

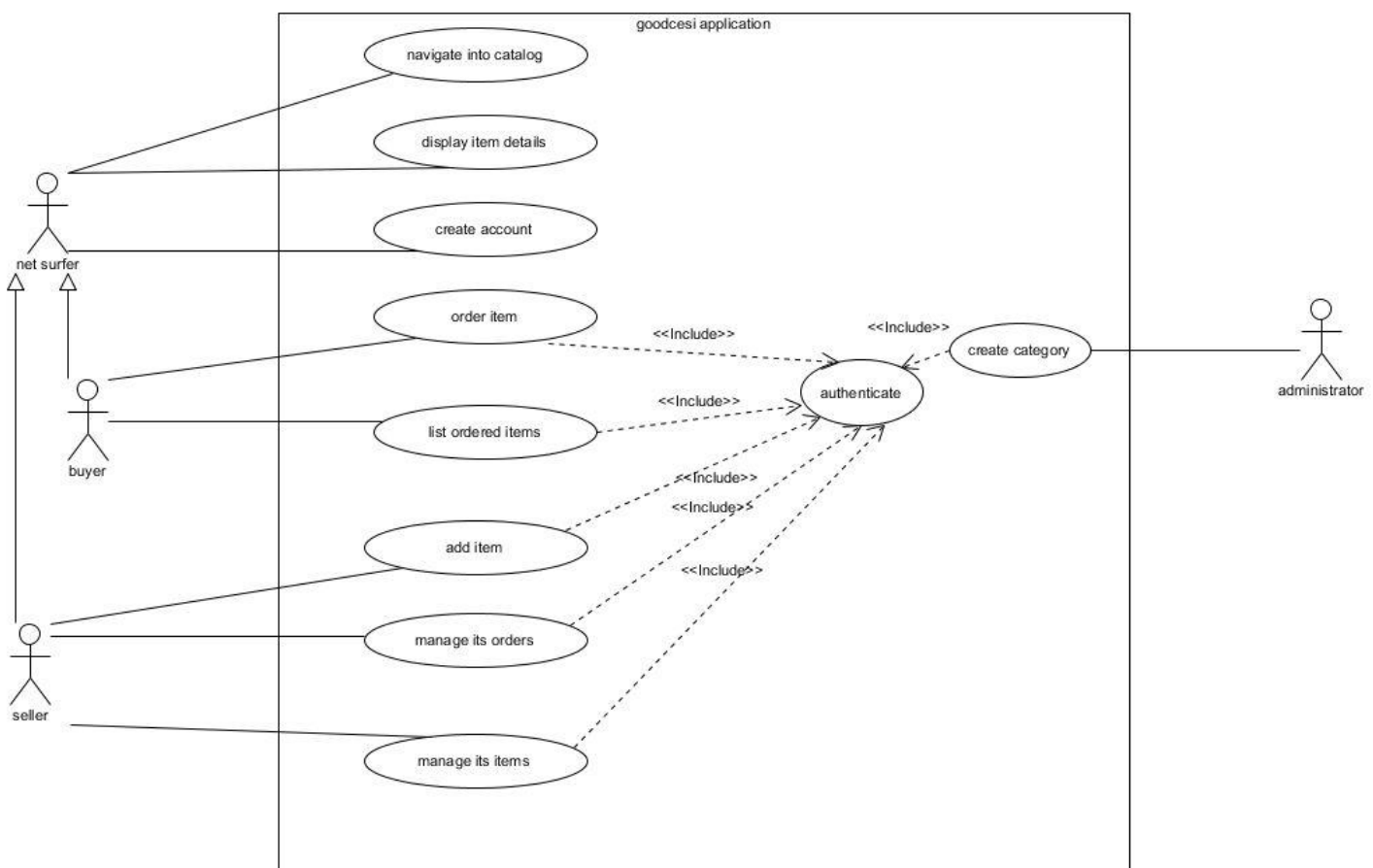
L'objectif de cette corbeille est de vous faire travailler sur les concepts de composants managés de type bean CDI et session beans : parmi les concepts abordés on retrouvera le mécanisme d'injection de dépendance, le concept d'objet contextuel, la notion de container et de cycle de vie des objets. Des parties optionnelles vous permettront de travailler sur des services supplémentaires fournis par ces deux standards.

Cette corbeille vous accompagnera pas à pas dans la mise en conformité de l'application goodcesi. Cette corbeille est à réaliser au fur et à mesure de votre AER pour cimenter vos connaissances via la pratique.

