utilisera dans ce cas l'implémentation **`BasicHibernateDatabaseManagerImpl`**, et devra alors faire coller son mapping aux structures de la base de données.

On trouvera par exemple dans le fichier **`/properties/dao/dao.xml`**, pour une base maîtrisée par l'application :

```
<bean
    id="databaseManager1"
    class="[...].hibernate.UpdatableHibernateDatabaseManagerImpl" >
  <property
    name="sessionFactoryBeanName"
    value="sessionFactory" />
  <property
    name="createSessionFactoryBeanName"
    value="createSessionFactory" />
  <property
    name="updateSessionFactoryBeanName"
    value="updateSessionFactory" />
</bean>
```

Comme on le voit, on indique au gestionnaire de base de données quelles sont les *session factories* (« usines à sessions ») à utiliser pour accéder à la base de données, en créer les structures et les mettre à jour.

Note : il s'agit bien des noms des *beans* des *session factories*, et non des références aux *beans* (car la simple instanciation de ces *beans* provoque la mise à jour de la structure de la base de données. Ils sont donc déclarés en **`lazy-init="true" et seront instanciés par l'application –via DatabaseUtils- et non par Spring`**).

La déclaration d'un gestionnaire de base de données dont l'application n'a pas la maîtrise se fera de la manière suivante :

```
<bean
    id="databaseManager2"
    class="[...].hibernate.BasicHibernateDatabaseManagerImpl" >
  <property
    name="sessionFactoryBeanName"
    value="sessionFactory" />
</bean>
```

Les session factories (« usines à session »)

Les trois *beans* **sessionFactory**, **createSessionFactory** et **updateSessionFactory** (le premier seulement si l'on ne maîtrise pas la structure de la base de données) héritent tous du même *bean* abstrait **abstractHibernateSessionfactory** :

```
<bean
    id="abstractHibernateSessionFactory"
    abstract="true"
    class="[...].orm.hibernate3.LocalSessionFactoryBean" >
  <property
    name="configLocation"
    value="classpath:/properties/dao/hibernate/hibernate.cfg.xml" />
  <property name="mappingLocations">
    <list>
      <value>
        classpath:/properties/dao/hibernate/mapping/Class1.hbm.xml
      </value>
      ...
    </list>
  </property>
</bean>
```

Cette déclaration indique que la configuration de l'accès physique à la base de données se trouve dans le fichier **/properties/dao/hibernate/hibernate.cfg.xml**, et que les *mappings* des classes se trouvent dans le répertoire **/properties/dao/hibernate/mapping**.

Le lecteur se reportera à la documentation Hibernate en ce qui concerne le contenu du fichier **/properties/dao/hibernate/hibernate.cfg.xml**.

Les *beans* **sessionFactory**, **createSessionFactory** et **updateSessionFactory** précisent simplement leur mode d'accès particulier (la propriété **hibernateProperties** surcharge les valeurs données dans le fichier **/properties/dao/hibernate/hibernate.cfg.xml**).

```
<bean id="sessionFactory"
    parent="abstractHibernateSessionFactory"
    lazy-init="true" >
</bean>
```

```
<bean  id="createSessionFactory"
    parent="abstractHibernateSessionFactory"
    lazy-init="true" >
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.hbm2ddl.auto">create</prop>
    </props>
  </property>
</bean>
```

```
<bean id="updateSessionFactory"
    parent="abstractHibernateSessionFactory"
    lazy-init="true" >
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>
</bean>
```

Les trois modes de Hibernate sont utilisés par esup-commons :

- L'accès normal à la base de données s'appuie sur le *bean* **sessionFactory** (accès web et accès *batch*),
- La création se fait en instanciant le *bean* **createSessionFactory** (via la tâche *ant* **init-data**),
- La mise à jour se fait en instanciant le *bean* **updateSessionFactory** (via la tâche *ant* **upgrade**).

Mapping avec la base de données

Les fichiers de *mapping* sont dans le répertoire **/properties/dao/hibernate/mapping** et portent, par convention, l'extension **.hbm.xml**

Voici ici l'exemple d'une classe **Entry**, qui appartient au *package* **org.esupportail.formation.domain** et a comme attributs **id**, **value** et **date** (de types **long**, **java.lang.String** et **java.sql.Timestamp**) :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="org.esupportail.formation.domain.beans">
  <class name="Entry" table="d_entry">
    <id name="id" type="long">
      <column name="id"/>
      <generator class="native" />
    </id>
    <property name="value" type="string">
      <column name="valu" length="250"
              not-null="true" unique="true"/>
    </property>
    <property name="date" type="timestamp">
      <column name="dat" not-null="true"/>
    </property>
  </class>
</hibernate-mapping>
```

Dans l'exemple, **id** est la clé primaire dans la table. **id** est dissocié de **value** qui a aussi une contrainte d'unicité dans la base de données. C'est une des bonnes pratiques pour *Hibernate* que de ne donner aucune notion métier à la clé primaire en base pour garantir l'évolutivité de l'application.

Utilisation de HQL

HQL est un langage d'interrogation de base de données de *Hibernate*. Il est orienté objet et a une forme proche du SQL mais travaille sur les objets *Java* définis dans vos fichiers de *mapping* et pas sur des noms de tables de votre base de données.

Note : *Hibernate* propose aussi une autre méthode d'interrogation, dite « requêtes par critères ». Il s'agit d'une *API* et plus d'un langage comme *HQL*. Cette dernière n'est pas abordée dans ce document.

Contrairement à SQL, le mot clé **SELECT** de HQL n'est pas obligatoire. S'il n'est pas utilisé ce sont des objets qui sont retournés. S'il est utilisé il est possible de seulement retourner les propriétés de ces objets.

Après le mot-clé **FROM** on n'utilise pas de noms de tables mais des noms de classes. Ce nom de classe est sensible à la casse comme en *Java*. Le nom du *package* n'est pas obligatoire car *Hibernate* a un mécanisme d'*auto-import*.

Exemple de requête *HQL* simple qui récupère toutes les instances de la classe **Thing** dans la base de données :

```
"FROM Thing"
```

Avec *HQL* il est possible de faire des jointures (mot clé **JOIN**), d'utiliser le mot clé **WHERE** afin de limiter les objets à retourner par la requête. Il est aussi possible d'utiliser des fonctions d'agrégation (**sum(…)**, **count(*)**) et des expressions (**upper()**, **+**, **between**) dans les requêtes *HQL*. De même, le support de sous-requêtes et des clauses **ORDER BY** et **GROUP BY** est offert.

Pour plus de renseignements sur la construction des requêtes *HQL*, le lecteur se reportera à la documentation de *Hibernate*.

Dans esup-commons HQL est notamment utilisé pour :

- Compter le nombre d'instances persistantes d'une classe (nombre de lignes dans une table) avec la méthode **getQueryIntResult**. Par exemple :
  ```
  getQueryIntResult("select count(*) from Entry");
  ```
- Sélectionner les objets à faire apparaître dans un paginateur *Hibernate.*

Parmi les autres recommandations pour *Hibernate* on trouvera le fait de devoir surcharger les méthodes **hashCode()** et **equals()**, par exemple :

```
public boolean equals(final Object obj) {
  if (obj == null) {
    return false;
  }
  if (!(obj instanceof Entry)) {
    return false;
  }
  return id == ((Entry) obj).getId();
}

public int hashCode() {
  return super.hashCode();
}
```

Exercice 27 : Modifier le *mapping Hibernate*

Écrire la classe **Entry**. Ecrire le fichier de *mapping* correspondant et le référencer depuis **dao.xml**. Exécuter la tâche *ant* **init-data** et vérifier que la table correspondante a bien été créée dans la base de données.

Comment ça marche

Comme vu plus haut, les points d'entrée de *esup-commons* s'assurent du respect du modèle *one-session-per-request*.

Lors de l'ouverture d'une session Hibernate à une base de données, on appelle le code suivant :

```
TransactionSynchronizationManager.bindResource(
    sessionFactory, new SessionHolder(session));
```

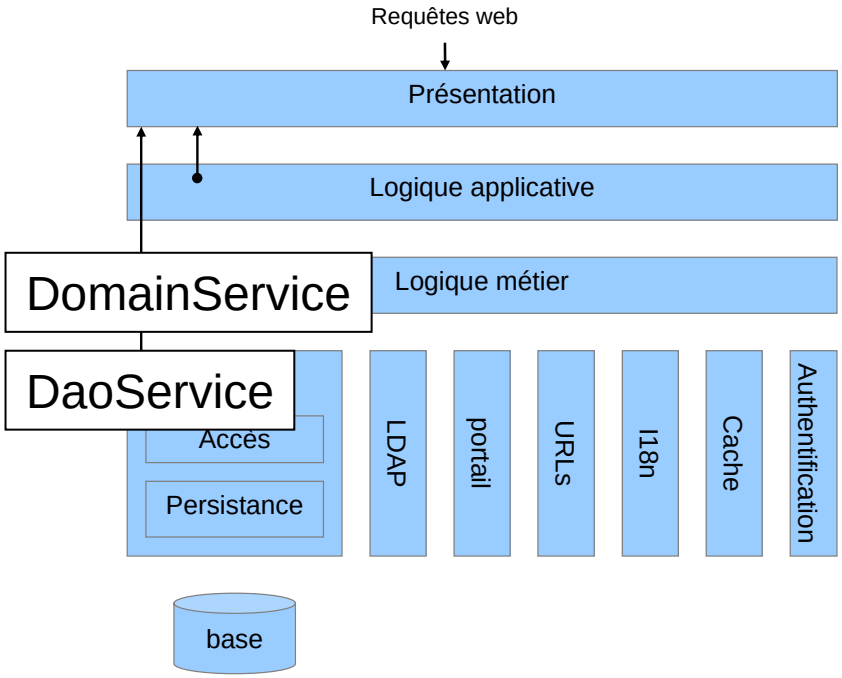
Cela associe la session créée (**session**) à l'usine à sessions (**sessionFactory**) qui l'a créée, pour le *thread* courant.

Pour accéder aux données, le service d'accès aux données **HibernateDaoService** utilise la méthode **getHibernateTemplate()** (héritée de la classe **HibernateDaoSupport**). Cette méthode **getHibernateTemplate()** récupère la session courante dans les données du *thread* courant à partir de l'usine à session (**sessionFactory**), injectée dans le *bean* **daoService** par *Spring*.

Ce mécanisme ingénieux permet d'éviter toute adhérence entre le service métier et le service d'accès aux données. La classe statique **DatabaseUtils** permet de réduire toute adhérence entre les points d'entrée de esup-commons et les gestionnaires de données utilisés.

10.5. Écriture du code d'accès aux objets métiers

esup-commons préconise le développement en couches :



Accès aux données

Pour pouvoir avoir accès à l'objet **Entry** donné en exemple au paragraphe précédent la couche métier doit disposer de méthodes dans l'interface **DaoService**.

Cela se traduira, par exemple, par deux méthodes permettant, d'une part, d'ajouter une **Entry** en base de donnée et, d'autre part de récupérer les objets **Entry** depuis la base de données :

```
void addEntry(Entry entry);
List<Entry> getEntries();
```

Ces deux méthodes de l'interface doivent se retrouver dans son implémentation concrète (par exemple **HibernateDaoService**).

Service métier

Ensuite, ce sont les contrôleurs qui vont avoir besoin de dialoguer avec la couche métier.

A nouveau, nous allons passer par une interface (**DomainService**) qui, par exemple, définira ces deux méthodes :

```
Entry addEntry(String value)
List<Entry> getEntries()
```

Ces deux méthodes seront implémentées dans une classe concrète (par exemple **`DomainServiceImpl`**).

Note : ici la couche métier n'apporte pas beaucoup de service. Typiquement, la méthode **`addEntry`** prend en paramètre une chaînes de caractères, crée une instance de la classe **`Entry`**, (**`new Entry()`**) lui donne la valeur passée en paramètre et la date courante puis la passe à la couche de persistance (la date courante sera obtenue grâce à **`new Timestamp(System.currentTimeMillis())`**).

Exercice 28 : Implémenter les méthodes d'accès aux données

Écrire les méthodes ci-dessus dans les classes ad hoc.

Exercice 29 : Ajouter une entrée dans la base de données

Faire en sorte que l'appui sur le bouton de **`test1.jsp`** enregistre la valeur de **`test1Controller.myInput`** comme une nouvelle instance de **`Entry`** dans la base de données.

Contrôler la présence d'une nouvelle entrée dans la base de données à chaque clic.

Exercice 30 : Afficher sur une page *JSF* une liste de données de la base

Afficher les objets de type **`Entry`** de la base de données sous forme d'une liste en dessous du bouton de **`test1.jsp`**.

On doit pour cela parcourir les entrées à l'aide de :

```
<t:dataList value="#{test1Controller.entries}" var="entry">
  <e:li value="#{entry.value} (#{entry.date})" />
</t:dataList>
```

10.6. Accès aux données avec Ibatis

Toute dépendance entre *esup-commons* et *Hibernate* a été supprimée en version 0.13.0. Une implémentation *Ibatis* des gestionnaires de bases de données est en cours de réflexion.

10.7. Applications sans base de données

Les applications qui ne s'appuient pas sur une base de données doivent tout simplement déclarer le *bean* **`databaseManagerStore`** suivant :

```
<bean
     id="databaseManagerStore"
     class="[…].services.database.EmptyDatabaseManagerStoreImpl"
/>
```

11. Pagination

Il n'est pas rare dans les applications web de vouloir afficher des listes de valeurs. Parce que l'espace visible sur l'écran du client est limité, on souhaite souvent afficher les résultats sur plusieurs pages ; c'est ce qu'on appelle la pagination.

La **première difficulté** de la pagination est donc de **n'afficher qu'une partie des résultats d'un ensemble plus important**, et de proposer une navigation visuelle entre les pages.

La **deuxième difficulté** consiste, pour les accès aux bases de données, à **ne récupérer que les valeurs de la base qui doivent être affichées** ; cela est indispensable lorsque le nombre de résultat est énorme, assez gros en tout cas pour saturer la mémoire des processus.

Nous montrons donc dans un premier temps comment on peut paginer des données sans se soucier de leur récupération. La dernière partie montre comment écrire des paginateurs qui ne récupèrent de la base de données que les données à afficher.

11.1. Écriture d'un paginateur simple

Nous supposons qu'une page de l'application doit afficher les choses (**`Thing`**) du service (**`Department`**) courant de la page.

Notre paginateur (**`ThingPaginator`**) étend la classe abstraite **`ListPaginator<Thing>`**, qui implémente elle-même l'interface **`Paginator<Thing>`**. :

```
public class ThingPaginator extends ListPaginator<Thing> {
  ...
}
```

Son constructeur positionne l'attribut **`domainService`** (le service métier) qui sera utilisé ultérieurement pour récupérer les choses (**`Thing`**) de la base de données :

```
public ThingPaginator(final DomainService domainService) {
  super(null, 0);
  this.domainService = domainService;
}
```

Un autre attribut **`department`** est utilisé pour mémoriser le service duquel le paginateur doit récupérer les choses. Cet attribut est positionné par le *setter* correspondant :

```
public ThingPaginator setDepartment(final Department department) {
  this.department = department;
  return this;
}
```

Le paginateur implémente enfin la méthode de récupération des données proprement dite :

```
protected List<Thing> getData() {
  return this.domainService.getThings(department);
}
```

Le paginateur ainsi écrit peut être utilise par un contrôleur.

Note : les paginateurs ne s'appuient pas forcément sur une base de données ; on peut ainsi imaginer un paginateur qui traitera les fichiers trouvés dans un répertoire donné. Dans ce cas, le service métier n'est pas nécessaire.

Exercice 31 : Écrire un paginateur simple

Écrire un paginateur simple pour récupérer tous les objets de type **`Entry`** de la base de données.

11.2. Utilisation d'un paginateur

Dans le code Java

Un paginateur sera typiquement un attribut d'un contrôleur. Nous prenons ici pour exemple le contrôleur des « choses », qui les affiche de manière paginée :

```
private ThingPaginator paginator;
```

La méthode **reset()** est appelée automatiquement par la méthode **afterPropertiesSet()** de la classe **AbstractDomainAwareController**. Dans cette méthode, on initialise le paginateur et on charge ses première données (avec un service vide, la liste des choses récupérées sera vide) :

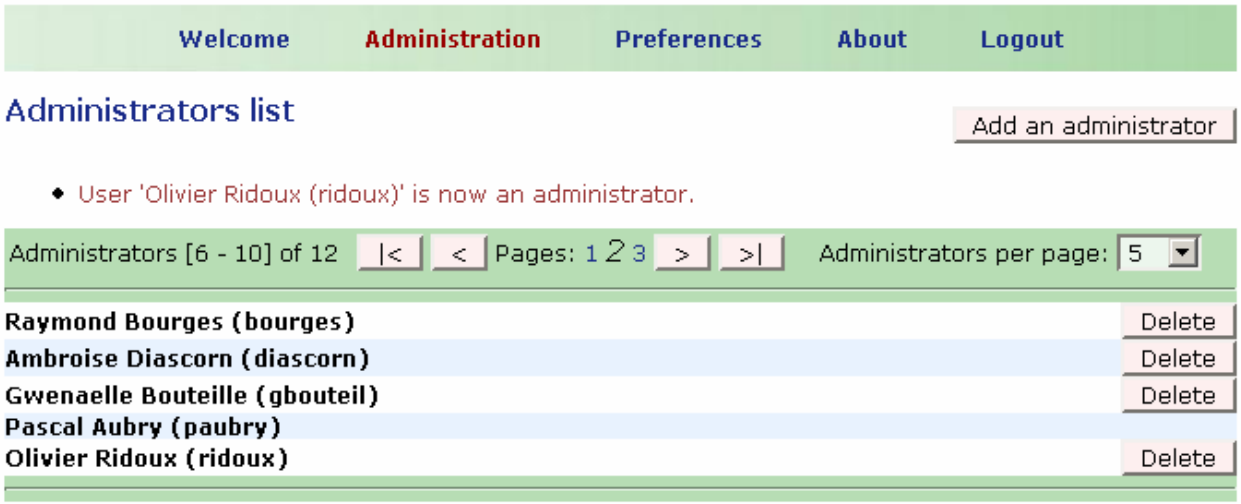
```
paginator = new ThingPaginator(getDomainService());
paginator.loadData();
```

à chaque fois que le service de la page change, il suffit d'en informer le paginateur et de recharger ses données (la méthode **reloadData()** provient de l'interface **Paginator**) :