1. Architecture de GoodCesi

GoodCesi est un prototype d'application web de type le bon coin. L'application permet à des internautes d'acheter des produits d'occasion.

Les captures d'écran présentées se trouvent dans le dossier *images*.



gc_uc.jpg

Ce diagramme présente les fonctionnalités implémentées dans le prototype.

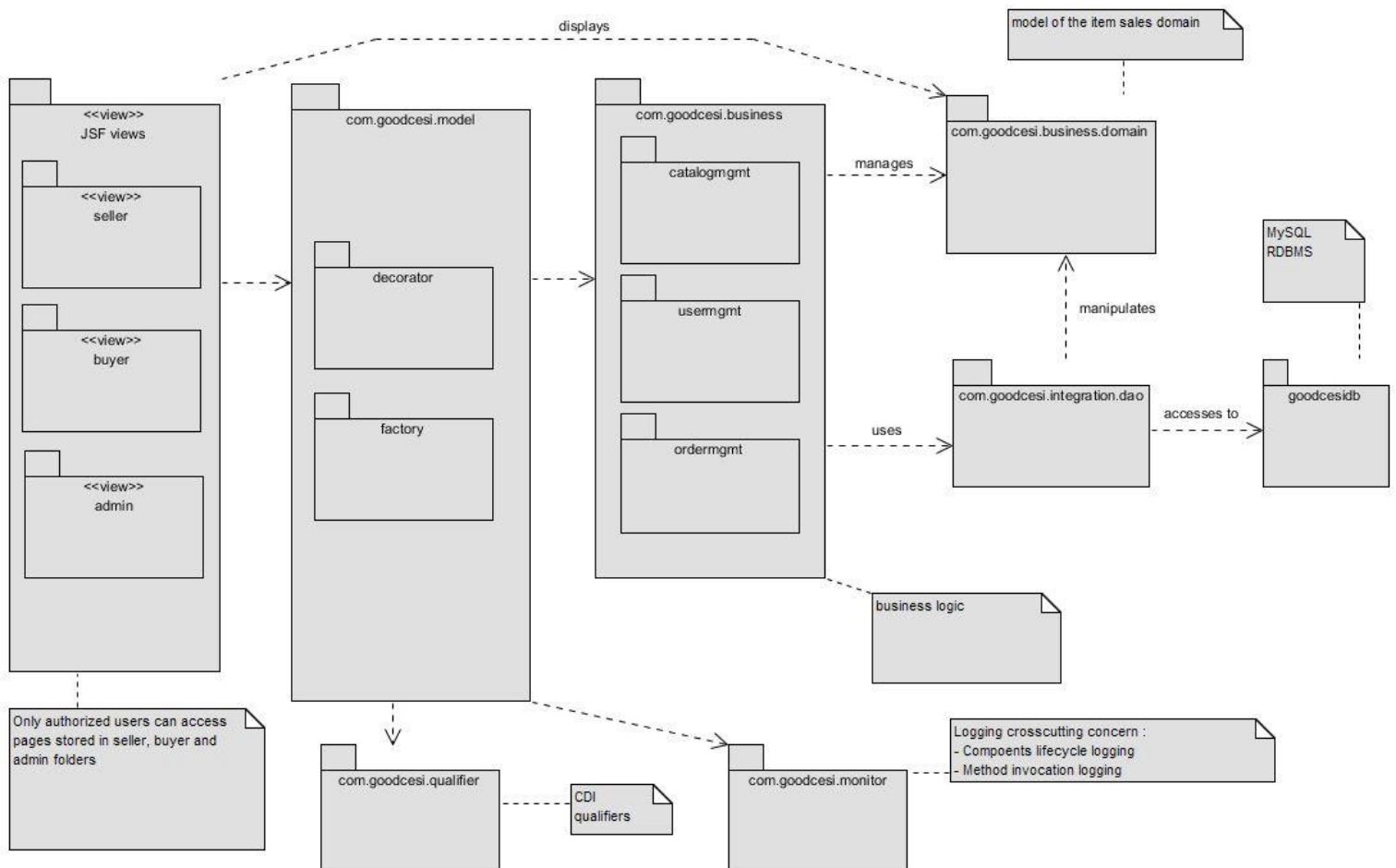
Les internautes correspondent à des utilisateurs non authentifiés. Ils peuvent parcourir le catalogue des produits. Ils ont la possibilité de s'inscrire afin d'accéder aux espaces réservés

aux acheteurs et aux vendeurs. Un internaute a la possibilité de s'inscrire en tant qu'acheteur et / ou vendeur. Concernant l'espace administrateur la seule fonctionnalité implémentée est la création de catégorie d'articles.

Il faut être authentifié et avoir le rôle d'acheteur pour pouvoir acheter un article. Un acheteur ne peut évidemment pas acheter ses propres articles. Un acheteur authentifié peut accéder à sa liste d'articles commandés. Seuls les articles marqués comme envoyés par le vendeur sont affichés.

Un vendeur peut lui mettre en ligne un nouvel article et modifier ses articles. Il peut aussi gérer ses commandes. Cette dernière fonctionnalité se limite à simplement passer une commande en statut « envoyée ».

Voyons maintenant, l'architecture logique de l'application au travers du diagramme de paquetages suivant. Chaque paquetage regroupe les artefacts / les composants ayant le même type de responsabilité au sein de l'application.



gc_pack.jpg

Le paquetage *vues JSF* n'est pas un paquetage au sens Java, il correspond aux vues web JSF qui vont fournir l'IHM au client (navigateur). Les sous-paquetages contiennent les pages uniquement accessibles à un type d'utilisateur. Par exemple, *buyer* contient les vues

uniquement accessibles aux utilisateurs ayant le rôle d'acheteur. En résumé, ce paquetage contient donc les artefacts ayant la responsabilité de fournir l'interface web.

Le paquetage *com.goodcesi.model* contient les composants chargés de lier les vues web JSF à la couche de traitement métier. Les composants sur lesquels vous travaillerez dans la première partie (Contexts and Dependency Injection) se trouvent tous dans ce paquetage.

Le paquetage *com.goodcesi.monitor* contient des composants permettant de journaliser, suivre ce qui se passe lors de l'exécution de l'application.

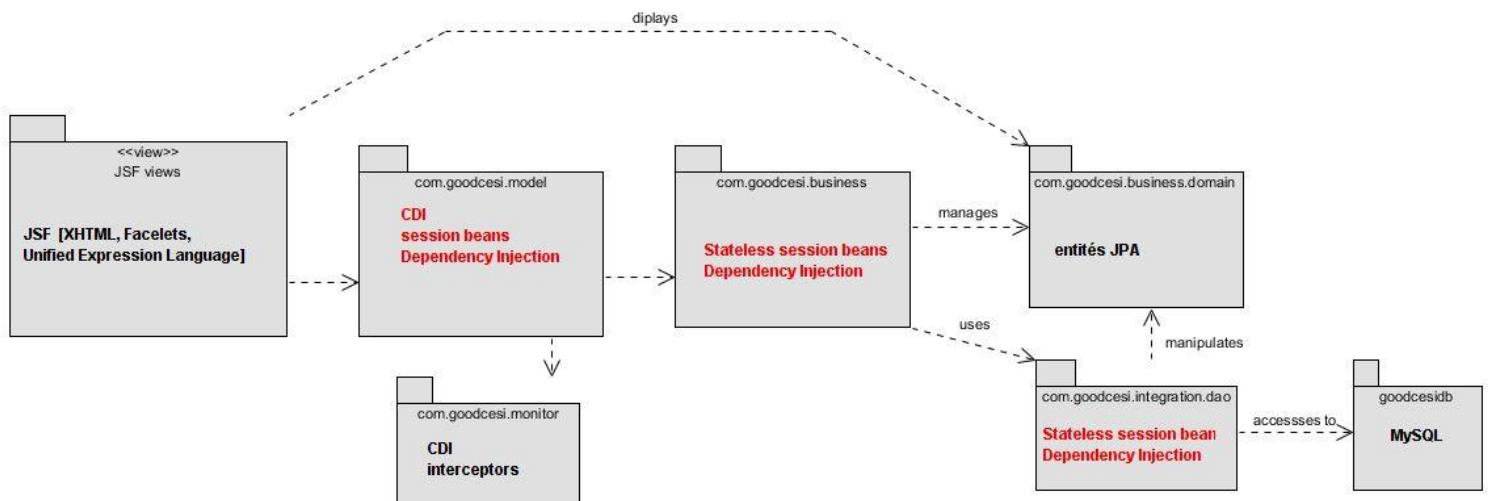
Le paquetage *com.goodcesi.business* correspond au cœur de la logique métier, c'est-à-dire qu'il contient les composants ayant la responsabilité d'implémenter les fonctionnalités métiers décrites dans le diagramme de cas d'utilisation. Ce paquetage utilise le paquetage *com.goodcesi.integration.dao* qui contient le service d'accès aux données.