```
paginator.setDepartment(department).reloadData();
```

Dans une page JSF

Nous montrons dans cette partie comment présenter un paginateur, pour obtenir un affichage de ce genre :



On commence par englober le tout d'un formulaire, nécessaire pour faire fonctionner les boutons de navigation :

```
<h:form id="administratorsForm">
```

On commence ensuite une table pour parcourir les entrées du paginateur (on suppose ici que le paginateur est un attribut **paginator** d'un contrôleur **controller**) :

```
<e:dataTable
    rendered="#{not empty controller.paginator.visibleItems}"
    id="data"
    rowIndexVar="variable"
    value="#{controller.paginator.visibleItems}"
    var="thing" border="0"
    style="width:100%"
    cellspacing="0"
    cellpadding="0">
```

L'attribut **rendered** fait que cette table ne sera affichée que lorsque le paginateur a des éléments visibles. A chaque tour de boucle, la variable **thing** parcourt la liste **controller.getPaginator().getVisibleItems()**.

On ajoute ensuite à la table une entête, sous forme d'un *facet* **`<f:facet name="header">`**. C'est ce *facet* qui contiendra :

- Les numéros des éléments affichés,
- Les liens vers les pages proches de la page courante
- Une boite de dialogue déroulante permettant de changer le nombre d'éléments affichés par page.

On parcourt ensuite des colonnes dans lesquelles on affiche ce que l'on veut, par exemple :

```
<t:column>
  <e:text value="#{thing.value}" />
</t:column>
```

Exercice 32 : Afficher un paginateur sur une page *JSF*

Afficher le paginateur écrit à l'exercice précédent sur la vue **`test1.jsp`** (à la place de la liste précédemment).

Note : il est avantageux de partir d'un exemple existant pour afficher un paginateur, on pourra par exemple se référer au fichier **`/webapp/stylesheets/administrators.jsp`** du projet *esup-blank*.

11.3. *Écriture d'un paginateur* Hibernate

L'intérêt d'un paginateur *Hibernate* est de ne récupérer de la base de données que les éléments qui doivent être affichés.

Cela se fait en étendant la classe **`AbstractHibernatePaginator<E>`**, qui possède la méthode abstraite suivante :

```
String getQueryString();
```

Cette méthode doit retourner la requête *HQL* (*Hibernate Query Language*) qui correspond à la récupération de l'ensemble des éléments, sans tenir compte de ceux qui seront affichés ou non (la classe s'occupe ensuite de limiter les éléments récupérés).

Lorsque cette requête *HQL* est invariable, il est possible d'étendre la classe **`FixedAbstractHibernatePaginator<E>`**, qui initialise sa requête *HQL* par son constructeur.

Un exemple de requête *HQL* fixe est **`"FROM Thing"`**, qui récupère simplement toutes les instances de la classe **`Thing`** dans la base de données.

Exercice 33 : Écrire un paginateur *Hibernate*

Écrire un paginateur *Hibernate* pour récupérer les objets de type **`Entry`** de la base de données et les afficher dans la page **`test1.jsp`** (à la place du paginateur simple).

12. Accès à l'annuaire LDAP

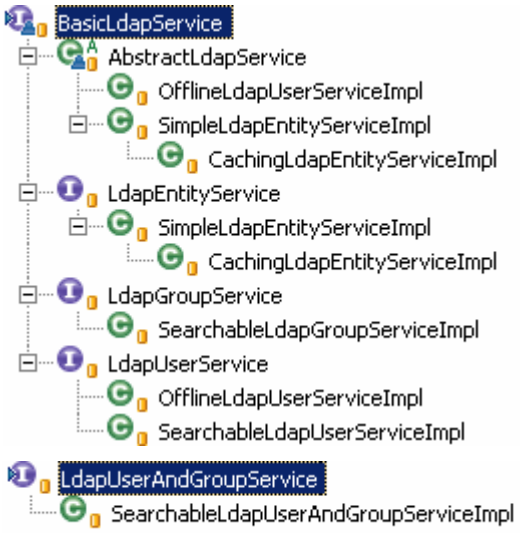
Le lien vers le SI de l'établissement se fait en particulier vers l'annuaire *LDAP*. *esup-commons* offre des fonctionnalités concernant les utilisateurs, les groupes et de manière plus générale n'importe quelle entité LDAP :

- Recherche d'utilisateurs,
- Recherche des attributs d'un utilisateur donné,
- Validation d'un filtre pour un utilisateur donné,
- Recherche de groupes,
- Recherche des sous-groupes d'un groupe,
- Recherche des membres d'un groupe,
- Validation d'un filtre pour un groupe donné,
- Recherche d'entités,
- Recherche des attributs d'une entité donnée,
- Validation d'un filtre pour une entité donnée.

Note : les fonctionnalités offertes sont beaucoup mois ambitieuses que celle de l'actuel canal annuaire, l'accent a été mis sur la facilité de mise en œuvre. En revanche, il ne tient qu'à un développeur motivé d'enrichir le source actuel pour amener les fonctionnalités manquantes.

Tous les *beans* concernant *LDAP* sont définis dans le fichier de configuration **`/properties/ldap/ldap.xml`**.

De nombreuses classes sont disponibles :



12.1. Manipulation des utilisateurs LDAP

Nous ne détaillons dans ce document que la manipulation des utilisateurs *LDAP*. Il sera facile au lecteur d'étendre les principes montrés ci-dessous à la manipulation des groupes et des entités *LDAP* quelconques.

Utilisation basique

L'implémentation utilisée pour manipuler seulement les utilisateurs LDAP est **`SearchableLdapUserServiceImpl`**.

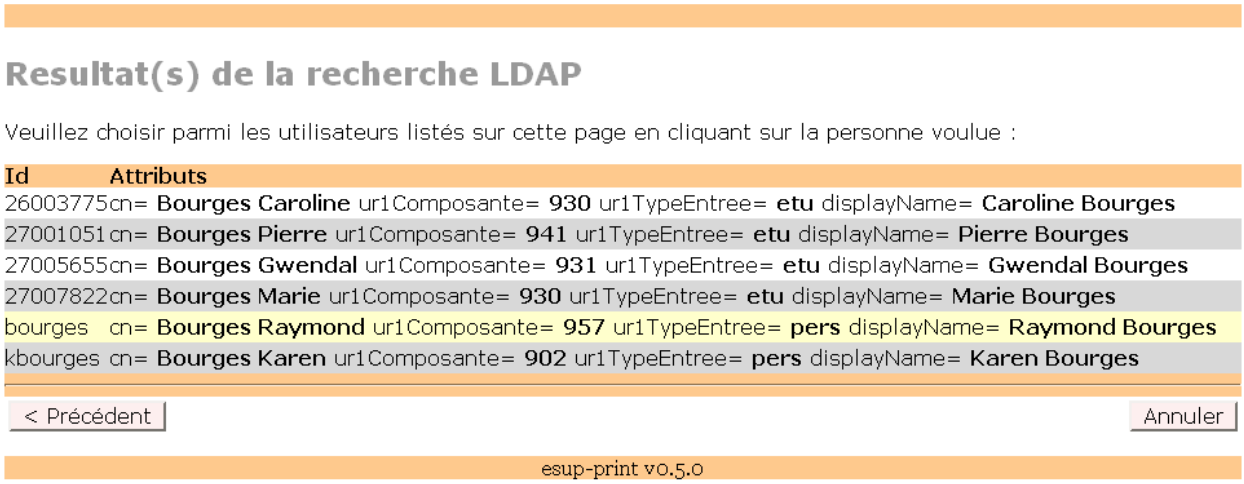
La déclaration d'un *bean* **ldapUserService** de cette classe ressemblera à :

```
<bean
  id="ldapUserService"
  class="[...].commons.services.ldap.SearchableLdapUserServiceImpl"
  >
  <property name="ldapTemplate" ref="ldapTemplate" />
  <property name="uidAttribute" value="uid" />
  <property name="searchAttribute" value="cn" />
  <property name="searchDisplayedAttributes">
    <list>
      <value>displayName</value>
      <value>ur1composante</value>
      <value>ur1typeEntree</value>
    </list>
  </property>
  <property name="otherAttributes">
    <list>
      <value>homeDirectory</value>
    </list>
  </property>
  <property name="testFilter" value="cn=*bourges*" />
</bean>
```

La propriété **uidAttribute** donne le nom de l'attribut *LDAP* qui contient l'identifiant unique des utilisateurs de l'annuaire.

La propriété **searchAttributes** donne le nom de l'attribut sur lequel on effectue des recherches. Dans l'exemple ci-dessous, la recherche du mot **bourges** utilisera le filtre **cn=*bourges***.

La propriété **searchDisplayAttributes** donne les noms des attributs qui seront affichés à l'utilisateur lors du choix d'un utilisateur parmi plusieurs (après une recherche dans l'annuaire). Par exemple :



La propriété **otherAttributes** donne les noms des attributs qui seront remontés lors des requêtes *LDAP*, pour être utilisés dans du code *Java*.

La propriété **testFilter** est utilisée par la tâche *ant* **test-ldap**.

Cette classe s'appuie sur la bibliothèque *LdapTemplate* :

```
<bean id="ldapTemplate" class="net.sf.ldaptemplate.LdapTemplate">
  <property name="contextSource" ref="contextSource" />
</bean>

<bean
    id="contextSource"
    class="net.sf.ldaptemplate.support.LdapContextSource">
  <property name="url" value="ldap://ldap.esup-portail.org:389" />
  <property name="userName" value="" />
  <property name="password" value="" />
  <property name="base" value="ou=people,dc=esup-portail,dc=org"/>
  <property name="baseEnvironmentProperties">
    <map>
      <entry key="com.sun.jndi.ldap.connect.timeout" value="5000" />
    </map>
  </property>
</bean>
```

Mise en cache des requêtes LDAP

Il est possible d'injecter un gestionnaire de cache aux instances de cette classe pour leur faire cacher le résultat des requêtes afin de moins solliciter l'annuaire *LDAP* (pour plus de performances).

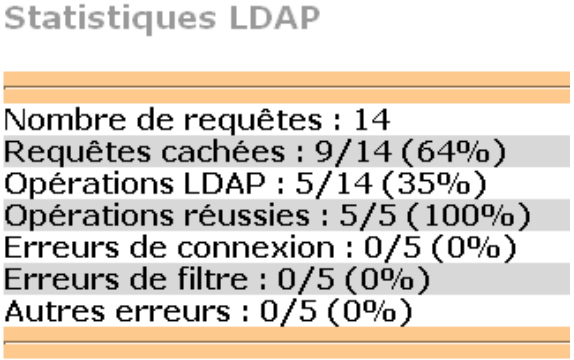
Il suffit de rajouter à la déclaration précédente :

```
<bean
  id="ldapUserService"
  class="[...].commons.services.ldap.SearchableLdapUserServiceImpl"
  >
  ...
  <property name="cacheManager" ref="cacheManager" />
  <property name="cacheName" value="" />
</bean>
```

Le *bean* **cacheManager** est en général défini dans le fichier de configuration **/properties/cache/cache.xml**. Le nom du cache est optionnel.

Accès aux statistiques LDAP

La classe **SearchableLdapUserServiceImpl** supporte également la récupération de statistiques sur son utilisation, sous forme d'une liste de chaînes de caractères internationalisées (d'où la nécessité du service d'internationalisation **i18nService**). Ces chaînes peuvent par exemple être affichées sur l'interface d'une application web :



Accès LDAP hors connexion

De la même manière que la classe **`OfflineFixedUserAuthenticator`** pour l'authentification, la classe **`OfflineLdapServiceImpl`** permet de travailler hors connexion :

```
<bean
  id="ldapService"
  class=" [...].commons.services.ldap.OfflineLdapServiceImpl"
  />
```

Applications sans accès LDAP

Les applications sans accès à l'annuaire LDAP doivent tout simplement ne pas déclarer de *bean* **`ldapUserService`**.

Utilisation du service LDAP depuis du code Java

Exceptions

Les exceptions lancées par les appels aux méthodes de **`LdapUserService`** peuvent lancer les exceptions suivantes :

- **`LdapConnectionException`**, lorsque l'annuaire LDAP est inaccessible,
- **`LdapBadFilterException`**, lorsqu'un mauvais filtre est utilisé,
- **`LdapMiscException`**, pour toute autre erreur.

Il appartient au programmeur d'attraper ou non ces exceptions en fonction du contexte de l'application.

Recherche d'un utilisateur par son identifiant

Pour rechercher un utilisateur *LDAP* à partir d'un identifiant par programmation, il faut appeler le service *LDAP* de la manière suivante :

```
String uid = "bourges";
LdapUser user = ldapService.getLdapUser(uid);
```

Cette méthode lance l'exception **`UserNotFoundException`** si l'utilisateur est introuvable.

Exercice 34 : Chercher un utilisateur dans l'annuaire *LDAP*

Faire en sorte que l'appui du bouton de **`test1.jsp`** n'ajoute une entrée dans la base que si la valeur de **`myInput`** correspond bien à un utilisateur de l'annuaire *LDAP*. Afficher un message d'erreur sinon (utiliser un message prédéfini de esup-commons).

Une manière de faire possible est l'utilisation d'un validateur, par exemple :

```
public void validateMyInput(
            FacesContext context,
            UIComponent componentToValidate,
            Object value) throws ValidatorException {
    String string = (String) value;
    try {
      LdapUser ldapUser = ldapService.getLdapUser(string);
    } catch (UserNotFoundException e) {
      throw new ValidatorException(
          getFacesErrorMessage(
             "LDAP_SEARCH.MESSAGE.NO_RESULT", string));
        }
    }
}
```

Note : on injectera le *bean* **`ldapService`** dans le contrôleur **`Test1Controller`** pour qu'il puisse accéder aux services *LDAP*.

Recherche des utilisateurs correspondant à un motif

Pour rechercher les utilisateurs *LDAP* à partir d'un motif, il faut appeler le service *LDAP* de la manière suivante :

```
String token = "ourge" ;
List<LdapUser> users = ldapService.getLdapUsersFromToken(token);
```

La liste retournée peut éventuellement être vide.

Recherche des utilisateurs à partir d'un filtre LDAP

Pour rechercher les utilisateurs *LDAP* correspondant à un filtre *LDAP*, il faut appeler le service *LDAP* de la manière suivante :

```
String filter = "&(departmentNumber=univ*)(employeeType=student)" ;
List<LdapUser> users = ldapService.getLdapUsersFromFilter(filter);
```

La liste retournée peut éventuellement être vide.

Test d'un filtre LDAP

Pour tester la syntaxe d'un filtre *LDAP*, il faut appeler le service *LDAP* de la manière suivante :

```
String errorMessage = ldapService.testLdapFilter(filter);
```

Le filtre est valide si le message d'erreur retourné est **`null`**.

Vérification d'un filtre LDAP *pour un utilisateur*

Pour savoir si un utilisateur vérifie un filtre *LDAP*, il faut appeler le service de la manière suivante :

```
Boolean matched = ldapService.userMatchesFilter(uid, filter);
```

La méthode retourne **`true`** si l'utilisateur vérifie le filtre.

Récupération des statistiques LDAP

La récupération des statistiques des accès *LDAP* se fait en appelant le service *LDAP* de la manière suivante :

```
List<String> strings = ldapService.getStatistics(locale);
```

Notes :

- Il faut auparavant s'assurer que l'implémentation du service supporte la récupération des statistiques en appelant **`ldapService.supportStatistics()`**.
- Les statistiques peuvent être réinitialisées en appelant la méthode **`ldapService.resetStatistics()`**.

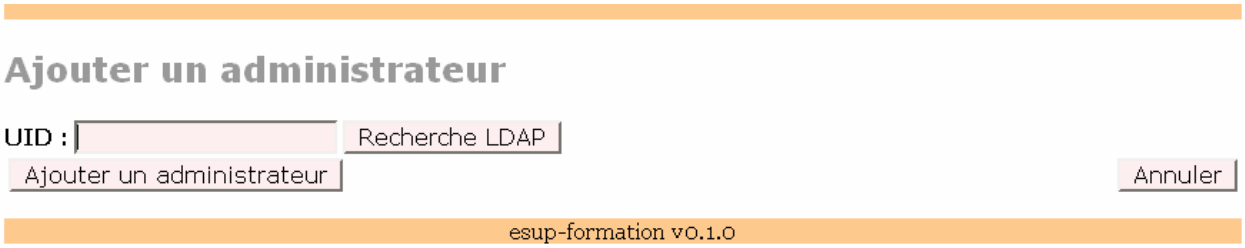
Intégration de la recherche d'utilisateurs dans les pages JSF

Nous montrons ici comment simplement inclure dans les pages web d'une application la recherche d'un utilisateur *LDAP*.

Cinématique

Nous prenons ici le cas d'une page d'administration, qui permet de gérer les administrateurs d'une application. L'utilisateur de l'application doit pouvoir ajouter des administrateurs en cherchant dans un annuaire *LDAP*.

La page d'ajout, **`administratorAdd.jsp`**, ressemble à :



On souhaite que l'appui sur le bouton « Recherche *LDAP* » amène sur la page de recherche suivante (**`ldapSearch.jsp`**) :

L'appui sur « Suivant » doit afficher la page **`ldapSearchResults.jsp`** suivante :

L'utilisateur doit alors pouvoir sélectionner un des utilisateurs (en cliquant dessus) et revenir sur la page **`administratorAdd.jsp`**, en remplissant sa boite de dialogue.

Toute cette cinématique est disponible de base dans *esup-commons*, nous allons détailler cet exemple pour bien comprendre son fonctionnement.

Page appelante

Le contrôleur **administratorsControler**, chargé de toutes les interactions avec l'utilisateur pour la partie « administration » de l'application, implémente l'interface **LdapCaller**. Il possède donc une méthode **setLdapUid()** qui pourra être appelée par le contrôleur de la recherche *LDAP* en cas de succès. Cela remplira d'ailleurs automatiquement la boîte de dialogue de **administratorAdd.jsp** puisque celle-ci est liée à la propriété **ldapUid** du contrôleur **administratorsControler** :

```
<e:inputText
    id="ldapUid"
    value="#{administratorsController.ldapUid}"
    required="true" />
```

Le source *JSF* du bouton « recherche *LDAP* » est le suivant :