Le paquetage *com.goodcesi.business.domain* contient les entités représentant les concepts du domaine implémenté, à savoir la vente en lignes d'articles de particuliers à particuliers. Dans le cas de GoodCesi, les concepts manipulés sont Utilisateur, Article, Commande, etc. Ces entités métiers sont gérées par les composants du paquetage *com.goodcesi.business*. Le composant d'accès aux données (DAO) intervient pour créer, supprimer, modifier, rechercher ses entités stockées en base.

Dans la deuxième partie (EJB- Session Beans) vous travaillerez sur les paquetages *com.goodcesi.domain*, *com.goodcesi.integration.dao* et aussi sur *com.goodcesi.model*.

Dans la 3^{ème} partie JSF vous travaillerez sur les vues JSF et un peu aussi sur le paquetage *com.goodcesi.model*.

Passons à l'architecture technique présentant une vue très haut niveau des technologies utilisées.



gc_archiTech.jpg

Ce diagramme présente les différentes technologies utilisées.

JSF est le standard pour créer une couche de présentation web dynamique. Ce standard est fortement intégré avec la spécification CDI. Vous étudierez les bases de JSF au cours des 4 premiers prosits / activités.

En rouges les technologies et concepts centraux à travailler dans cette corbeille.

La spécification CDI permet de lier les vues JSF avec le cœur de la logique métier (paquetage `com.goodcesi.business`). D'un point de vue architecture MVC les composants managés CDI vont faire office de point d'entrée dans le modèle.

Les session beans Stateless implémentent la logique métier, autrement-dit les fonctionnalités décrites précédemment dans le diagramme de cas d'utilisation. Ceux sont les services métier.

A noter que le standard Session Bean (EJB) est aussi utilisé dans le paquetage `com.goodcesi.model` pour augmenter les capacités de certains composants managés CDI.

Le mécanisme d'injection est utilisé pour « associer » les différents services /composants entre eux.

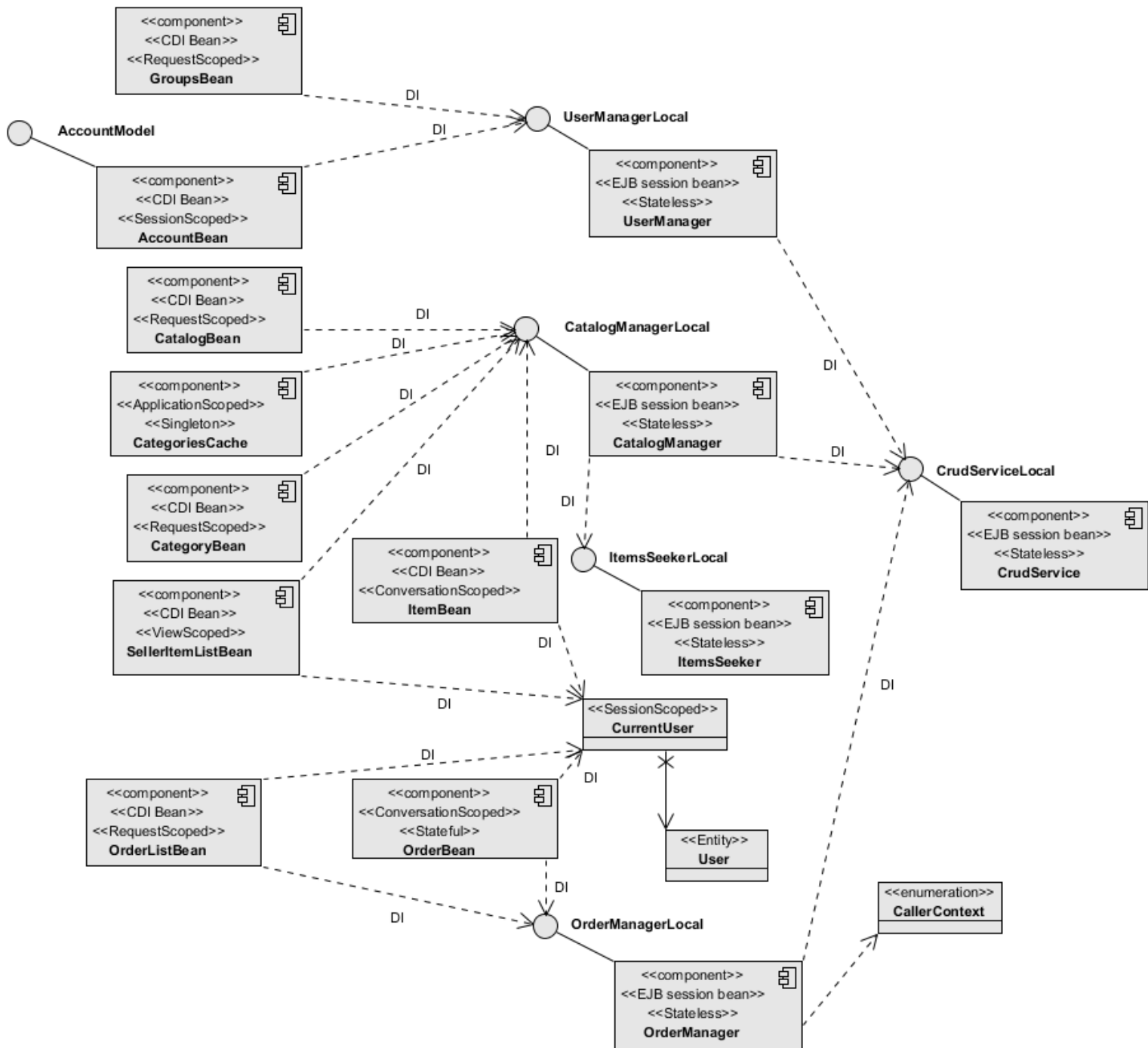
Injection, CDI, et session beans sont les concepts centraux que cette corbeille va vous faire pratiquer. Vous travaillerez aussi un peu JSF.