```
<e:commandButton
    value="#{msgs['_.BUTTON.LDAP']}"
    action="ldapSearch"
    immediate="true">
  <t:updateActionListener
      value="#{administratorsController}"
      property="#{ldapSearchController.caller}" />
  <t:updateActionListener
      value="userSelectedToAdministratorAdd"
      property="#{ldapSearchController.successResult}" />
  <t:updateActionListener
      value="cancelToAdministratorAdd"
      property="#{ldapSearchController.cancelResult}" />
</e:commandButton>
```

Fonctionnement

Lorsque l'utilisateur appuie sur ce bouton, l'application :

- Indique au contrôleur **ldapSearchController** que l'appelant est le contrôleur **administratorsController** en positionnant sa propriété **caller** (de type **LdapCaller**) pour appeler sa méthode **setLdapUid()** en cas de recherche fructueuse),
- Positionne la propriété **successResult** (resp. **cancelResult**) du contrôleur **ldapSearchController** à **userSelectedToAdministratorAdd** (resp. **cancelToAdmlinistratorAdd**). Cette valeur sera utilisée par le contrôleur **ldapSearchController** comme code de retour de la callback de sélection des utilisateurs (resp. de la callback d'annulation de la recherche).

La recherche *LDAP* proposée par *esup-commons* possède son propre code *Java* (le contrôleur **ldapSearchController**) et ses propres pages *JSF* (**ldapSearch.jsp** et **ldapSearchResults.jsp**), dont il n'est pas nécessaire de connaître le contenu pour les utiliser.

Règles de navigation

Il faut néanmoins indiquer à *JSF* les règles de navigation entre les différentes pages, comme indiqué ci-après (dans **/properties/jsf/navigation-rules.xml**).

L'appui du bouton « Recherche *LDAP* » depuis la page **administratorAdd.jsp** envoie vers la page **ldapSearch.jsp** :

```
<navigation-rule>
  <display-name>administratorAdd</display-name>
  <from-view-id>/stylesheets/administratorAdd.jsp</from-view-id>
  <navigation-case>
    <from-outcome>adminAdded</from-outcome>
    <to-view-id>/stylesheets/administrators.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>cancel</from-outcome>
    <to-view-id>/stylesheets/administrators.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>ldapSearch</from-outcome>
    <to-view-id>/stylesheets/ldapSearch.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Sur la page **ldapSearch.jsp**, l'appui du bouton « Annuler » ramène sur la page **administratorAdd.jsp**, l'appui sur le bouton « Suivant » passe à la page **ldapSearchResults.jsp** s'il y a des résultats (**usersFound**) ; s'il n'y a pas de résultat, le résultat de la *callback* du bouton « Suivant » est **null** et la page reste inchangée.

```
<navigation-rule>
  <display-name>ldapSearch</display-name>
  <from-view-id>/stylesheets/ldapSearch.jsp</from-view-id>
  <navigation-case>
    <from-outcome>usersFound</from-outcome>
    <to-view-id>/stylesheets/ldapResults.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>cancelToAdministratorAdd</from-outcome>
    <to-view-id>/stylesheets/administratorAdd.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Enfin, sur la page **ldapSearchResults.jsp**, l'appui du bouton « Annuler » (**cancelToAdministratorAdd**) ainsi que la sélection d'un utilisateur parmi la liste proposée (**userSelectedToAdministratorAdd**) ramènent tous les deux à la page **administratorAdd.jsp** (dans le deuxième cas, la méthode **setLdapUid()** du contrôleur **administratorController** aura été appelée) :

```
<navigation-rule>
  <display-name>ldapResults</display-name>
  <from-view-id>/stylesheets/ldapResults.jsp</from-view-id>
  <navigation-case>
    <from-outcome>userSelectedToAdministratorAdd</from-outcome>
    <to-view-id>/stylesheets/administratorAdd.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>cancelToAdministratorAdd</from-outcome>
    <to-view-id>/stylesheets/administratorAdd.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>back</from-outcome>
    <to-view-id>/stylesheets/ldapSearch.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

En procédant de cette manière, le développeur peut faire appel à la recherche LDAP en utilisant le même contrôleur (**`ldapSearchController`**) depuis plusieurs pages différentes (et donc plusieurs contrôleurs associés).

12.2. Manipulation des groupes LDAP

La manipulation des groupes *LDAP* se fait généralement en utilisant un bean **`ldapGroupService`** implémentant l'interface **`LdapGroupService`**, par exemple de la classe **`SearchableLdapGroupServiceImpl`**.

Cette classe manipule des objets implémentant l'interface LdapGroup :

```
LdapGroup getLdapGroup(String id) ;;
boolean groupMatchesFilter(String id, String filter);
List<LdapGroup> getLdapGroupsFromToken(String token);
List<LdapGroup> getLdapGroupsFromFilter(String filterExpr);
List<String> getSearchDisplayedAttributes();
```

12.3. Manipulation des utilisateurs et des groupes LDAP

esup-commons offre une classe qui permet de manipuler en même temps les utilisateurs et les groupes *LDAP*, **`SearchableLdapUserAndGroupServiceImpl`**.

En plus des méthodes offertes par les classes **`SearchableLdapUserServiceImpl`** et **`SearchableLdapGroupServiceImpl`**, cette classe permet de récupérer les utilisateurs membres d'un groupe :

```
List<String> getMemberIds(LdapGroup group);
List<LdapUser> getMembers(LdapGroup group);
```

12.4. Manipulation des entités LDAP quelconques

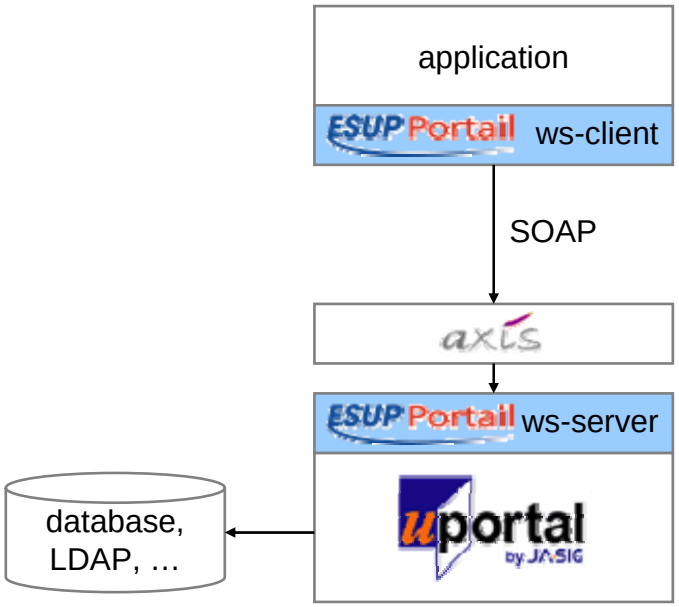
esup-commons permet enfin de manipuler n'importe quel type d'entité LDAP, à l'aide de la classe **`CachingLdapEntityServiceImpl`**.

13. Accès aux informations du portail

Une des problématiques en mode *portlet* est liée au fait qu'une *portlet* s'exécute dans un contexte (*Tomcat*) différent de celui du portail. Il est ainsi impossible pour l'application d'accéder aux objets du portail, notamment pour récupérer les groupes et attributs utilisateurs (comme on le faisait avec de simple canaux *uPortal* (implémentations de l'interface **`IChannel`**).

13.1. Principe

esup-commons s'appuie sur le projet *esup-portal-ws* (http://sourcesup.cru.fr/esup-portal-ws) pour l'accès aux informations du portail. Celui est basé sur un *web service* :



Le *JAR* de la partie cliente de *esup-portal-ws* est intégrée à *esup-commons*. L'application a donc à toutes les fonctionnalités de *esup-portal-ws*, pourvu que la partie serveur soit installée sur le portail auquel l'application va demander des informations.

Le lecteur se reportera à la documentation de *esup-portal-ws* pour savoir comment configurer la partie serveur.

13.2. Utilisation

Afin d'accéder aux groupes et attributs utilisateurs du portail, le développeur doit configurer le *bean* **`portalService`** dans le fichier de configuration *Spring* **`/properties/portal/portal.xml`**. Le *bean* **`portalService`** doit implémenter l'interface **`PortalService`**, qui possède les méthodes suivantes :

Une fois injecté (via *Spring*, par exemple dans un contrôleur), le *bean* **`portalService`** donne accès à toutes les informations voulues. Voici quelques exemples d'utilisation :

```
PortalService portalService = (PortalService)
beanFactory.getBean("portalService");
PortalGroup rootGroup = portalService.getRootGroup();
PortalGroup group = portalService.getGroupById("local.10");
PortalUser user = portalService.getUser("paubry");
List<String> values = portalService.getUserAttributeValues(
                                        "paubry", "departmentNumber");
if (portalService.isUserMemberOfGroup("paubry", "local.10")) {
    ...
}
```

Voici enfin un exemple de configuration du *bean* **portalService**, en utilisant une implémentation qui offre des fonctionnalités de cache pour plus de performances :

```
<bean
    id="portalService"
    class="[...].ws.client.support.uportal.CachingUportalServiceImpl"
    >
<property
    name="url"
    value="http://localhost:8080/services/UportalService" />
<property name="testUserId" value="paubry" />
<property name="testGroupId" value="local.0" />
<property
    name="testGroupName"
    value="Tous les groupes de personnes" />
<property name="cacheManager" ref="cacheManager" />
</bean>
```

Notes :

- L'attribut **url** est l'*URL* sur laquelle le *web service* du portail est exposé, via *Axis*.
- Les attributs **testUserId** et **testGroupId** servent à tester le *web service*, via la tâche *ant* **test-portal**.

Dans le cas où le développeur ne souhaite pas utiliser l'accès aux informations du portail, il utilisera la configuration suivante :

```
<bean
    id="portalService.servlet"
    class="[...].services.portal.NotSupportedPortalServiceImpl" />
```

Dans ce cas, l'utilisation du *bean* **portalService** provoquera une exception.

Note : aucun exercice n'est prévu sur ce point car l'installation de *esup-portal-ws* sort de la portée de cette formation.

14. Numérotation des versions

esup-commons résout le problème de la cohérence entre les versions de l'application et de la base sur laquelle elle s'appuie, en particulier dans les environnements cluster de nos portails.

14.1. A quoi correspondent les numéros de version ?

Ensuite, les numéros de version sont toujours de la forme **`x.y.z-t`**, maintenu dans le fichier **`/properties/misc/application.xml`** :

- **`x`** : le n° majeur de version, peut être changé en cas d'ajouts/changements de fonctionnalités importants, ou pour des refontes majeures des applications. Par convention, le changement de n° majeur peut impliquer pour l'utilisateur des manipulations manuelles pour la mise à jour, alors que toutes les autres mises à jour doivent se faire de manière (semi-) automatique.
- **`y`** : le n° mineur de version, qui correspond en général à un niveau de fonctionnalités. Pour un changement de n° mineur, l'exploitant est toujours invité à exécuter la tâche *ant* **`uprade`**, qui met à jour la structure de la base. C'est à cette occasion que le développeur peut faire exécuter des mises à jour supplémentaires, par exemple sur les données elles-mêmes, en allant modifier le *bean* **`versionningService`**.
- **`z`** : le n° de *patch*. Par convention, on conserve le n° de patch lorsque l'on ne modifie pas les fonctionnalités. à noter qu'une mise à jour pour un même n° de version mineur se fait de manière automatique, sans nécessiter l'appel de la tâche *ant* **`uprade`**.
- **`t`** : le n° de packaging, qui change seulement lorsque l'on modifie la documentation, le déploiement ou les fichiers d'exemple.

A chaque fois que l'on modifie la structure de la base, il suffit d'incrémenter le n° de version mineur : *esup-commons* force l'appel de ant **`upgrade`** (en lançant une exception) tant que le n° mineur de la base et celui de l'application ne sont pas les mêmes.

L'appel de la tâche **`upgrade`**, en plus de mettre à jour la structure de la base de données, peut également être l'occasion de mettre à jour des données à l'aide de code Java exécuté par le *bean* **`versionningService`** (défini dans **`/properties/init/init.xml`**), qui est appelé par la tâche **`upgrade`**.

Note : le *bean* **`versionningService`** est également appelé de manière automatique par l'application lorsque seul le n° de *patch* diffère. Il est donc possible également dans ce cas d'exécuter du code de mise à jour des données.

Exercice 35 : Créer une incompatibilité de version entre application et base

Rajouter un attribut **`value2`** à la classe **`Entry`**, et modifier le *mapping* en conséquence. Incrémenter le n° de version mineur de l'application. Redéployer et vérifier que l'application n'est plus accessible.

Note : le service métier pourra laisser l'attribut **`value2`** à la valeur **`null`**.

Exercice 36 : Mettre à jour la base de données

Exécuter la tâche *ant* **`upgrade`**. Constater les modifications dans la structure de la base de données et vérifier que l'application est de nouveau accessible.

14.2. Comment ne pas gérer les numéros de version ?

Lorsque l'on ne s'appuie pas sur une base de données, on ne veut pas avoir à gérer les problèmes de cohérence du numéro de version entre application et base de données.

Il suffit alors de s'appuyer sur un *bean* **`versionningService`** ne faisant rien du tout, par exemple en le déclarant de la classe **`VoidVersionningServiceImpl`**.

14.3. Comment faciliter les mises à jour ?

Distribuer une mise à jour d'un logiciel est en général chose facile, d'autant plus avec *esup-commons* car il suffit d'appeler une tâche *ant*.

La mise à jour sur une plateforme de production est en général beaucoup plus délicate à cause de la nécessité de maintenir la cohérence des numéros de version de l'application et de la base de données. Ce point est résolu par le *bean* **`versionningService`**, comme montré plus haut. Elle est également fastidieuse, car tous les fichiers de configuration doivent être recopiés depuis l'installation antérieure.

Ce dernier problème est résolu à l'aide la tâche ant **`recover-config`**, qui récupère automatiquement les anciens fichiers de configuration et les installe dans la nouvelle instance de l'application. Nous décrivons dans cette partie comment fonctionne la tâche **`recover-config`**, et ce que doit faire le développeur pour qu'elle fonctionne correctement.

Comment la tâche `recover-config` fonctionne-t-elle ?

Toutes les versions de l'application sont sensées être installées au même niveau sur le système de fichiers, ce qui fait que l'application sait qu'elle doit rechercher dans le répertoire immédiatement supérieur (**`../`**).

La propriété **`${recover.previous-versions}`** est une liste de versions (séparées par des virgules), dont la tâche **`recover-config`** va chercher la présence automatiquement. Dans la pratique, pour chaque numéro de version **`x.y.z`**, la tâche recherche les répertoires **`../application-x.y.z`** et **`../application-quick-start-x.y.z`**.

Une fois le répertoire trouvé, la tâche récupère les fichiers indiqués par la propriété **`${recover.files}`** et les recopie dans la nouvelle instance de l'application.

Comment un développeur fait-il fonctionner la tâche `recover-config` ?

Le développeur doit tout d'abord mettre à jour la propriété **`${recover.previous-versions}`** à chaque fois qu'il incrémente le n° de version de l'application, sans quoi les versions non indiquées par la propriétés ne seront pas prises en compte.

Il doit ensuite indiquer quels sont les fichiers qui doivent être récupérés par la tâche.

Comme dit plus haut, les fichiers de configuration à récupérer sont donnés par la propriété **`${recover.files}`**, qui elle même construite à partir de **`${commons.recover.files}`** et **`${app.recover.files}`**. Le développeur doit donc indiquer quels sont les fichiers de configuration de l'application qu'il souhaite récupérer automatiquement à l'aide de **`${app.recover.files}`**, à configurer dans le fichier **`build.xml`**. On trouvera par exemple :

```
<property name="app.recover.files" value="
    properties/init/init.xml
    properties/domain/domain.xml
    properties/i18n/bundles/NetworkAppliance_*.properties
    properties/i18n/bundles/Custom_*.properties
    " />
```

Comment un administrateur utilise-t-il la tâche `recover-config` ?

Il suffit d'installer la nouvelle version au même niveau que la précédente sur le système de fichiers ; c'est un prérequis. L'exécution de la tâche **`recover-config`** suffit ensuite pour récupérer les fichiers de configuration automatiquement (le numéro de l'ancienne version est détectée automatiquement).

Pour récupérer d'autres fichiers que ceux explicitement prévus par le développeur, il suffit de spécifier ces fichiers dans la propriété **`${custom.recover.files}`**, par exemple :

```
custom.recover.files=\
webapp/media/enabled.jpg \
properties/applicationContext.xml \
properties/local/local.xml \
src/edu/domain/application/domain/*.java
```

15. Distribuer une application

esup-commons prévoit des tâches ant afin de distribuer votre application.

Au sommet de ces tâches *ant* on trouve la tâche **_dist** qui se charge de générer la documentation des sources (*Javadoc*) du projet (et de *esup-commons*) pour la déposer dans le répertoire **/docs/api**. Ensuite, la tâche construit le site web de l'application en intégrant le contenu du répertoire **/docs** (**api** y compris). Le site web est aussi mis sous forme d'un **zip** dans le répertoire **/dist**. Elle se charge enfin de construire, dans **/dist**, les fichiers **zip** de l'application qui va ainsi pouvoir être mise à la disposition des utilisateurs.

esup-commons prévoit les fichiers à intégrer aux fichiers *zip* qui seront distribués. Mais il est possible de personnaliser ce comportement pour, par exemple :

- Intégrer plus de fichiers de configuration que ne le fait *esup-commons* par défaut.
- Ajouter des fichiers spécifiques à votre application.

Pour cela il faut modifier, dans le fichier **buil-devel.xml**, les propriétés :

- **${app.zip-files.shared.include}** qui sont les fichiers à inclure dans le *zip* de la version *servlet*/*portlet* de votre application et le *zip* de la version *quick-start* de votre application.
- **${app.zip-files.include}** qui sont les fichiers, en plus de **${app.zip-files.shared.include}**, à inclure spécifiquement dans le *zip* de la version *servlet*/*portlet* de votre application
- **${app.zip-files.quick-start.include}** qui sont les fichiers, en plus de **${app.zip-files.shared.include}**, à inclure spécifiquement dans le *zip* de la version *quick-start* de votre application
- **${app.zip-files.exclude}** qui sont les fichiers à exclure de vos distributions.
- **${commons.zip-files.exclude}** qui sont les fichiers de la hiérarchie esup-commons qu'il ne faut pas mettre dans vos distributions.

Exercice 37 : Créer une distribution

Lancer la tâche **_dist** et observer les fichiers créés.

Note : définir la propriété **app.release** dans le fichier **build-devel.properties**.

16. Gestion de la documentation

La documentation peut être maintenue au sein même des sources de l'application, dans le répertoire **/docs**, sous un des deux formats suivants :

- Au format *HTML* ; le développeur utilise dans ce cas l'éditeur *HTML* de son choix (DreamWeaver, …) pour maintenir la documentation.
- Au format *XML* (*docbook*).