Les intercepteurs permettent de tracer le cycle de vie ou l'invocation de méthodes des différents composants managés CDI et session beans. Les intercepteurs java EE mettent en œuvre le concept de programmation orientée aspect.

JPA est la technologie ORM standard de Java EE. Ce standard sera travaillé dans le prosit 2.

JPA permet de mettre en correspondance le « monde objet » avec le monde relationnel. Une entité JPA est donc un objet Java (et ses relations) mis en correspondance via JPA avec la base de données sous-jacente. JPA est utilisé pour réaliser des opérations de recherche, sauvegarde, modification, suppression sur des objets et de répercuter celles-ci sur la base. Les entités JPA représentent le modèle objet des concepts du domaine d'étude à savoir les acteurs

Le SGDB utilisé par l'application est MySQL.



gc_comp.jpg

Le diagramme de composants ci-dessus vous présente les différents composants managés (beans CDI et EJB session beans) sur lesquels vous aller travailler.

Le rôle des composants est décrit dans le code source fourni.

Les composants stéréotypés <<CDI Bean>> sont des beans CDI. Les stéréotypes de scope <<XXXScoped>> précisent le scope de ces beans CDI, grossièrement le contexte pour lequel

il y a création d'une instance de bean CDI. Par exemple, pour CategoryBean, un scope de requête est spécifié. Ainsi pour chaque requête http, il y aura donc une instance de CategoryBean qui vivra le temps de la requête. Le container CDI crée et associe une instance pour chaque requête s'exécutant. Vous pourrez suivre le cycle des instances (création / destruction) grâce aux intercepteurs.

Les composants EJB sont stéréotypés <<EJB session bean>>. Les stéréotypes <<Stateless>> <<Stateful>> et <<Singleton>> précisent quel est le type de session bean. Une fois de plus notez qu'un composant managé peut être à la fois un bean CDI et un session bean. C'est le cas d'OrderBean et CategoriesCache. Le diagramme ne spécifie pas <<EJB session bean>> pour ces 2 composants, mais ça en est.

Notez aussi que tous les session beans Stateless sont exposés à leur client au travers d'une interface (vue) locale métier.

CurrentUser embarque l'entité JPA User représentant un utilisateur authentifié. Il y aura une instance par session http utilisateur. Une instance CurrentUser sera créée pour toute session http dans le cas où un utilisateur est authentifié. Ce composant n'est pas stéréotypé <<CDI bean>> car dans notre conception ce n'est pas obligatoirement un bean managé CDI. Il sera configuré puis injecté une fois l'authentification réussie. Le bean CDI CurrentUserFactory situé dans le paquetage com.goodcesi.model.factory n'est pas présenté dans le diagramme. C'est ce bean CDI qui fait office de fabrique produisant une instance CurrentUser configurée avant injection ou utilisation de celle-ci.