Le mode de génération de la documentation est contrôlé par la propriété **${docs.format}**, définie dans le fichier **build-devel.properties** ou bien directement dans le fichier **build-devel.xml**.

- Si elle vaut **html**, alors *esup-commons* considère que la documentation est au format *HTML* ;
- Si elle vaut **docbook**, alors *esup-commons* considère que la documentation est au format *docbook* ;
- Si la propriété est définie sans valeur (vide), alors seule la documentation des sources est générée ;
- Si la propriété n'est pas définie, aucune documentation n'est générée.

La tâche *ant* **_doc**, elle-même appelée par la tâche **_dist**, génère la documentation du projet, dans le répertoire **/website** de votre application.

16.1. Ajout de fichiers de configuration d'exemple à la documentation

Dans les deux premiers cas cités précédemment, il est possible de demander à *esup-commons* de copier automatiquement certains fichiers d'exemple dans le répertoire **/docs/examples**. Il suffit pour cela de spécifier le nom de ces fichiers dans les propriétés **${docs.examples.commons}** et **${docs.examples.app}** (les fichiers sont alors trouvés dans les projets *Eclipse* correspondants). On utilisera par exemple :

```
docs.examples.commons=examples=\
/properties/ldap/ldap-example.xml \
/properties/cache/cache-example.xml
docs.examples.app=examples=\
/properties/domain/domain-example.xml
```

16.2. Génération de la documentation des sources

La documentation des sources de votre application (ainsi que ceux de *esup-commons*) est générée par la tâche *ant* **_commons-javadoc**, appelée par la tâche **_doc** (et donc par la tâche **_dist**).

Les fichier *HTML* produits par *Javadoc* sont situés dans le répertoire **/docs/api**, ils sont ensuite recopiés dans le répertoire **/website**.

16.3. Documentation au format HTML

Comme indiqué plus haut, *esup-commons* génère la documentation à partir de fichiers *HTML* si la propriété **${docs.format}** est positionnée à la valeur **html**.

L'organisation des fichiers de documentation au format *HTML* est complètement libre et laissée à l'initiative du développeur.

Dans le cas de l'utilisation de *DreamWeaver*, les fichiers de modèle pourront être situés dans le répertoire **/docs/Templates**, répertoire qui n'est pas copié par les tâches *ant*.

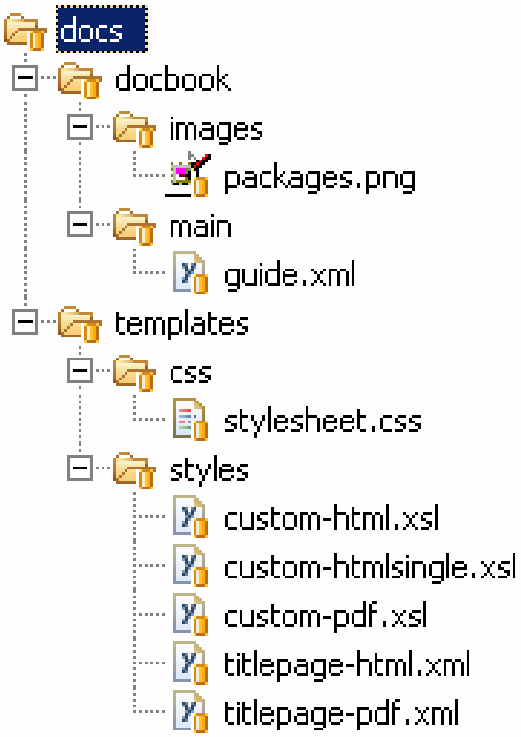
16.4. Documentation au format docbook

Comme indiqué plus haut, *esup-commons* génère la documentation à partir de fichiers *XML* si la propriété **`${docs.format}`** est positionnée à la valeur **`docbook`**.

Gérer la documentation au format *docbook* permet au développeur de générer ensuite la documentation automatiquement aux formats *HTML* et *PDF* :

- Si la propriété **`${docs.docbook.htmlsingle}`** est égale à **`true`**, il sera généré un fichier *HTML* par fichier *XML* source.
- Si la propriété **`${docs.docbook.html}`** est égale à **`true`**, il sera généré un fichier *HTML* par chapitre.
- Si la propriété **`${docs.docbook.pdf}`** est égale à **`true`**, il sera généré un fichier *PDF* par fichier *XML* source.

L'organisation des fichiers de documentation au format *docbook* est imposée par esup-commons, car les tâches *ant* de génération de la documentation s'attendent à trouver une structure précise pour la génération de la documentation finale :



Fichiers sources

Les sources *XML* sont trouvés par *esup-commons* dans un ou plusieurs répertoires que vous nommerez comme vous le souhaitez, dans **`/docs/docbook`** (mais les fichiers ne doivent se trouver directement dans le répertoire **`/docs/docbook`**). Par défaut, vous trouverez un répertoire **`/docs/docbook/main dans esup-blank`** qui contient un exemple de document *XML*.

Images

Tous les médias pointés par votre document doivent être déposés dans le répertoire **`/docs/docbook/images`** présent à ce niveau.

Personnalisation

Le rendu *HTML* de la documentation peut être personnalisé en modifiant les fichiers CSS contenus du répertoire **`/docs/templates/css`**.

La transformation de *XML* à *HTML*/*PDF* peut être personnalisée en modifiant les feuilles *XSL* **`html-custom.xsl`**, **`htmlsingle-custom.xsl`** et **`pdf-custom.xsl`**, tous situés dans le répertoire **`/docs/templates/styles`**.

Options

La taille de sortie des documents *PDF* peut être imposée par la propriété **`${docs.docbook.pdf.page}`** ; les valeurs acceptées sont :

- USletter, 4A0, 2A0,
- A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
- B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,
- C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10

17. Configuration d'une application à l'aide de fichiers de propriétés

Pour simplifier la tâche des exploitants, il est possible de configurer une application en leur demandant d'éditer des fichiers de propriétés, plus simples que des fichiers XML de configuration *Spring*.

Pour cela, il suffit de déclarer un *bean* supplémentaire et d'écrire le fichier de propriétés.

On utilisera par exemple, en tête du fichier **`/properties/applicationContext.xml`** :

```
<bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property
      name="location"
      value="classpath:/properties/config.properties" />
</bean>
```

Le fichier de propriétés **`/properties/config.properties`** ressemblera par exemple à :

```
ldap.host=ldap.esup-portail.org
ldap.port=392
```

Les *beans Spring* peuvent ensuite faire référence à ces propriétés de la manière suivante :

```
<bean
    id="contextSource"
    class="org.springframework.ldap.support.LdapContextSource">
    ...
  <property
      name="url"
      value="ldap://${ldap.host}:${ldap.port}">
    ...
</bean>
```

18. Commandes *batch*

La problématique des commandes *batch* est la gestion des sessions *Hibernate* et des exceptions.

Les accès batch à la base de données se font à l'aide de tâches *ant*, on utilise dans ce cas le modèle *one-session-per-command* (une session par commande).

La méthode **`main()`** de la classe appelée par la tâche *ant* appelle la méthode **`dispatch()`** et attrape les exceptions :

```
public static void main(final String[] args) {
  try {
    ApplicationService applicationService
        = ApplicationUtils.createApplicationService();
    LOG.info(applicationService.getName() + " v"
        + applicationService.getVersion());
    dispatch(args);
  } catch (Exception e) {
    ExceptionUtils.catchException(e);
  }
}
```

Un exemple de méthode **`dispatch()`**, qui appelle **`test()`** :

```
protected static void dispatch(final String[] args) {
  switch (args.length) {
    case 0:
      syntax();
      break;
    case 1:
      if ("test".equals(args[0])) {
        test();
      } else {
        syntax();
      }
      break;
    default:
      syntax();
      break;
  }
}
```

Enfin la méthode **`test()`** proprement dite, qui fait tous ses accès à la base de données à travers la façade des services métier :

```
private static void test() {
  try {
    DatabaseUtils.open();  // ouverture session
    DatabaseUtils.begin(); // ouverture transaction
    DomainService domainService =
      (DomainService) BeanUtils.getBean("domainService");
    ...
    DatabaseUtils.commit(); // commit
    DatabaseUtils.close(); // fermeture session
  } catch (RuntimeException e) {
    DatabaseUtils.rollback(); // optionnel
    DatabaseUtils.close();    // rollback si commit non appelé
    throw e;
  }
}
```

Note : les opérations *commit* et *rollback* peuvent être appelées plusieurs fois sur la même session, selon besoin pour une plus fine granularité.

Exercice 38 : Écrire un programme *batch* appelé par une tâche *ant*

Ajouter une méthode **`testAddEntry()`** à la classe existante **`Batch`** qui ajoute une entrée avec la valeur **`BATCH`** dans la base de données, écrire une tâche *ant* **`test-addEntry`** (dans **`build.xml`**), l'exécuter et contrôler la présence de la nouvelle entrée dans la base de données.

19. Ajout de *web services*

Pour dialoguer avec une application à distance, *esup-commons* propose de s'appuyer sur les *web services*, pour des raisons évidentes de standardisation et d'interopérabilité.

Afin de faciliter le travail du développeur, la bibliothèque *xFire* a été choisie, elle rend l'exposition de services extrêmement simple.

Ce chapitre montre comment exposer des services via *xFire*, puis montre comment on peut accéder ce service depuis une application externe.

19.1. Écrire le service à exposer

Interface

La première étape consiste à écrire l'interface du service que l'on veut exposer, par exemple :

```
public interface Information {
  String getVersion();
}
```

Exercice 39 : Écrire l'interface d'un *web service*

Écrire une interface Information qui possède une méthode **`int getEntriesNumber()`** qui retourne le nombre d'objets de type **`Entry`** dans la base de données.

Implémentation

Il faut ensuite en écrire une implémentation, par exemple :

```
public class InformationImpl implements Information, InitializingBean
{
  private ApplicationService applicationService;
  public InformationImpl() {
    super();
  }
  public void afterPropertiesSet() {
    Assert.notNull(applicationService,
       "property applicationService of class "
       + this.getClass().getName() + " can not be null");
  }
  public String getVersion() {
    return applicationService.getVersion().toString();
  }
}
```

Exercice 40 : Écrire l'implémentation d'un *web service*

Ajouter une méthode **`getEntriesNumber()`** au service de données (interface et implémentation). Remonter cette méthode au niveau du service métier. Faire une implémentation de l'interface **`Information`** (**`InformationImpl`**) en s'appuyant sur le service métier.

19.2. Exposer le service

Au niveau de Spring

Il faut déclarer les *beans Spring* qui exposent le *web service* ; par convention, ces *beans* sont déclarés dans le fichier **`/properties/export/export.xml`**.

Le *bean* abstrait **abstractXFire** est introduit pour factoriser le code dans le cas où plusieurs services sont exposés.

```
<bean
    id="abstractXFire"
    class="org.codehaus.xfire.spring.remoting.XFireExporter"
    abstract="true" >
  <property name="serviceFactory" ref="xfire.serviceFactory" />
  <property name="xfire" ref="xfire" />
</bean>
```

On déclare ensuite le *bean* qui sera instancié à l'appel du *web service* :

```
<bean
    id="informationXFire"
    parent="abstractXFire" >
  <property
      name="serviceBean"
      ref="information" />
  <property
      name="serviceClass"
      value="org.esupportail.example.services.remote.Information"
  />
</bean>
```

On voit que le *bean* **informationXfire** est une façade pour le service réel, qui se nomme **information** :

```
<bean
    id="information"
    class="org.esupportail.example.services.remote.InformationImpl" >
  <property
      name="applicationService"
      ref="applicationService" />
</bean>
```

On importe enfin les les ressources *Spring* nécessaires à *xFire*, depuis le *jar* de *xFire* (présent dans le *classpath*) :

```
<import resource="classpath:org/codehaus/xfire/spring/xfire.xml" />
```

Le *WSDL* correspondant au web service ci-dessous est accessible à l'*URL* **http://localhost:8080/esup-application/xfire/Information?wsdl** (adapter à la configuration locale). Notons qu'il ne s'agit ci que d'un simple contrôle, la suite montre qu'il n'est pas nécessaire d'accéder au *WSDL* en fonctionnement normal.

On doit enfin importer le fichier **/properties/export/export.xml** depuis le fichier de configuration *Spring* **/properties/applicationContext.xml** :

```
<import resource="export/export.xml" />
```

xFire fait l'association entre l'URL d'appel du *web service* et le *bean* à instancier en se basant sur le nom de l'interface exposée (**Information**).

Au niveau de Tomcat

Il faut ensuite exposer le service, à l'aide de *xFire*.

Pour cela, on configure le contexte *Tomcat* en déclarant une *servlet* pour *xFire* (dans **web.xml**) :

```
<servlet>
  <servlet-name>xfire</servlet-name>
  <servlet-class>
    org.esupportail.commons.web.servlet.XFireServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>xfire</servlet-name>
  <url-pattern>/xfire/*</url-pattern>
</servlet-mapping>
```

Cette simple déclaration masque la difficulté technique à intégrer *xFire* dans esup-commons. En utilisant la *servlet* **XFireServlet** fournie par esup-commons, on obtient sans plus de développement la gestion des exceptions et l'ouverture/fermeture de session avec la base de données, exactement comme pour les accès web standards et les commandes *batch*.

Une fois l'application relancée, on peut déjà tester l'installation de *xFire* en testant l'*URL* **http://localhost:8080/esup-application/xfire** (ou celle correspondant à l'installation courante).

Exercice 41 : Exposer un *web service*

Exposer le service **InformationImpl** et tester la récupération du *WSDL*.

19.3. Accéder au service exposé

L'application *esup-example* est à la fois serveur et client du point de vue des *web services*. Une autre application nommée *esup-example-client* est disponible sur le dépôt *SVN* du projet, qui montre comment une application externe accède à un service exposé.

Copie de l'interface du service

La première chose à faire est de recopier l'interface du service exposé dans les sources l'application cliente (ici l'interface **Information**).

Déclaration du bean client

Il suffit ensuite d'utiliser les fonctionnalités offertes par *Spring* pour accéder aux *web services* ; on déclarera par exemple le *bean* **remoteInformation** suivant dans le fichier **/properties/client.xml** (pour ne pas interférer avec la partie serveur) :

```
<bean
    id="remoteInformation"
    lazy-init="true"
    class="org.codehaus.xfire.spring.remoting.XFireClientFactoryBean">
  <property
     name="serviceClass"
     value="org.esupportail.example.services.remote.Information" />
  <property
     name="wsdlDocumentUrl"
     value="http://localhost:8080/esup-example/xfire/Information?WSDL"
     />
</bean>
```

Utilisation du bean client

Le *bean* **remoteInformation** est alors utilisable exactement comme s'il s'agissait d'un service local. Si le *bean* est défini au niveau de l'application cliente dans le fichier **/properties/client.xml**, on utilisera par exemple le code qui suit pour accéder au *service web*.

On commence par récupérer le fichier principal de configuration de *Spring* :

```
private static final String SPRING_CONFIG_FILE =
      "/properties/client.xml";

ClassPathResource res = new ClassPathResource(SPRING_CONFIG_FILE);
```

On construit ensuite l'usine de *bean* pour ce fichier de configuration :

```
BeanFactory beanFactory = new XmlBeanFactory(res);
```

On récupère ensuite le *bean* **remoteInformation** déclaré :

```
private static final String BEAN_NAME = "remoteInformation";

Information remoteService =
      (Information) beanFactory.getBean(BEAN_NAME);
```

On appelle ensuite simplement la méthode désirée du *bean* :

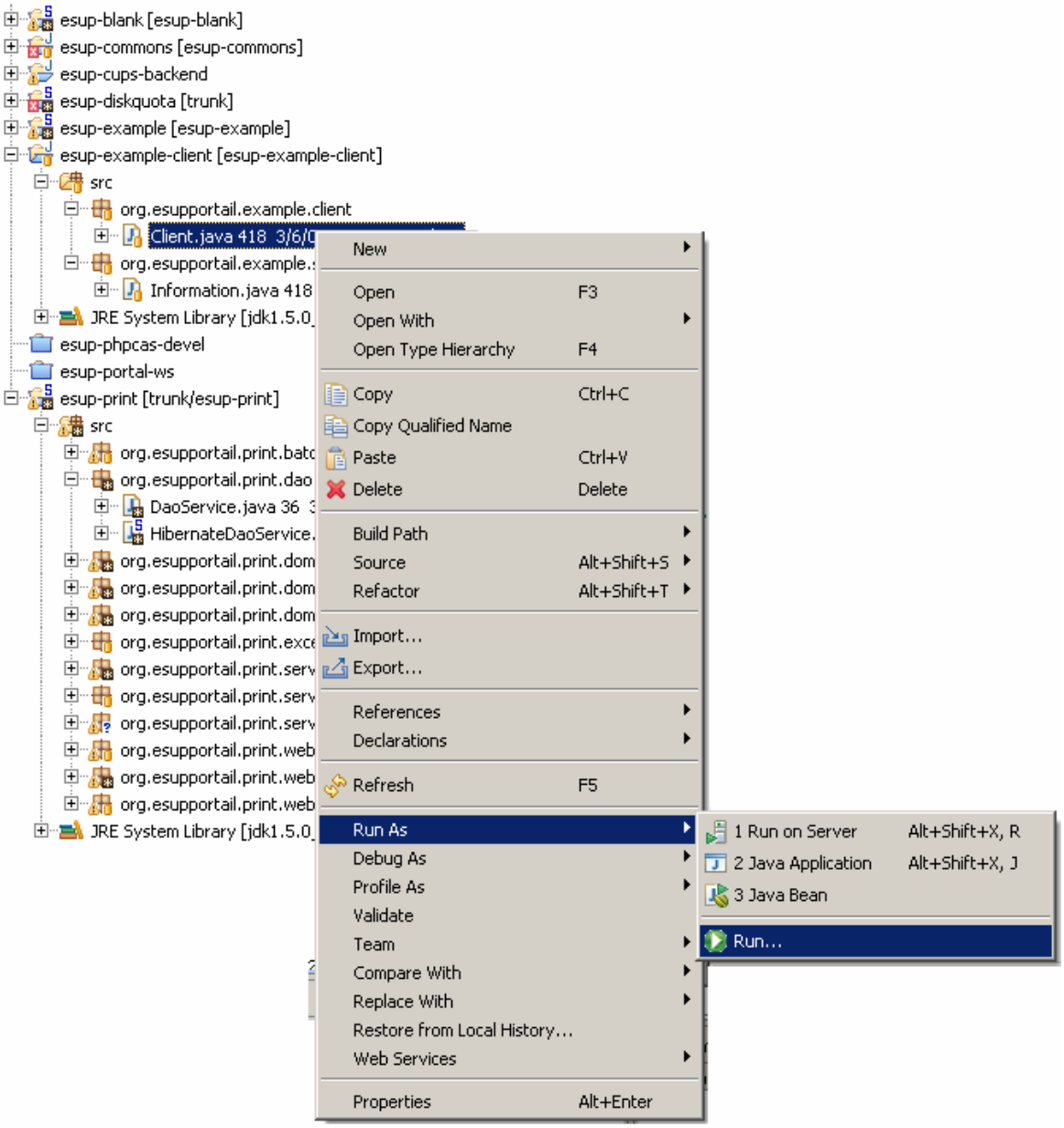
```
String version = remoteService.getVersion();
```