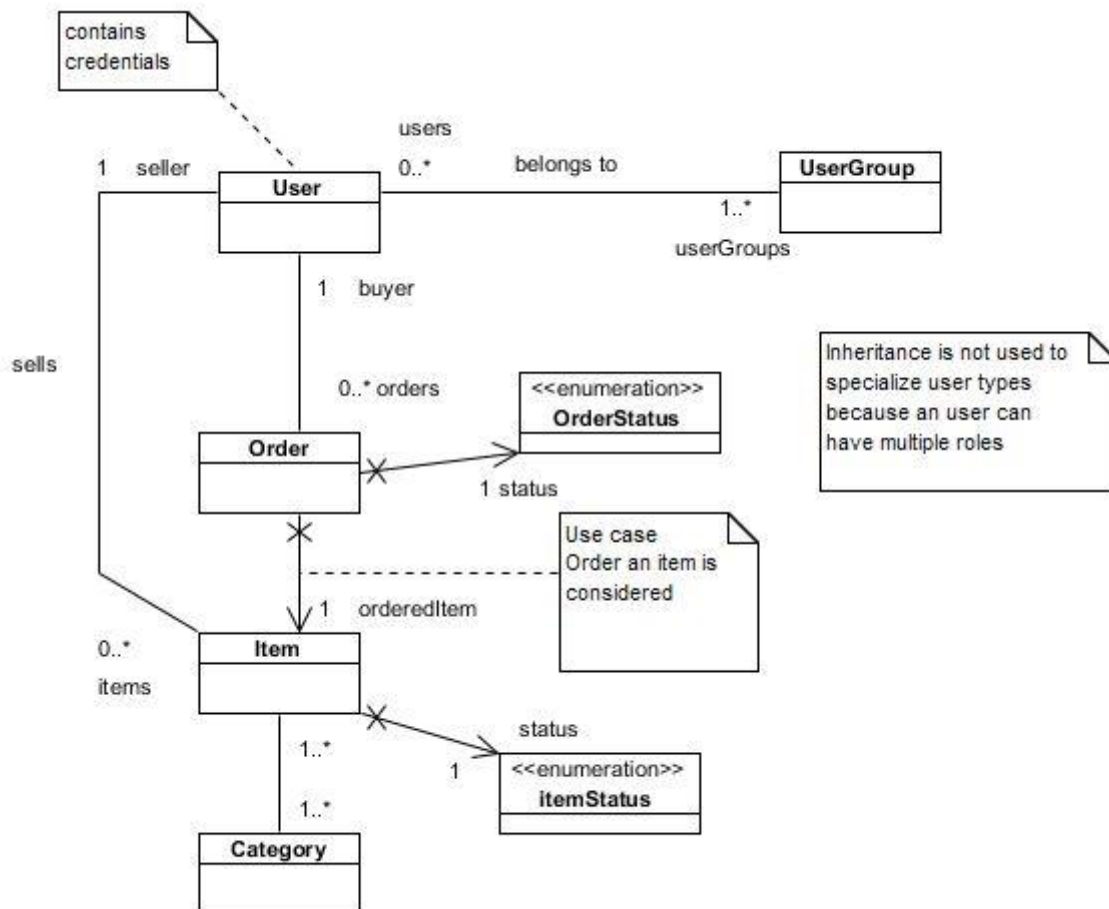


Ne confondez pas session http et donc SessionScoped avec session bean. Le « session » de session bean signifie juste interaction avec un client pendant un certain moment.

L'interaction entre les différents composants managés est mise en œuvre au travers du mécanisme d'injection de dépendance fourni par Java EE. Les relations de dépendances faibles marquées DI correspondent à l'injection de dépendance (DI = Dependency Injection).

Pour finir sur l'architecture de l'application, voici le modèle du domaine mis en relation avec la base de données correspondante. Ne perdez pas trop de temps sur JPA qui sera abordé dans le prochain proisit. Vous avez bien évidemment le droit d'être curieux et de vous intéresser à ce modèle pour mieux appréhender le sujet.

Comme il s'agit juste d'un prototype à but d'apprentissage, le modèle objet java aussi bien que le modèle relationnel de la base ont une conception simplifiée voire simpliste d'un domaine de vente de particulier à particulier.



gc_domain.jpg

Ces classes sont annotées avec des annotations de modélisation JPA.