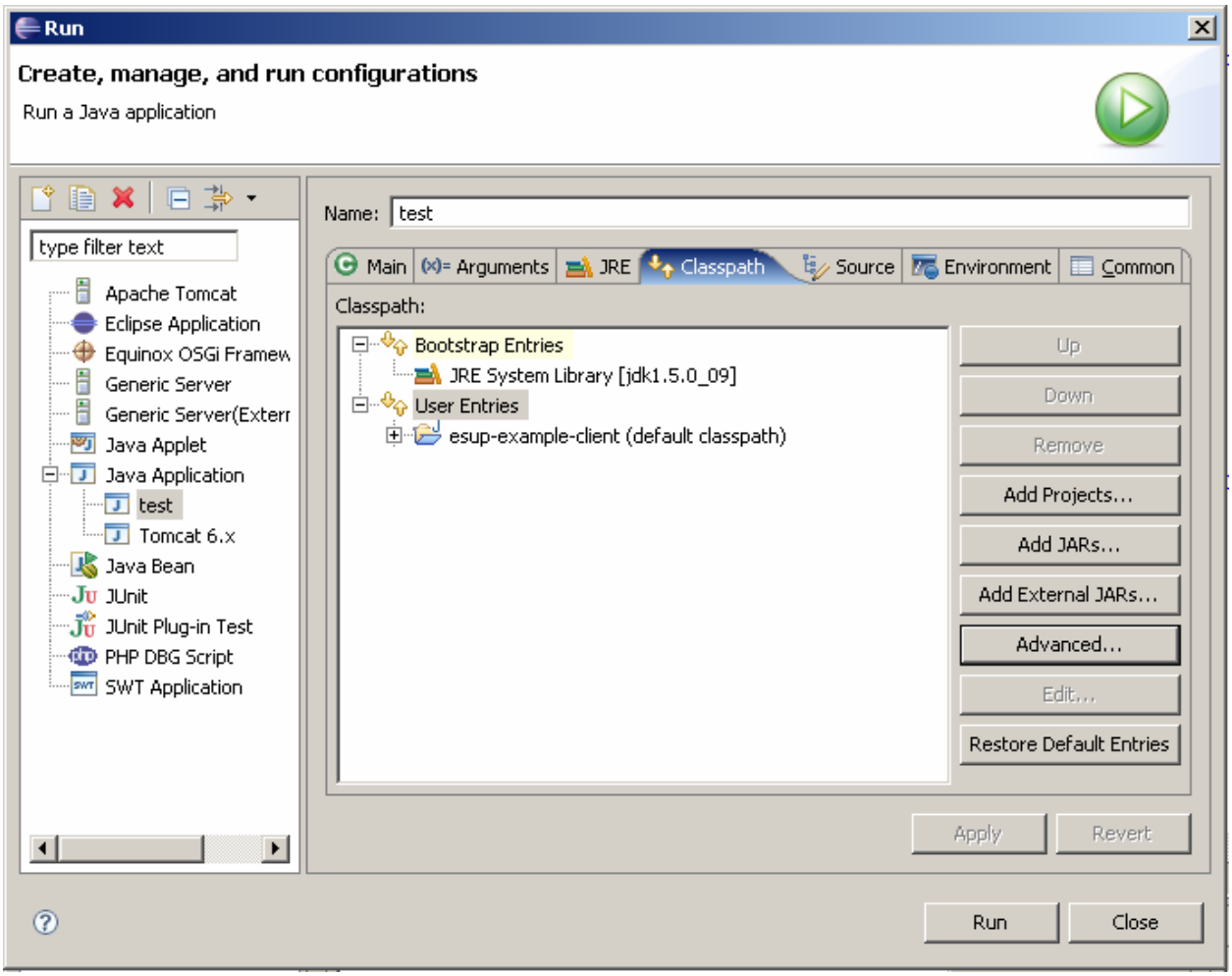
Pour tester le WS écrit précédemment, on se basera sur le projet *esup-example-client* disponible sur le dépôt *Subversion* à l'adresse : **http://subversion.cru.fr/esup-commons/trunk/esup-example-client**.

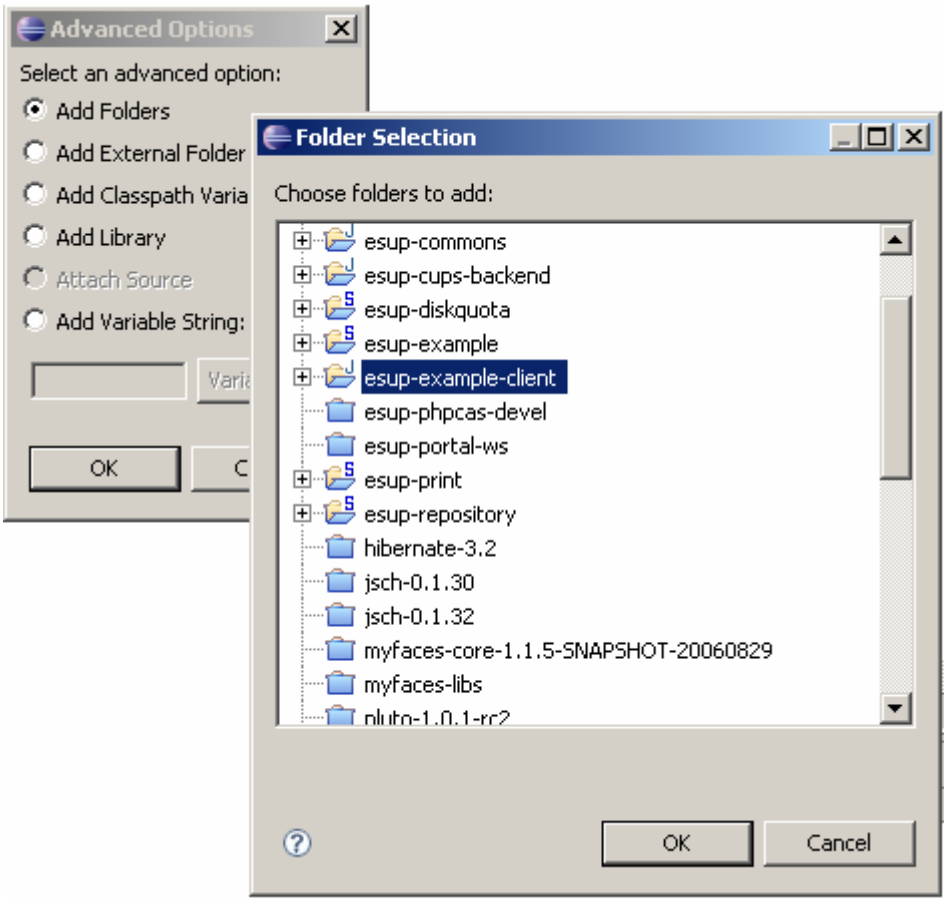
Exercice 42 : Créer le projet *Eclipse esup-example-client*

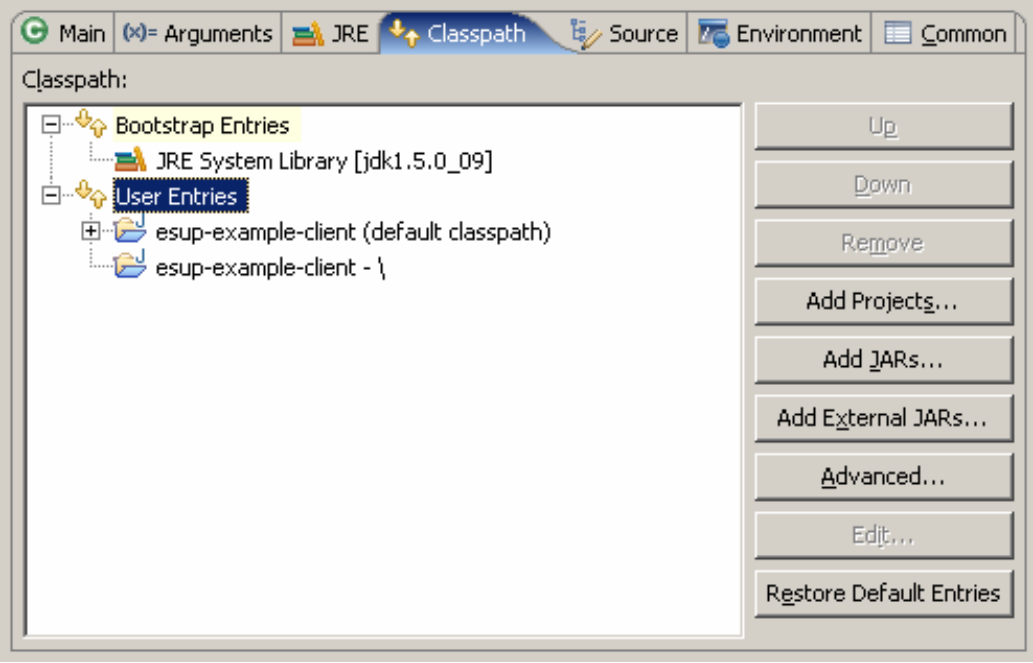
Rapatrier le projet *esup-example-client* depuis le dépôt *Subversion* comme nouveau projet *Eclipse*, le détacher du dépôt *SubVersion*, ajuster le répertoire source et le *classpath* (ajouter toutes les librairies du répertoire **/lib**).

Il faut, pour pouvoir exécuter le client, ajouter le répertoire racine du projet *Eclipse* dans le *classpath* de l'exécution de la classe **Client** pour accéder au fichier **/properties/applicationContext.xml** (les saisies d'écran ci-après montrent comment faire).


esup-blank [esup-blank]
esup-commons [esup-commons]
esup-cups-backend
esup-diskquota [trunk]
esup-example [esup-example]
esup-example-client [esup-example-client]
src
org.esupportail.example.client
Client.java 418 3/6/0
org.esupportail.example.
Information.java 418
JRE System Library [jdk1.5.0_
esup-phpcas-devel
esup-portal-ws
esup-print [trunk/esup-print]
src
org.esupportail.print.batc
org.esupportail.print.dao
DaoService.java 36 3
HibernateDaoService.
org.esupportail.print.dom
org.esupportail.print.dom
org.esupportail.print.dom
org.esupportail.print.exce
org.esupportail.print.serv
org.esupportail.print.serv
org.esupportail.print.serv
org.esupportail.print.web
org.esupportail.print.web
org.esupportail.print.web
JRE System Library [jdk1.5.0_
New
Open F3
Open With
Open Type Hierarchy F4
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
References
Declarations
Refresh F5
Run As
Debug As
Profile As
Validate
Team
Compare With
Replace With
Restore from Local History...
Web Services
Properties Alt+Enter
1 Run on Server Alt+Shift+X, R
2 Java Application Alt+Shift+X, J
3 Java Bean
Run...







Exercice 43 : Tester un *web service* comme client

Modifier le projet *esup-example-client* pour qu'il appelle le *web service* écrit précédemment.

Utilisation du client dans le même contexte Tomcat que le serveur

Notons finalement que, si le serveur et le client tournent dans le même contexte *Tomcat* (c'est le cas dans l'application *esup-example* lorsqu'elle est à la fois serveur et client de son *web service*), il faut nécessairement positionner la propriété **`lookupServiceOnStartup`** du *bean* **`remoteInformation`** à **`false`**.

20. Gestion des liens hypertextes (directs)

On appelle « lien hypertexte direct » une *URL* qui, entrée dans le navigateur de l'utilisateur, positionne l'application dans un état donné. Ces liens peuvent être communiqués aux utilisateurs (par courrier électronique, ou simplement affichés sur une page de l'application) et stockés par eux (en signet) pour être réutilisés à n'importe quel moment.

Une illustration de cette fonctionnalité est la remontée d'alertes de *esup-helpdesk* ; dans les courriers électroniques, on communique aux utilisateurs une URL qui amène directement sur la page du ticket correspondant, sans que l'utilisateur ait à naviguer vers ce ticket.

20.1. Générer des URLs (directes) vers l'application

La génération des liens directs s'appuie sur le *bean* **`urlGenerator`**, déclaré dans le fichier de configuration **`/properties/urlGeneration/urlGeneration.xml`**.

Ce *bean* doit implémenter l'interface **`UrlGenerator`**, qui possède les méthodes suivantes :

```
String url(final Map<String, String> params);
String url();
String urlViaCas(final Map<String, String> params);
String urlViaCas();
```

Comme on le voit, il existe des méthodes pour générer des *URLs* vers l'application directement (avec ou sans paramètres) et d'autres en passant par un serveur CAS.

Les *URLs* construites seront différentes selon que l'on est en mode *portlet* ou *servlet*, une implémentation est prévue pour chacun des cas.

Configuration en mode portlet

Il faut en mode *portlet* spécifier :

- L'*URL* du serveur *CAS*,
- L'*URL* de l'application.

Le fichier **`urlGeneration.xml`** ressemblera à :

```
<bean
    id="urlGenerator"
    class="[...].services.urlGeneration.UportalUrlGeneratorImpl"
    >
  <property
      name="casLoginUrl"
      value="https://cas.domain.edu/login?service=%s" />
  <property
      name="uportalFunctionnalName"
      value="esup-application" >
  <property
      name="uportalUrl"
      value="http://uportal.domain.edu/Login" />
</bean>
```

L'*URL* du serveur *CAS* (**`casLoginUrl`**), facultative, est utilisée par les méthodes **`urlViaCas()`** ; dans le cas où elle est omise (quand il n'y a pas d'authentification *CAS*), les méthodes **`urlViaCas()`** lancent une exception.

Le paramètre **`uportalFunctionnalName`** est le nom sous lequel le canal de la *portlet* a été publié dans *uPortal*.

Configuration en mode servlet

Il faut en mode *servlet* spécifier :
- L'*URL* du serveur *CAS*,
- L'*URL* de l'application.

Le fichier **urlGeneration.xml** ressemblera à :

```
<bean
    id="urlGenerator"
    class="[...].services.urlGeneration.ServletUrlGeneratorImpl"
    >
  <property
      name="casLoginUrl"
      value="https://cas.domain.edu/login?service=%s" />
  <property
      name="servletUrl"
      value="http://application.domain.edu:port/path" >
</bean>
```

L'*URL* du serveur *CAS* (**casLoginUrl**), facultative, est utilisée par les méthodes **urlViaCas()** ; dans le cas où elle est omise (quand il n'y a pas d'authentification *CAS*), les méthodes **urlViaCas()** lancent une exception. Le motif **%s** est remplacé à l'exécution par l'*URL* de l'application, pour le retour du navigateur après l'authentification.

Exemple

Supposons que l'on veuille générer une *URL* directe vers l'application qui l'amène directement sur une page montrant les caractéristiques d'un utilisateur. Nous prenons les paramètres **page** (auquel nous donnerons la valeur **showUser**) et **userId** (auquel nous donnerons comme valeur l'identifiant de l'utilisateur dont on veut afficher les caractéristiques).

La génération d'une telle *URL* directe via *CAS* dans un contrôleur se fait de la manière suivante (on suppose que le contrôleur a accès au générateur d'*URL*, via *Spring*) :

```
Map<String, String> params = new HashMap<String, String>();
params.put("page", "showUser");
params.put("userId", id);
String urlViaCas = getUrlGenerator().urlViaCas(params);
```

Exercice 44 : Générer un lien direct

Afficher sur la vue **test1.jsp** un lien direct vers l'application avec un paramètre **param** égal à l'attribut **myInput**.

On pourra pour afficher l'URL sous forme d'un lien direct déclarer une clé de bundle :

```
URL.TEXT = <a href="{0}">{0}</a>
```

Et utiliser cette clé de cette manière dans test1.jsp :

```
<e:text value="#{msgs['URL.TEXT']}" escape="false" >
    <f:param value="#{test1Controller.url}" />
</e:text>
```

20.2. Déchiffrer les URLs directes pour positionner l'application dans un état donné

Du point de vue du programmeur, l'appréhension d'un lien direct consiste à :

1. Décoder les paramètres de l'*URL*,
2. Positionner les variables d'état (les contrôleurs de l'application) dans un état donné,
3. Envoyer sur la page *JSF* voulue.

Comment ça marche (partie visible)

La difficulté technique de l'opération tient au fait que, dans l'esprit d'*esup-commons*, le mécanisme doit fonctionner à la fois en mode *servlet* et en mode *portlet*. Pour faciliter le travail du programmeur, la difficulté est abstraite et consiste simplement à écrire un redirecteur.

Le redirecteur est un *bean* nommé **`deepLinkingRedirector`** déclaré dans le fichier de configuration **`/properties/deepLinking/deepLinking.xml`**. Il implémente l'interface **`DeepLinkingRedirector`**, qui possède la méthode :

```
String redirect(final Map<String, String> params);
```

Cette méthode doit retourner la vue vers laquelle sera redirigée l'application en fonction des paramètres passés (qui sont déjà décodés par *esup-commons*), ou **`null`** pour la page d'accueil.

Si l'on ne souhaite pas utiliser ce mécanisme, il suffit d'utiliser le redirecteur par défaut, qui retourne toujours **`null`** :

```
<bean
    id="deepLinkingRedirector"
    class="[...].web.deepLinking.VoidDeepLinkingRedirectorImpl"
    scope="session"
/>
```

Si on souhaite utiliser des liens directs, il suffit alors d'écrire un redirecteur. Un exemple est donné plus loin.

Comment ça marche en mode servlet

Le nom du redirecteur par défaut utilisé par la classe **`FacesServlet`** est **`deepLinkingRedirector`**.

A chaque requête, la *servlet* appelle le redirecteur et s'il retourne une valeur différente de **`null`**, dirige la requête vers la vue voulue (les contrôleurs ont été mis à jour par le redirecteur).

Note : on peut le changer en ajoutant un paramètre à la déclaration de la *servlet* dans **`web.xml`** (ça ne sert a priori à rien mais c'est possible) :

```
  <init-param>
    <param-name>deep-linking-redirector</param-name>
    <param-value>myRedirectorBean</param-value>
  </init-param>
```

Comment ça marche en mode portlet

Dans **`portlet.xml`**, on passe à la *portlet* **`FacesPortlet`** le paramètre suivant :

```
<init-param>
  <name>default-view-selector</name>
  <value>[...].web.deepLinking.UportalDefaultViewSelector</value>
</init-param>
```

Ce sélecteur de vue s'appuie ensuite sur le *bean* **`deepLinkingRedirector`** pour déterminer la vue à afficher, en prenant en compte d'éventuels paramètres passés dans l'URL.

Exemple

Supposons que l'on veuille décoder l'*URL* directe construite précédemment. Nous montrons ici quel serait le corps de la méthode **`redirect()`**.

Les opérations à effectuer sont :

- Analyser les paramètres pour savoir s'il faut faire une redirection,
- Exécuter du code métier (ici trouver l'utilisateur qui correspond aux paramètres),
- Positionner le contrôleur de la vue cible,
- Rediriger vers cette vue.

Analyse des paramètres

```
String page = params.get("page");
if (page == null) {
  return null;
}
if (!params.get("page").equals("showUser")) {
  addErrorMessageInvalidParameter("page", params.get("page"));
  return null;
}

String userId = params.get("userId");
if (userId == null) {
  addErrorMessageMissingParameter("userId");
  return null;
}
```

Code métier

```
User user = domainService.getUser(userId);
if (user == null) {
  addErrorMessage(
      null,
      "DEEP_LINKS.MESSAGE.USER_NOT_FOUND",
      userId);
  return null;
}
```

Positionnement du contrôleur

```
usersControler.setCurrentUser(user);
```

Redirection vers la vue cible

```
return "/stylesheets/showUser.jsp";
```

Exercice 45 : Écrire un redirecteur de liens directs

Écrire une vue **`test3.jsp`** qui affiche l'attribut **`value`** d'un contrôleur **`test3Controller`** (de classe **`Test3Controller`**). Écrire un redirecteur qui, lorsque le paramètre **`param`** est présent, assigne la valeur du paramètre à l'attribut **`value`** du contrôleur **`test3Controller`**, puis redirige l'utilisateur vers la vue **`test3.jsp`**.

Note : on partira avantageusement de l'implémentation fournie dans le projet *esup-example*.

21. Authentification

esup-commons n'authentifie pas les utilisateurs en tant que tel. Il est en revanche capable, en s'appuyant sur un service d'authentification externe, de savoir quel est l'utilisateur connecté à l'application web en cours.

Plusieurs moyens sont possibles pour cela. *esup-commons* s'appuie sur un service d'authentification externe, le *bean* **`authenticationService`**, pour authentifier les utilisateurs. Ce *bean*, qui doit implémenter l'interface **`AuthenticationService`**, possède une méthode **`getCurrentUserId()`**, qui renvoie l'identifiant de l'utilisateur courant.

Le *bean* **`authenticationService`** est défini dans **`/properties/auth/auth.xml`**. Plusieurs classes implémentant l'interface **`AuthenticationService`** sont disponibles dans le package **`org.esupportail.commons.services.authentication`**, elles sont décrites ci après.

21.1. Modes disponibles

CAS

La classe **`CasFilterAuthenticator`**, renvoie l'identifiant de l'utilisateur authentifié par le filtre J2EE CAS :

```
<bean
    id="authenticationService"
    class="[...].authentication.CasFilterAuthenticator" />
```

Pour que cela fonctionne, il faut bien sûr que le filtre *CAS* soit correctement configuré.

Portail (JSR-168)

La classe **`PortalAuthenticator`** renvoie l'identifiant de l'utilisateur connecté au portail, selon *JSR*-168 :

```
<bean
    id="authenticationService"
    class="[...].authentication.PortalAuthenticator" >
  <property name="uidPortalAttribute" value="uid" />
</bean>
```

La propriété **`uidPortalAttribute`** est le nom de l'attribut du portail qui sera considéré comme l'identifiant de l'utilisateur. Il doit être déclaré dans le fichier **`/webapp/WEB-INF/portlet.xml`**.

Cette implémentation ne fonctionne qu'en mode *portlet*.

Mixte CAS / Portail

La classe **`PortalOrCasFilterAuthenticator`** s'appuie d'abord sur *JSR*-168, puis sur le filtre *CAS* (fonctionne à la fois en mode *portlet* ou *servlet*) :

```
<bean
    id="authenticationService"
    class="[...].authentication.PortalOrCasFilterAuthenticator" >
  <property name="uidPortalAttribute" value="uid" />
</bean>
```

Cette implémentation fonctionne à la fois en mode *portlet* et en mode *servlet*, il n'est donc pas nécessaire de la changer lorsque l'on change de mode de déploiement.

Remote user

La classe **RemoteUserAuthenticator** s'appuie sur la variable **REMOTE_USER** :

```
<bean
    id="authenticationService"
    class="[...].authentication.RemoteUserAuthenticator" />
```

L'utilisation de cette classe n'est pas recommandée.

Authentification hors connexion

La classe **OfflineFixedUserAuthenticator** renvoie toujours le même identifiant.

```
<bean
    id="authenticationService"
    class="[...].authentication.PortalOrCasFilterAuthenticator" >
  <property name="userId" value="paubry" />
</bean>
```

Elle peut être utilisée lorsque l'on travaille à la mise au point hors connexion (pratique dans le train).

21.2. Authentification en mode batch

En mode batch, la définition du *bean* **authenticationService** n'est pas obligatoire.

21.3. Utilisation du service d'authentification

Le service d'authentification s'appelle de la manière suivante :

```
String uid = authenticationService.getCurrentUserId();
```

Si **uid** est **null**, alors l'utilisateur n'est pas authentifié.

22. Déploiement en *servlet*

Le déploiement en *quick-start* utilisé jusque là est en fait un déploiement *servlet*, pour lequel esup-commons installe un *Tomcat* et le configure à la place de l'utilisateur. Ce déploiement est très intéressant, en particulier pour les utilisateurs ne connaissant pas l'administration de *Tomcat*.

Néanmoins, les exploitant maîtrisant *Tomcat* préfèrent en général s'appuyer sur leur existant ; Cela peut leur permettre notamment de faire tourner plusieurs applications sur un même *Tomcat*.

Nous donnons dans cette partie les indications pour déployer les applications bâties sur *esup-commons* en *servlet*.

22.1. Les fichiers de configuration

`build.properties`

Pour déployer une application sous forme de *servlet* (dans un conteneur existant), il faut indiquer :

- L'endroit où sont déployées les *servlet*s avec la propriété **`${deploy.home}`**, par exemple **`/usr/local/servlets`**.

Un **`build.properties`** minimal sera de la forme suivante :

```
deploy.type=servlet
deploy.home=/usr/local/servlets/esup-application
```

Note : le déploiement de l'application est toujours précédé d'une compilation, qui produit le répertoire **`/build`**. Ces fichiers sont ensuite copiés dans **`${deploy.home}`**.

`server.xml`

Le fichier **`server.xml`** de *Tomcat* doit indiquer le contexte de la *servlet* :

```
<Context
  path="/esup-application"
  docBase="/usr/local/servlets/esup-application"
/>
```

`web.xml`

```
<servlet>
  <display-name>Faces Servlet</display-name>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    org.esupportail.commons.web.servlet.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

L'extrait de **`web.xml`** ci-dessus ne mentionne que la déclaration de la *servlet* principale. Pour l'intégralité du fichier, se référer à **`web-servlet-example.xml`**.

22.2. La gestion des logs

Cf Annexe A.

22.3. Les feuilles de style (CSS)

Cf Annexe B.

23. Déploiement en *portlet*

Un point fort d'*esup-commons* est de pouvoir déployer les applications en *portlet* ou en *servlet*, tout en utilisant strictement le même code, et c'est dans cet objectif que *JSF* a été choisi par rapport à *Spring* car son MVC est indépendant du mode de déploiement.

Nous donnons dans cette partie les indications pour déployer les applications bâties sur *esup-commons* en *portlet*.

23.1. Les fichiers de configuration

build.properties

Dans la version actuelle de *esup-commons*, le déploiement en WAR n'est pas possible ; lors de l'appel de la tâche **deploy**, les fichiers produits dans le répertoire **${deploy.home}** sont prêts à être utilisés par *Pluto* (le fichier **web.xml** n'a pas besoin d'être transformé).

Pour déployer une application sous forme de *portlet* (dans un portail existant), il faut indiquer :

- L'endroit où sont déployées les *portlet*s avec la propriété **${deploy.home}**, par exemple **/usr/local/uPortal/portlets**.

Un **build.properties** minimal sera de la forme suivante :

```
deploy.type=portlet
deploy.home=/usr/local/uPortal/portlets/esup-application
```

Note : le déploiement de l'application est toujours précédé d'une compilation, qui produit le répertoire **/build**. Ces fichiers sont ensuite copiés dans **${deploy.home}**.

server.xml

Le fichier **server.xml** de *Tomcat* doit indiquer le contexte de la *portlet* :

```
<Context
  path=""
  docBase="/usr/local/webapps/uPortal"
/>
<Context
  path="/esup-application"
  docBase="/usr/local/uPortal/portlets/esup-application"
/>
```

web.xml

```
<servlet>
  <servlet-name>esup-application</servlet-name>
  <servlet-class>
    org.apache.pluto.core.PortletServlet
  </servlet-class>
  <init-param>
    <param-name>portlet-class</param-name>
    <param-value>
      org.esupportail.commons.web.portlet.FacesPortlet
    </param-value>
  </init-param>
  <init-param>
    <param-name>portlet-guid</param-name>
    <param-value>esup-application.esup-application</param-value>
  </init-param>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>esup-application</servlet-name>
  <url-pattern>/esup-application/*</url-pattern>
</servlet-mapping>
```

L'extrait de **web.xml** ci-dessus ne mentionne que la déclaration de la *servlet* implémentant la *portlet*. Pour l'intégralité du fichier, se référer à **web-portlet-example.xml**.

Le paramètre **portlet-guid** a une importance toute particulière. C'est un élément déterminant dans la communication entre le portail et la *portlet*. C'est lui qui sera utilisé dans la configuration du portail pour faire référence à la *portlet*. Il est de la forme **<Nom du context>.<Nom de la portlet>**. **<Nom du context>** est le nom du *context Tomcat* dans lequel la *portlet* est déployée. **<Nom de la portlet>** correspond à la propriété **portlet-name** du fichier **portlet.xml** vu ci-dessous.

portlet.xml

```
<portlet>
  <portlet-name>esup-application</portlet-name>
  <portlet-class>
    org.esupportail.commons.web.portlet.FacesPortlet
  </portlet-class>
  <init-param>
    <name>default-view</name>
    <value>/stylesheets/welcome.jsp</value>
  </init-param>
  <init-param>
    <name>default-view-selector</name>
    <value>
    org.esupportail.commons.web.deepLinking.UportalDefaultViewSelector
    </value>
  </init-param>
  <expiration-cache>-1</expiration-cache>
  <supports>
    <mime-type>text/html</mime-type>
  </supports>
  <portlet-info>
    <title>esup-example</title>
    <short-title>esup-application portlet</short-title>
    <keywords>esup application</keywords>
  </portlet-info>
</portlet>
<user-attribute>
  <description>the uid of the portal user</description>
  <name>uid</name>
</user-attribute>
```

L'écriture du fichier **portlet.xml** est très simple en utilisant le fichier d'exemple **portlet-example.xml**.

Un fichier **portlet.xml** peut contenir des balises **user-attribute**. Ces balises donnent la liste des attributs que le portail peut communiquer à la *portlet*.

Dans l'exemple ci-dessus, on retrouve une balise **user-attribute** avec **uid** comme attribut communiqué du portail à la *portlet*. Cet attribut doit être le même que celui utilisé pour la propriété **uidPortalAttribute** de la classe **PortalOrCasFilterAuthenticator** si c'est elle qui est utilisée comme *bean* **authenticationService** dans le fichier **/properties/auth.xml**.

Note : le fichier **portlet.xml** est légèrement différent lorsque l'on utilise le filtre *MyExtensions*, nécessaires pour l'utilisation des certains composants de la bibliothèque

Tomahawk. La partie « 24.2 Réception d'un fichier par le serveur » montre cette autre configuration en détail.

23.2. L'intégration dans uPortal

Une fois votre *portlet* déployée vous devez encore la faire connaître de *uPortal* pour pouvoir l'utiliser.

Note : Les premiers essais montrent qu'il ne faut pas déployer une *portlet* dans le répertoire **`appBase`** par défaut de *Tomcat* (typiquement **`webapps`**) mais définir un *context* spécifique hors de ce répertoire pour que la communication entre portail et *portlet* soit possible.

Il existe deux façons de faire connaître votre *portlet* de *uPortal*. Une consiste à utiliser le Channel Manager *uPortal*. L'autre consiste à utiliser la tâche ant **`pubchan`** de *uPortal*.

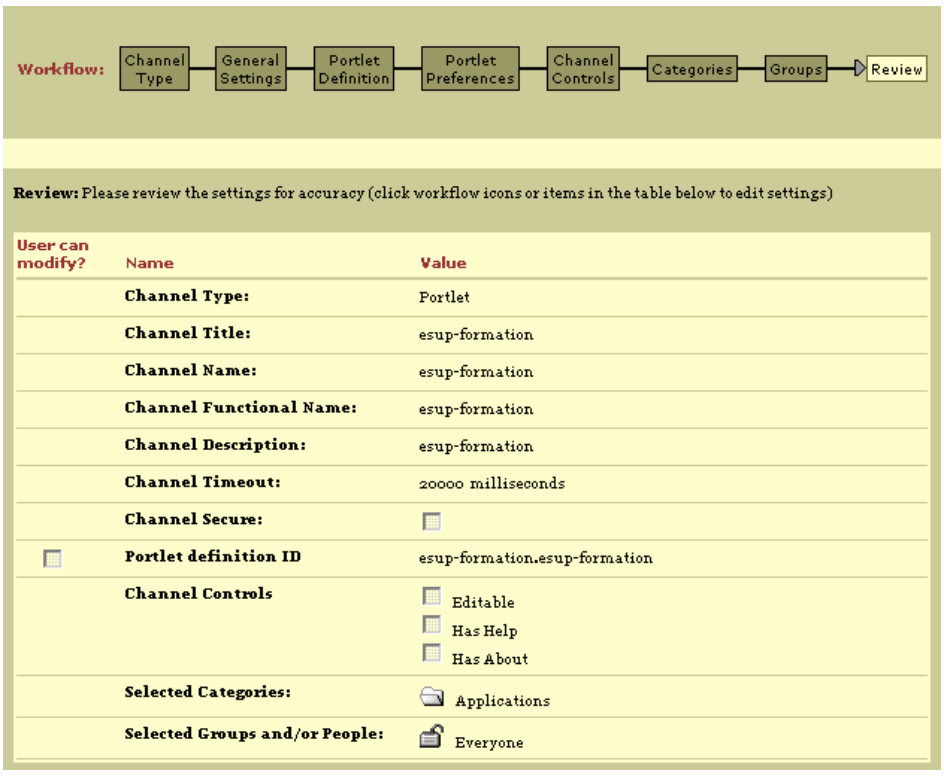
Note : pour tester votre *portlet* vous utilisez ensuite les préférences utilisateur du portail pour vous allouer la *portlet* dans votre environnement ou utiliserez le mécanisme des fragments pour pousser la *portlet* à un ensemble d'utilisateurs du portail. Le lecteur se reportera à la documentation *uPortal* à ce sujet.

Utilisation du Channel Manager uPortal

Se connecter avec un compte **`admin`** sur le portail, puis :

- Utiliser le bouton  pour lancer le *Channel Manager*.
- Utiliser ensuite le lien « *Publish a new channel* » puis choisir un canal de type « *portlet* ».
- Saisir les informations communes à tous les canaux comme pour tout autre canal intégré dans *uPortal*.
- Dans la partie « *Portlet Definition* » saisir comme « *Portlet definition ID* » le **`portlet-guid`** du fichier **`web.xml`** vu ci-dessus.

En fin de procédure vous devriez avoir un écran semblable à celui-ci :



Utilisation de la tâche pubchan de uPortal

uPortal propose un mécanisme permettant d'enregistrer un canal en utilisant un fichier de définition XML.

Pour cela, on doit utiliser la tâche *ant* **pubchan** qui est disponible avec *uPortal* et lui passer en paramètre le fichier de définition de la *portlet*.

Un fichier d'exemple est disponible dans le répertoire **/utils/uPortal** de *esup-blank* sous le nom **esup-formation-portlet-chanpub.xml**. Il convient de copier ce fichier dans le répertoire **/properties/chanpub** de votre *uPortal*.

Exemple de commande :

```
ant uportal.pubchan -Dchannel=esup-formation-portlet-chanpub.xml
```

Exercice 46 : Déployer une *portlet*

Pour déployer le portail suivre les instructions ci-dessous :

1. Décompresser l'archive du dernier **esupdev** fournie sur le site esup-portail.org

2. Modifier les entrées suivantes dans le fichier **esup.properties** :

```
java_home=D:/formation/JDK_1.5
esup.root=D:/formation/esupdev-2.5-esup-2
esup.keystore=D:/formation/utils/cas/cru-root.keystore
esup.public.host=localhost
esup.public.port=:8080
esup.ldap.host=ldapglobal.univ-rennes1.fr
esup.ldap.baseDN=ou=People,dc=univ-rennes1,dc=fr
esup.cas.auth=true
esup.cas.host=sso-cas.univ-rennes1.fr
esup.db.auth=true
esup.db.username=root
esup.db.url=jdbc:mysql://localhost/uportal
esup.db.jdbcDriverJar=mysql-connector-java-3.0.15-ga-bin.jar
esup.db.className=com.mysql.jdbc.Driver
esup.db.db-version=4.0.18-max-log
esup.webservices.axis=true
esup.webservices.axis.groups=true
```

3. Modifier **ant.bat**, par exemple :

```
SET JAVA_HOME=D:\formation\JDK_1.5
SET ESUPDEV=D:\formation\esupdev-2.5-esup-2
```

4) Définir un *context* pour la portlet. Pour cela, Copier **UpdateEsup/Tomcat/conf/server.xml** dans un **Custom/Tomcat/conf/server.xml** et modifier ce dernier pour y ajouter le contexte de l'application, par exemple :