Les crédeniels contenus dans **User** correspondent au login et mot de passe de l'utilisateur authentifié.

Comme expliqué plus haut un utilisateur peut être à la fois vendeur et acheteur d'où la relation plusieurs-plusieurs entre **User** et **UserGroup**.

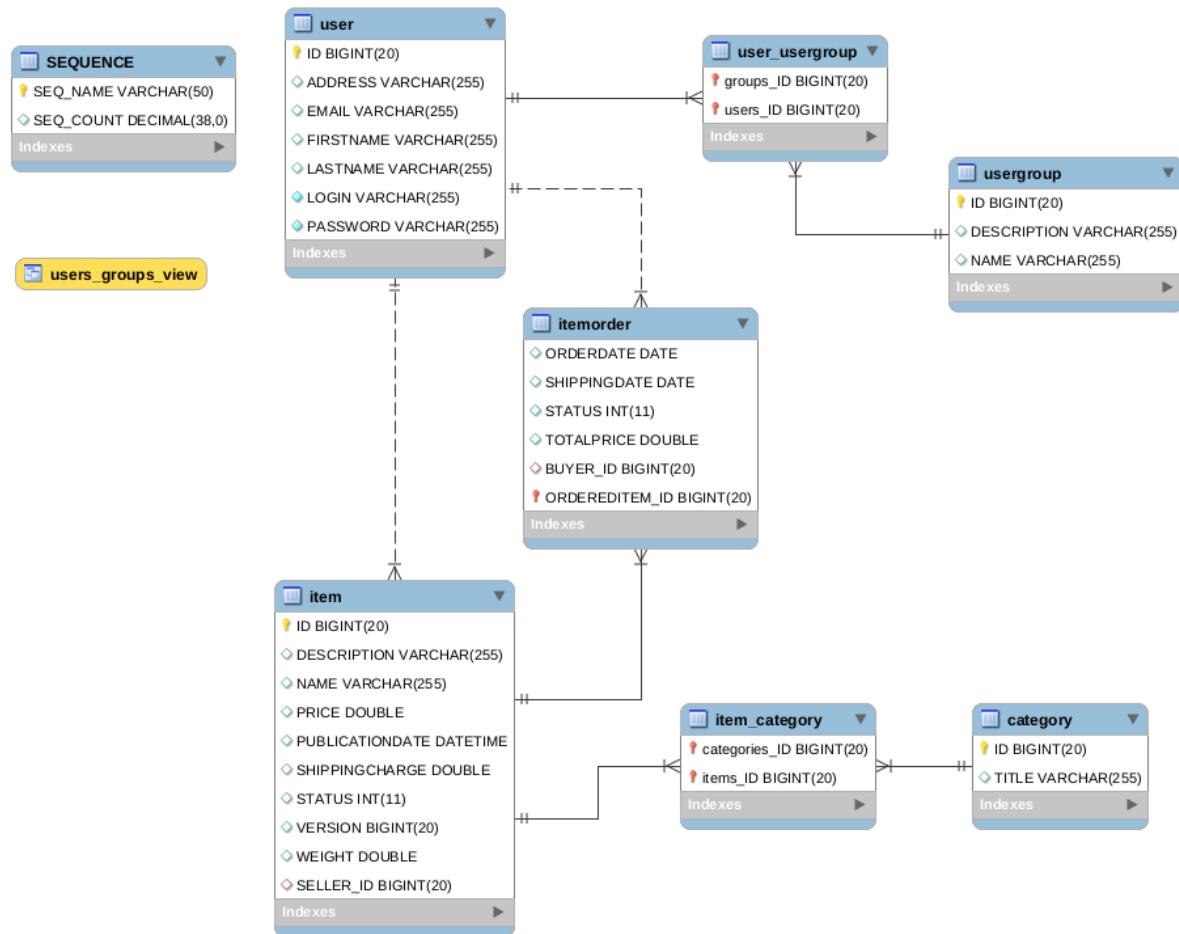
Notez aussi qu'un acheteur (buyer) peut passer plusieurs commandes (**Order**). Une commande ne correspondant qu'à un seul article (**Item**).

Les articles peuvent appartenir à plusieurs catégories.

Les crédeniels et les groupes utilisateur permettront de mettre en place la sécurité de l'application.