```
<Context
    path="/esup-formation"
    docBase="D:/formation/esupdev-2.5-esup-2/uPortal-quick-
start/webapps/esup-formation"
/>
```

5. Lancer les tâches *ant* **esup.init**, **esup.deploy** et **esup.db.init** pour installer le portail

6. Se connecter au portail (**http://localhost:8080/uPortal**) en utilisant l'utilisateur **admin** et le mot de passe **admin** pour déclarer la *portlet* avec le *Channel Manager.*

23.3. La gestion des logs

Cf Annexe A.

23.4. Les feuilles de style (CSS)

Cf Annexe B.

24. Téléchargement de fichiers

24.1. Envoi d'un fichier au client

L'envoi d'un fichier au client est très simple dans une *servlet*, il nécessite en revanche quelques acrobaties techniques pour les *portlet*s, qui sont sensées ne produire des données que pour être affichées par le portail. La seule solution dans ce mode est d'envoyer les données à l'aide d'une *servlet* additionnelle. Un obstacle supplémentaire à l'implémentation vient du fait que l'on souhaite dans *esup-commons* que le code soit exactement le même que l'on déploie en *portlet* ou en *servlet*.

Comment ça marche (partie cachée)

La solution implémentée par *esup-commons* est la suivante : lorsqu'une application veut envoyer des données au client sous forme de fichier :

- Elle positionne dans la session *HTTP* les attributs **`downloadData`**, **`downloadContentType`** et **`downloadFilename`**, qui contiennent respectivement les données, le type *MIME* et le nom du fichier à destination du client ;
- Elle redirige ensuite le navigateur vers la *servlet* de téléchargement,
- La *servlet* de téléchargement, requêtée par le client, puise les informations laissées par l'application dans la session *HTTP* et les envoie sous forme de fichier au client.

La *servlet* de téléchargement est déclarée ainsi dans le fichier **`web.xml`** :

```
<servlet>
  <servlet-name>Download Servlet</servlet-name>
  <servlet-class>
    org.esupportail.commons.web.servlet.DownloadServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Download Servlet</servlet-name>
  <url-pattern>/download</url-pattern>
</servlet-mapping>
```

Comment s'en servir (partie visible)

Lorsqu'un programmeur souhaite envoyer au client un fichier, il utilise la class statique **`DownloadUtils`**.

Si le fichier existe sur le système de fichiers, il appellera simplement la méthode **`setDownload()`** en passant le chemin du fichier sur le système de fichiers. Par exemple :

```
DownloadUtils.setDownload("/usr/local/esup/copyright.zip");
```

S'il a en sa possession le fichier sous forme d'une instance de **`java.io.File`**, il la passera de la même manière :

```
DownloadUtils.setDownload(file);
```

Il peut également envoyer des données sous la forme d'un tableau de **`byte`** :

```
byte [] bytes = ...;
DownloadUtils.setDownload(bytes, "filename.txt", "text/plain");
```

Notes :

- L'appel de la méthode **`setDownload()`** positionne les attributs en session et effectue la redirection.
- Si le nom du fichier et/ou le type *MIME* n'est pas spécifié, le navigateur se débrouillera tout seul sans.

Exercice 47 : Envoyer un fichier au client

Ajouter un bouton Téléchargement sur la vue **`test2.jsp`** qui envoie au client un fichier quelconque (tester avec plusieurs types différents).

24.2. Réception d'un fichier par le serveur

Utilisation

Le formulaire de téléchargement doit utiliser la balise *Tomahawk* **`t:inputUploadFile`**, dont la valeur doit pointer sur un attribut de contrôleur de type **`UploadedFile`**.

```
<h:form id="uploadForm" enctype="multipart/form-data" >
  <t:inputFileUpload id="file"
      value="#{filesController.uploadedFile}"
      storage="memory"
      required="true"/>
  <h:commandButton value="#{msgs['FILES.BUTTON.ADD']}"
      action="#{filesController.uploadFile}"/>
  <e:message for="file" />
</h:form>
```

La méthode associée au formulaire s'appuie alors sur l'attribut **`uploadedFile`** de cette manière (on stocke ici les fichiers sous le nom qu'ils avaient sur le client dans un répertoire donné par la méthode **`getUploadedFilesDirectory()`**) :

```
public String uploadFile() {
  if (uploadedFile == null) {
    addErrorMessage(null, "FILES.MESSAGE.NO_FILE_GIVEN");
    return null;
  }
  String name = null;
  try {
    name = uploadedFile.getName();
    // a hack for IE
    if (name.contains("\\")) {
      name = name.substring(name.lastIndexOf('\\') + 1);
    }
    OutputStream out = new FileOutputStream(
        new File(getUploadedFilesDirectory() +   "/" + name));
    InputStream in = uploadedFile.getInputStream();
    byte[] buf = new byte[UPLOADED_FILES_BUFFER_SIZE];
    int len;
    while ((len = in.read(buf)) > 0) {
      out.write(buf, 0, len);
    }
    in.close();
    out.close();
  } catch (IOException e) {
    addErrorMessage(null, "FILES.MESSAGE.UPLOAD_ERROR", name,
    e.getMessage());
  }
  addInfoMessage(null, "FILES.MESSAGE.UPLOAD_OK", name);
  return null;
}
```

Configuration

En déploiement servlet

La balise *Tomahawk* **`t:inputUploadFile`** nécessite l'utilisation du filtre *Tomahawk* **`MyFacesExtensionsFilter`**, qui doit être configuré dans le fichier **`web.xml`** de la manière suivante.

On déclare le filtre :

```
<filter>
  <filter-name>MyFacesExtensionsFilter</filter-name>
  <filter-class>
    org.apache.myfaces.webapp.filter.ExtensionsFilter
  </filter-class>
  <init-param>
    <param-name>maxFileSize</param-name>
    <param-value>20m</param-value>
  </init-param>
</filter>
```

On applique ensuite le filtre aux pages *JSF* de l'application :

```
<filter-mapping>
  <filter-name>MyFacesExtensionsFilter</filter-name>
  <servlet-name>esup-example</servlet-name>
</filter-mapping>
```

Si l'on souhaite utiliser les extensions *Tomahawk* comme les arbres (**`tree2`**), on applique le filtre à une autre URL, utilisée par *Tomahawk* pour délivrer les ressources statiques (scripts, images, …) :

```
<filter-mapping>
  <filter-name>MyFacesExtensionsFilter</filter-name>
  <url-pattern>/faces/myFacesExtensionResource/*</url-pattern>
</filter-mapping>
```

En déploiement portlet

Les filtres des *servlets* ne sont directement applicables aux *portlets*. On utilise alors les bibliothèques *tomahawk-bridge*, *portals-bridge-filter* et une version modifiée de *faces-response-filter*.

On indique tout d'abord dans **`web.xml`** que la *servlet* principale est **`FilterPortlet`**, de la bibliothèque *portals-bridge-filter* :

```
<servlet>
  <servlet-name>esup-example</servlet-name>
  <servlet-class>org.apache.pluto.core.PortletServlet</servlet-class>
  <init-param>
    <param-name>portlet-class</param-name>
    <param-value>
        org.apache.portals.bridges.portletfilter.FilterPortlet
    </param-value>
  </init-param>
  <init-param>
    <param-name>portlet-guid</param-name>
    <param-value>esup-example.esup-example</param-value>
  </init-param>
</servlet>
```

On applique ensuite deux filtres à la *portlet* **FacesPortlet** de *esup-commons*, dans **portlet.xml** :

```
<portlet>
  <portlet-name>esup-example</portlet-name>
  <portlet-class>
    org.apache.portals.bridges.portletfilter.FilterPortlet
  </portlet-class>
  <init-param>
    <name>portlet-class</name>
    <value>org.esupportail.commons.web.portlet.FacesPortlet</value>
  </init-param>
  <init-param>
    <name>portlet-filters</name>
    <value>
jp.sf.pal.facesresponse.FacesResponseFilter,jp.sf.pal.tomahawk.filter.
ExtensionsPortletFilter
    </value>
  </init-param>
  <init-param>
    <name>default-view</name>
    <value>/stylesheets/welcome.jsp</value>
  </init-param>
  <init-param>
    <name>default-view-selector</name>
    <value>
    org.esupportail.commons.web.deepLinking.UportalDefaultViewSelector
    </value>
  </init-param>
  <expiration-cache>0</expiration-cache>
  <supports>
    <mime-type>text/html</mime-type>
  </supports>
  <portlet-info>
    <title>esup-example</title>
    <short-title>esup-example demo portlet</short-title>
    <keywords>esup example</keywords>
  </portlet-info>
</portlet>
```

Il faut également en déploiement *portlet* ne pas vérifier la présence du filtre *MyExtensionsFilter* dans le fichier **web.xml** :

```
<context-param>
  <param-name>org.apache.myfaces.CHECK_EXTENSIONS_FILTER</param-name>
  <param-value>false</param-value>
</context-param>
```

25. La gestion des caches

Si la gestion du cache est en général anecdotique (peu importante en terme de développement), elle intervient en général à plusieurs endroits dans les applications et son intégration dans *esup-commons* permet :

- D'uniformiser les mécanismes utilisés,
- De centraliser la configuration des caches.

esup-commons s'appuie sur la bibliothèque *Ehcache*. Elle est utilisée en particulier pour cacher les requêtes à l'annuaire *LDAP*, ainsi qu'à cacher les exceptions remontées par courrier électronique.

L'utilisation de *EhCache* à toute autre fin est très simple, comme nous le montrons ci-après.

25.1. Configuration

Un *bean* gestionnaire de caches nommé **`cacheManager`** est déclaré dans le fichier de configuration **`/properties/cache/cache.xml`** :

```
<bean
    id="cacheManager"
    class="[...].cache.ehcache.EhCacheManagerFactoryBean"
    >
  <property
      name="configLocation"
      value="classpath:/properties/cache/ehcache.xml" />
</bean>
```

Comme on le voit, la configuration de ce gestionnaire de caches est localisée dans le fichier de propriétés **`/properties/cache/ehcache.xml`**, il est au format *EhCache*.

Le lecteur se reportera à la documentation complète de *EhCache* pour peaufiner la configuration des caches. Elle est simple et consiste à définir des paramètres pour chaque zone de cache. On trouvera par exemple pour le cache des requêtes *LDAP* :

```
<cache
  name="org.esupportail.commons.services.ldap.CachingLdapServiceImpl"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="300"
  timeToLiveSeconds="600"
  overflowToDisk="true"
/>
```

25.2. Utilisation

L'utilisation dans le code *Java* est très simple, il faut seulement s'assurer que tout objet stocké en cache implémente l'interface **`Serializale`**.

Lorsque l'on veut cacher un objet, il faut un cache, que l'on récupère de cette manière à partir d'un gestionnaire de cache :

```
String cacheName = getClass().getName() ;
if (!cacheManager.cacheExists(cacheName)) {
  cacheManager.addCache(cacheName);
}
cache = cacheManager.getCache(cacheName);
```

Note : le nom donné à la récupération du cache correspond à l'attribut **`name`** de la configuration de *EhCache*.

Il faut ensuite définir comment associer une clé unique à tout objet que l'on veut stocker dans le cache. On prendra souvent une chaîne de caractère, mais pas toujours.

Par exemple, si l'on veut stocker le résultat d'une requête *HTTP*, on prendra comme clé l'*URL* de la requête. Un court morceau de code illustrant comment on peut cacher des requêtes est le suivant :

```
String getUrlContent(String url) {
  Element element = cache.get(url);
  if (element != null) {
    return (String) element.getObjectValue();
  }
  String content = // do the request here
  cache.put(new Element(url, content);
  return content;
}
```

26. Envoi de courrier électronique

Le service d'envoi de courriers électroniques a été initialement développé pour le gestionnaire d'exceptions (certaines implémentations permettent d'envoyer les rapports d'exception par courrier électronique). Ce service est disponible à toute autre fin (envoi d'informations, remontée d'alertes, …).

26.1. Les implémentations

esup-commons prévoit plusieurs implémentations du service d'envoi de mails :

- **`SimpleSmtpServiceImpl`** permet l'envoi simple de mails.
- **`AsychronousSmtpServiceImpl`** permet l'envoi de mails à l'aide d'un fil d'exécution spécifique de priorité basse. Cette fonctionnalité évite les timeout observés lors de l'expédition d'un grand nombre de mails.

C'est cette dernière implémentation qui est généralement conseillée.

26.2. Configuration

Le service d'envoi de mail se configure à l'aide du fichier **`/properties/smtp/smtp.xml`**.

Exemple de déclaration :

```
<bean
  id="smtpService"
  class="org.esupportail.commons.services.smtp.AsynchronousSmtpService
Impl">
  <property name="servers">
    <list>
      <ref bean="smtpServer1" />
    </list>
  </property>
  <property name="fromAddress" ref="smtpFromAddress"/>
  <property name="interceptAddress" ref="smtpInterceptAddress"/>
  <property name="testAddress" ref="smtpInterceptAddress"/>
</bean>
```

Propriétés :

- **`Servers`** est une liste de serveurs à utiliser pour envoyer les messages. Les serveurs sont utilisés suivant leur ordre d'apparition dans le fichier. Si, et seulement si, une exception est levée lors de l'envoi du mail sur un serveur le suivant est utilisé.
- **`fromAddress`** pointe vers un *bean* définissant l'expéditeur des messages.
- **`interceptAddress`** pointe vers un *bean* définissant le destinataire des messages en phase de mise au point. Tous les messages sont envoyés à cette adresse afin d'éviter un effet de *spam* (y compris les exceptions, la propriété **`recipientEmail`** du fichier **`/properties/exceptionHandling/exceptionHandling-example.xml`** n'étant pas prise en compte). Il suffit de supprimer cette propriété pour retrouver un comportement normal.
- **`testAddress`** pointe vers un *bean* définissant le destinataire du message généré par la tâche ant **`test-smtp`**.

Les *beans* **`smtpFromAddress`**, **`smtpInterceptAddress`** et **`smtpInterceptAddress`** sont tous de type **`javax.mail.internet.InternetAddress`**.

Exemple de déclaration :

```
<bean
  id="smtpFromAddress"
  class="javax.mail.internet.InternetAddress">
    <property name="address" value="webmaster@domain.org" />
    <property name="personal" value="WebMaster" />
</bean>
```

Propriétés :

- **address** est l'adresse email.
- **Personal** est le nom de l'expéditeur qui apparaîtra dans le client de messagerie du destinataire.

Les *beans* listés dans la propriété **Servers** de **smtpService** sont tous de type **org.esupportail.commons.services.smtp.SmtpServer**.

Exemple de déclaration :

```
<bean
  id="smtpServer1"
  class="org.esupportail.commons.services.smtp.SmtpServer">
    <property name="host" value="smtp.domain.org"/>
    <property name="port" value="25"/>
    <property name="user" value=""/>
    <property name="password" value=""/>
</bean>
```

Propriétés :

- **host** est l'adresse internet du serveur SMTP
- **port** est le port du service SMTP sur la machine **host**
- **user** est le nom de l'utilisateur utilisé pour envoyer les messages. Si cette propriété est vide ou commentée la connexion au service SMTP se fait en anonyme.
- **password** est le mot de passe lié à **user**. Cette propriété est ignorée en cas de connexion anonyme.

26.3. Utilisation

Pour qu'un de vos *beans* puisse utiliser le service d'envoi de mails il faut qu'il ait une référence, définie dans le fichier de configuration *Spring*, sur un **smtpService**.

Ensuite il peut utiliser ce service, par exemple :

```
smtpService.send(to, emailSubject, htmlBody, textBody);
```

- **to** est le destinataire (un **InternetAddress**).
- **emailSubject** est le sujet du mail.
- **htmlBody** et **textBody** sont, respectivement, les contenus HTML et texte du message. Au moins un de ces deux paramètres doit être non **null**. Si les deux sont renseignés, le mail généré est alors de type **MimeMultipart**.

27. Intégration de *FckEditor*

Afin de disposer de manière native d'un éditeur *WYSIWYG*, *esup-commons* propose une intégration de *FckEditor* via *FCK Faces* (cf. http://sourceforge.net/projects/fck-faces/).

27.1. Utilisation

L'utilisation de *FckEditor* est très simple. Il suffit d'ajouter dans votre formulaire une balise **`<fck:editor>`** et d'y associer un attribut de votre contrôleur. On trouvera par exemple dans un formulaire :

```
<fck:editor value="#{testController.text}" toolbarSet="Example"/>
```

L'attribut **`toolbarSet`** est facultatif et précise le menu à utiliser. S'il n'est pas présent, le menu **`Default`** est utilisé.

Le *taglig* de préfix **`fck`** doit être défini dans l'entête de votre page *JSP*, en utilisant la syntaxe *JSP* classique :

```
<%@ taglib prefix="fck" uri="http://www.fck-faces.org/fck-faces"%>
```

Ou bien la syntaxe *XML* :

```
<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    version="2.0"
    xmlns:f=http://java.sun.com/jsf/core
    xmlns:h=http://java.sun.com/jsf/html
    xmlns:t=http://myfaces.apache.org/tomahawk
    xmlns:fck="http://www.fck-faces.org/fck-faces">
```

27.2. Configuration

Le fichier **`web.xml`** définit un paramètre du contexte d'exécution nommé **`org.fckfaces.CUSTOM_CONFIGURATIONS_PATH`** qui permet de préciser à *FCK Faces* quel fichier *JavaScript* de configuration doit être utilisé :

```
<context-param>
  <param-name>org.fckfaces.CUSTOM_CONFIGURATIONS_PATH</param-name>
  <param-value>/media/FCKeditor/custom-fckconfig.js</param-value>
</context-param>
```

Ce fichier permet, par exemple, de définir une barre de menu spécifique, comme précisé dans l'attribut **`toolbarSet`** de la balise **`<fck:editor>`** :

```
FCKConfig.ToolbarSets["Example"] = [
  ['Undo', 'Redo'],
  ['Cut', 'Copy', 'Paste', 'PasteText'],
  ['FontFormat'],
  ['Bold', 'Italic', 'Underline', 'StrikeThrough', '-',
   'Subscript', 'Superscript', '-', 'RemoveFormat'],
  ['OrderedList', 'UnorderedList', '-', 'Outdent', 'Indent'],
  ['Link', 'Unlink']
];
```

Exercice 48 : Utiliser *FckEditor*

Ajouter un éditeur *WYSIWYG* à la vue **`test2.jsp`** et lier la valeur du texte de l'éditeur à l'attribut value de **`test2Controller`**.

28. Accès à des services protégés par *CAS*

Certaines applications peuvent avoir besoin d'accéder à des services protégés par CAS (des *backend services*). Elle doivent pour cela transmettre à ces services un *Proxy Ticket CAS* (*PT*), récupéré à partir du serveur CAS sur présentation d'un *PGT*. Elles doivent donc avoir en leur possession ce *PGT*, valable pour l'application, et pour cet utilisateur seulement.

28.1. *Configuration en déploiement* portlet

Pour un déploiement *portlet*, la stratégie employée est la suivante : le portail (*uPortal*) passe à la *portlet* un *PT*, à partir duquel la *portlet* pourra récupérer le *PGT* convoité. Cela implique des modifications dans *uPortal*, légères.

Nous montrons tout d'abord les modifications à effectuer dans *uPortal* pour qu'un *PT* soit passé aux *portlets* via une préférence *JSR-168*.

Nous montrons enfin comment, dans esup-commons, on récupère un *PT* que l'on pourra ensuite passer à des services pour authentifier l'utilisateur connecté.

Passage du PT du portail vers une portlet

Nouvel adaptateur pour les portlets *: `CCasProxyPortletAdapter`*

Récupérer ou écrire Ecrire la classe **`org.esupportail.portal.channels.CCasProxyPortletAdapter`**, qui hérite de **`CPortletAdapter`** en surchargeant sa méthode **`getUserInfo()`**. Cette méthode est appelée une fois à l'initialisation du canal pour remplir une table des paramètres passés à la *portlet* via le mécanisme de préférences de *JSR-168*. On y ajoute simplement ajoute un paramètre qui contient le *PT*.

Le nom du paramètre et l'*URL* pour lequel le *PT* est récupéré sont passés en paramètre à la publication de la *portlet* dans *uPortal* (cf ci-dessous). Ces deux paramètres sont :

- **`casProxyTicketPref`**, optionnel, par défaut **`casProxyTicket`**.
- **`casTargetService`**, obligatoire

Nouveau type de canal : `CasProxyPortlet`

On pourrait publier les *portlets proxy CAS* de type **`Custom`** en utilisant l'adaptateur ci-dessus, mais il semble plus indiqué de créer un nouveau type de canal dédié pour l'occasion. Pour cela, on crée donc un nouveau type de canal dans la base *uPortal*, de nom **`CasProxyPortlet`**, qui utilise la classe **`org.esupportail.portal.channels.CCasProxyPortletAdapter`** (au lieu de **`org.jasig.portal.channels.CPortletAdapter`** pour le type **`Portlet`**).

Dans une version ultérieure de *uPortal-esup*, on pourra ajouter les lignes suivantes à **<uportal>/properties/db/esup-data.xml** pour la table **UP_CHAN_TYPE** :

```
<row>
  <column><name>TYPE_ID</name><value>12</value></column>
  <column>
    <name>TYPE</name><value>[...].CCasProxyPortletAdapter</value>
  </column>
  <column>
    <name>TYPE_NAME</name><value>CasProxyPortlet</value>
  </column>
  <column>
    <name>TYPE_DESCR</name>
    <value>Adapter for CAS proxy JSR-168 Portlets</value>
  </column>
  <column>
    <name>TYPE_DEF_URI</name>
    <value>[...]/CCasProxyPortletAdapter.cpd</value>
  </column>
</row>
```

Publication de la portlet

Il faut publier la *portlet* en utilisant le nouvel adaptateur, on utilisera par exemple le fichier de configuration *XML* suivant :

```
<channel-definition>
    ...
    <type>CasProxyPortlet</type>
    <class>[...].channels.portlet.CCasProxyPortletAdapter</class>
    ...
    <parameters>
        <parameter>
            <name>portletDefinitionId</name>
            <value>esup-blank.esup-blank</value>
            <ovrd>N</ovrd>
        </parameter>
        <parameter>
            <name>casProxyTicketPref</name>
            <value>casProxyTicket</value>
            <description>The name of the JSR-168 preference used to
              pass the proxy ticket</description>
            <ovrd>N</ovrd>
        </parameter>
        <parameter>
            <name>casTargetService</name>
            <value>http://portal.domain.edu/application</value>
            <description>The CAS service of the portlet</description>
            <ovrd>N</ovrd>
        </parameter>
    </parameters>
</channel-definition>
```

C'est tout ce qu'il faut faire au niveau de *uPortal*. Une fois ces modifications faites, la *portlet* dispose d'une préférence supplémentaire (**casProxyTicket** par défaut), qui contient un *PT* pour l'utilisateur courant.

Le paramètre **casTArgetService** indique à *uPortal* pour quel service récupérer le *PT* à passer. Il ne s'agit pas d'une *URL* réelle, mais cette même URL doit être précisée à

l'initialisation du bean **casService** (propriété **service**, voir plus loin) qui sert à récupérer de nouveaux *PTs* pour les services accédés par l'application.

Note : l'adaptateur **CCasProxyPortletAdapter** peut également être utilisé pour publier des *portlets* qui n'ont pas besoin de ticket *CAS* ; il suffit dans ce cas de ne pas spécifier de paramètre **casTargetService** à la publication de la *portlet*.

Récupération du PT *transmis par le portail par la* portlet

Le PT transmis par le portail est récupéré par le *bean* **casService**, défini par exemple dans le fichier de configuration *Spring* **/properties/auth/auth.xml** :

```
<bean
    id="casService"
    class="[...].commons.services.cas.PortletCasServiceImpl" >
    <property
      name="service"
      value="http://portal.domain.edu/application" />
    <property
      name="casValidateUrl"
      value="https://cas.domain.edu/proxyValidate" />
    <property
     name="proxyCallbackUrl"
     value="https://portal.domain.edu/application/casProxyCallback" />
</bean>
```

Les propriétés du bean **casService** sont expliquées plus loin dans ce document.

C'est le *bean* **casService** qui récupère le *PT* transmis par le portail, à travers les préférences *JSR-168*. Par défaut, il s'attend à trouver le *PT* dans la préférence nommée **casProxyTicket** ; il est possible d'utiliser une autre préférence, dont on spécifie alors le nom à travers la propriété **casProxyTicketPref** du *bean* **casService**.

28.2. Configuration en déploiement servlet

TODO PA: à compléter

28.3. Utilisation du bean *casService*

Récupération d'un PGT

Afin de récupérer des *PTs* pour des services distants, l'application doit tout d'abord récupérer un *PGT*, obtenu en validant :

- le *PT* transmis par le portail dans le cas d'un déploiement *portlet* ;
- le *ST* ou le *PT* trouvé dans l'URL dans le cas d'un déploiement *servlet*.

La récupération du *PGT* est automatiquement faite « *au plus tard* » par le bean **casService**, c'est-à-dire juste avant de demander un premier *PT* pour un service distant ; la récupération du *PGT* est ainsi complètement transparente.


Il est possible de forcer la récupération du *PGT* avant la récupération du premier *PT* pour un service distant, pour éviter que le *PT* transmis par le portail ne se périme avant qu'on ne le valide. Il suffit alors d'appeler la méthode **validate()** du *bean* **casService**.

Les propriétés **service**, **casValidateUrl** et **proxyCallbackUrl** du *bean* **casService** ont les significations suivantes :

- La propriété **service** indique pour quel service récupérer des *PT*s. Cette propriété n'est pas une *URL* réelle, mais elle doit strictement correspondre au paramètre **casTargetService** utilisé lors de la publication de la *portlet*.
- La propriété **casValidateUrl** est l'*URL* du serveur *CAS* utilisée pour valider le *PT* transmis par le portail à la *portlet*.
- La propriété **proxyCallbackUrl** est l'*URL* utilisée par le serveur *CAS* pour communiquer à la *portlet* le *PGT*.


Pour que le *PGT* soit récupéré par l'application, il faut que l'application soit à l'écoute sur une *URL* de *callback*, celle donnée à la propriété **proxyCallbackUrl** du *bean* **casService**. Pour cela, il faut configurer une *servlet* supplémentaire dans le contexte *Tomcat* de l'application, dans le fichier **/properties/WEB-INF/web.xml** :

```
<servlet>
    <servlet-name>CasProxyCallback</servlet-name>
    <servlet-class>
      edu.yale.its.tp.cas.proxy.ProxyTicketReceptor
    </servlet-class>
    <init-param>
      <param-name>edu.yale.its.tp.cas.proxyUrl</param-name>
      <param-value>https://cas.domain.edu/proxy</param-value>
    </init-param>
    <load-on-startup>4</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>CasProxyCallback</servlet-name>
    <url-pattern>/casProxyCallback</url-pattern>
</servlet-mapping>
```

Récupération d'un PT *pour un service distant*

Il suffit simplement d'appeler la méthode **getProxyTicket()** du *bean* **casService** en précisant pour quel service distant on demande un *PT* :

```
String casTargetService = "http://webdav.domain.edu";
String pt = null;
try {
    pt = casService.getProxyTicket(casTargetService);
} catch (CasException e) {
    ...
}
```

Le *PT* obtenu peut alors être passé au service distant, par un moyen propre à l'application.


Il possible de configurer le *bean* **casService** pour qu'il effectue plusieurs tentatives en cas d'échec lors de la récupération du *PGT* ou des *PT*s pour les services distants. Il suffit pour cela d'initialiser les valeurs des propriétés **retries** (nombre de tentatives à refaire en cas d'échec, par défaut **0**) et **sleep** (nombre de secondes d'attente entre chaque tentative, par défaut **1**). Il convient d'utiliser des valeurs raisonnables, notamment en déploiement *portlet* car trop d'essais mèneraient à un *timeout* du canal de la *portlet*.

Annexe A. La gestion des *logs*

esup-commons propose une implémentation des *logs* basée, en interne, sur *commons-logging* d'Apache. *commons-logging* permet d'utiliser différents mécanismes de *log* (standard *Java*, *Log4j*, etc.). *esup-commons* est structuré pour utiliser *Log4j* et bénéficie ainsi de sa grande flexibilité de configuration.

Utilisation dans le code Java

Pour pouvoir utiliser un *logger* dans une classe de votre application vous devez le définir. Suivant le cas, vous aurez deux définitions. Exemple :

```
private static final Logger LOGGER = new LoggerImpl(NonClasse.class);
```

```
private final Logger logger = new LoggerImpl(getClass());
```

Le premier exemple est adapté à l'utilisation d'un *logger* à l'intérieur d'une classe utilitaire constituée de méthodes définies static. Le second exemple est adapté aux classes dynamiques. Dans ce cas l'utilisation de getClass() permet d'avoir une information sur la classe concrète utilisée. C'est particulièrement utile en cas d'héritage de classes.

Ensuite vous pouvez utiliser ce *logger* dans vos méthodes pour loguer en mode **`TRACE`**, **`DEBUG`**, **`INFO`**, **`WARN`** ou **`ERROR`**. Exemple :

```
logger.error("Nous avons un problème");
```

Afin de ne pas pénaliser les performances avec la gestion des *logs* en mode **`DEBUG`** et **`TRACE`** il est conseillé de tester leur activation. Exemple :

```
if (logger.isDebugEnabled()) {
  logger.debug("set language [" + locale + "] for user '" +
    currentUser.getId() + "'");
}
```

Notes : la classe **`LoggerImpl`** implémente l'interface **`Logger`** et vous propose des méthodes utilitaires qui permettent, par exemple, de loguer des listes de **`String`** on bien encore un temps d'exécution.

Configuration

La configuration *Log4j* est à faire dans les fichiers **`/properties/log4j.properties`** et **`/properties/log4j-batch.properties`** pour, respectivement, une utilisation en mode *servlet* ou *portlet*, une utilisation en mode *batch*.

Pour plus de renseignements sur la structure du fichier de propriétés, le lecteur se reportera à la documentation de *Log4j*.

Comment ça marche

En mode web (servlet *ou* portlet)

Les *logs* doivent être configurés au niveau du contexte de l'application, pour que chaque application tournant dans le *Tomcat* puisse être indépendante à ce niveau.

On configure la location du fichier de propriétés *Log4j* à l'aide du fichier **`web.xml`**.

On ajoute tout d'abord un paramètre de contexte nommé **webAppRootKey**. Ce paramètre est utile à *log4j* quand plusieurs applications fonctionnent sur le même serveur d'applications afin de loguer de façon spécifique pour chacune d'elles. Exemple :

```
<context-param>
  <param-name>webAppRootKey</param-name>
  <param-value>esup-lecture</param-value>
</context-param>
```

On ajoute ensuite un paramètre de contexte nommé **contextConfigLocation** pour préciser la localisation du fichier de configuration de *log4j*, par exemple :

```
<context-param>
  <param-name>log4jConfigLocation</param-name>
  <param-value>
    classpath:/properties/logging/log4j.properties
  </param-value>
</context-param>
```

On ajoute ensuite un *listener Spring* spécifique (qui doit être présent avant tout autre *listener Spring*) :

```
<listener>
  <listener-class>
    org.springframework.web.util.Log4jConfigListener
  </listener-class>
</listener>
```

En mode batch

On configure la location du fichier de propriétés *Log4j* à l'aide de la variable d'environnement **log4j.configuration**. Cette dernière est positionnée par *ant* dans la tâche permettant de lancer le *batch*. Exemple :

```
<target
    name="test-ldap"
    depends="commons-compile-batch,check-commons-batch-config"
    description="test the LDAP connection">
  <java fork="true" dir="${build.dir}"
    classname="org.esupportail.commons.batch.Batch">
    <classpath refid="batch.classpath" />
    <sysproperty key="log4j.configuration"
      value="file:${log4j.batch-config}" />
    <arg value="test-ldap"/>
  </java>
</target>
```

Annexe B. Les feuilles de styles (CSS)

Les feuilles de styles utilisées sont situées dans le répertoire **`/webapp/media`**. Leur utilisation est configurée dans le fichier de configuration *Spring* **`/properties/tags/tags.xml`**.

Déploiement portlet

En déploiement *portlet*, l'utilisation de la balise **`<e:page>`** de la *taglib esup-commons* n'inclut pas les feuilles de style spécifiées par le *bean* **`tagsConfigurator`**. En effet, dans le cas d'une *portlet*, le code *XHTML* renvoyé au portail ne comporte pas l'entête *HTML* **`<head>`**, où sont déclarées les feuilles de style. Par ailleurs, c'est bien un des rôles du portail de gérer l'apparence des canaux pour garantir l'homogénéité de l'ensemble.

En conséquence, pour un déploiement *portlet*, il faut modifier, si nécessaire, les feuilles de style au niveau du portail.

L'exploitant pourra s'appuyer facilement sur les feuilles de styles du répertoire **`/webapp/media`** et les adapter à son propre *look*.

Déploiement servlet

En déploiement *servlet*, l'utilisation de la balise **`<e:page>`** de la *taglib esup-commons* inclut automatiquement les feuilles de style spécifiées par le *bean* **`tagsConfigurator`**.

Si les feuilles de style fournies dans la distribution sont modifiées par l'exploitant, il doit alors penser à renseigner la propriété **`${custom.recover.files}`** (dans **`build.properties`**) pour qu'elles soient récupérées automatiquement par la tâche *ant* **`recover-config`**. (ce point sera étudié ultérieurement).

Déploiement quick-start

Le déploiement *quick-start* étant un déploiement *servlet* simplifié, la gestion des feuilles de style y est la même qu'en déploiement *servlet*.

Annexe C. Déboguage

Pour déboguer vos programmes nous proposons une solution basée sur l'utilisation de « *Java Platform Debugger Architecture* » (*JPDA*).

Cette solution est préférée à l'utilisation d'une solution basée sur un serveur d'applications intégrée à *Eclipse* pour différentes raisons. En effet, cette dernière solution manque de stabilité, impose un projet *Eclipse* particulier avec des contraintes dans les noms des répertoires et fait perdre le bénéfice de l'environnement ant de esup-commons.

Configurer votre projet

Pour utiliser le débogueur il faut éditer le fichier **`build-devel.properties`** pour y ajouter les paramètres suivants :

```
debug=true
```

```
jpdaAddress=5555
```

Le paramètre **`debug`** à **`true`** indique aux tâches *ant* de *esup-commons* de lancer *Tomcat* en mode *debug*.

Le paramètre **`JjpdaAdress`** permet d'indiquer le port qui devra être utilisé par le débogueur d'*Eclipse* pour communiquer avec le *Tomcat* lancé par le script ant de *esup-commons*.

Utilisation du débogueur d'Eclipse en mode JPDA

Nous recommandons au lecteur de suivre la très bonne documentation de Julien Marchal à ce sujet sans tenir compte de la partie introductive concernant le paramétrage de la *JVM* et de *Tomcat*. Cette partie est prise en charge par les paramètres vus ci-dessus. La documentation de Julien Marchal se trouve sur le site esup-portail.org :

http://www.esup-portail.org/consortium/espace/Normes_1C/tech/eclipse/debug.html

Historique de ce document

20070305 : version initiale 0.1

Version initiale pour la formation des 21 et 22 mars 2007 organisée pour le CRI de l'Université de Rennes 1.

20070325 : version 0.5

- Intégration des exercices
- Modification du plan (déploiement *servlet*/*portlet* en fin de formation)
- Ajout des parties manquantes
- Correction de nombreuses erreurs suite à la première session

20070330 : version 0.9

- Refonte de l'accès aux données (suppression de la dépendance à Hibernate)

20070423 : version 1.0

- Ajout de précisions sur le développement en *portlet*
- Ajout des annexes « gestion des logs » et « feuilles de style »
- Modification de la partie pratique sur les *web services*

20070515 : version 1.1

- Ajout de la configuration par fichiers de propriétés
- Ajout de la configuration pour l'utilisation des extensions *Tomahawk* en déploiement *portlet*

20070601 : version 1.2

- Ajout de l'authentification *CAS* à des services distants
- Ajout de la génération de la documentation (*HTML* et *docbook*)

20070909 : version 1.3

- Passage de *Callisto* à *Europa*
- Ajout de *HQL*
- Ajout du déboguage

20071008 : version 1.4

- Mise à jour pour esup-commons 0.17.2

Remerciements

Contributions

Pascal AUBRY
Université de Rennes 1

Laëtitia LE HIRESS
(alors étudiante à l'Université de Rennes 1)

Raymond BOURGES
Université de Rennes 1

Gwénaëlle BOUTEILLE
Université de Rennes 1

Cédric LEPROUST
Université de Rennes 1

Vincent BONAMY
Université de Rennes 1

Sylvain DE FEO
Université de Rennes 1

Rapports et corrections d'anomalies

- Benjamin BERTELLE
- Yves DESCHAMPS (Université de Lille 1)
- Olivier ZILLER (Université de Nancy 2)

Notes

Notes