Le modèle entités-relations traduit globalement la même chose mais d'un point de vue base de données. La vue `users_groups_view` contient la liste des utilisateurs et leurs groupes. Elle est utilisée pour configurer la sécurité physique dans le domaine Payara hébergeant l'application goodcesi.

La séquence est un objet MySQL permettant de générer des clés primaires.



gc_eer.jpg

2. Préparation de l'application

Il s'agit ici de mettre en place la solution dans son environnement pour réaliser la corbeille.

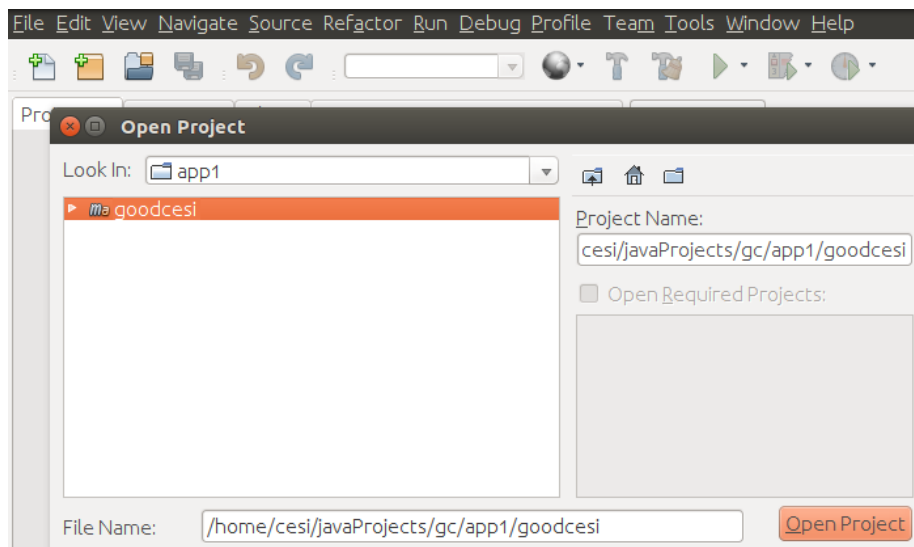
2.1. Génération de la base de données goodcesidb

Le dossier *script* contient le script *goodcesidb.sql* générant la base alimentée avec un jeu d'essai.

Exécuter le script depuis MySQL Workbench.

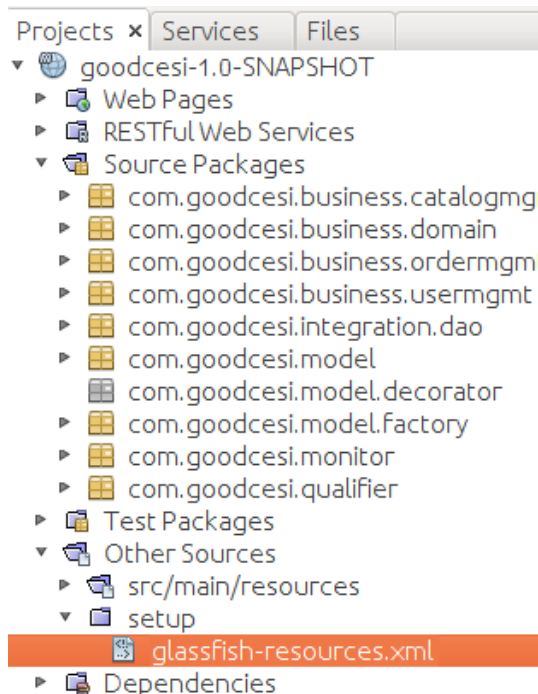
2.2. Ouverture du projet goodcesi depuis NetBeans

Dans le menu Files de NetBeans, sélectionnez Open Project... sélectionnez le projet maven de type Web Application localisé sous le dossier app1 fourni en ressource.



Le projet goodcesi-1.0-SNAPSHOT est désormais référencé dans l'onglet Projects de NetBeans.

goodcesi-1.0-SNAPSHOT contient le descripteur de déploiement glassfish-resources.xml qui définit la source de données JDBC pour connecter l'application à la base goodcesidb. Ce fichier se trouve dans Other Sources/setup.



Lors du premier déploiement de l'application sur le serveur la source de données jdbc/goodcesiDS et le pool de connexions (goodcesiPool) associé seront créés dans le domaine serveur. Pour plus d'informations, reportez-vous au document **PreparerEnvDevUbWin**.

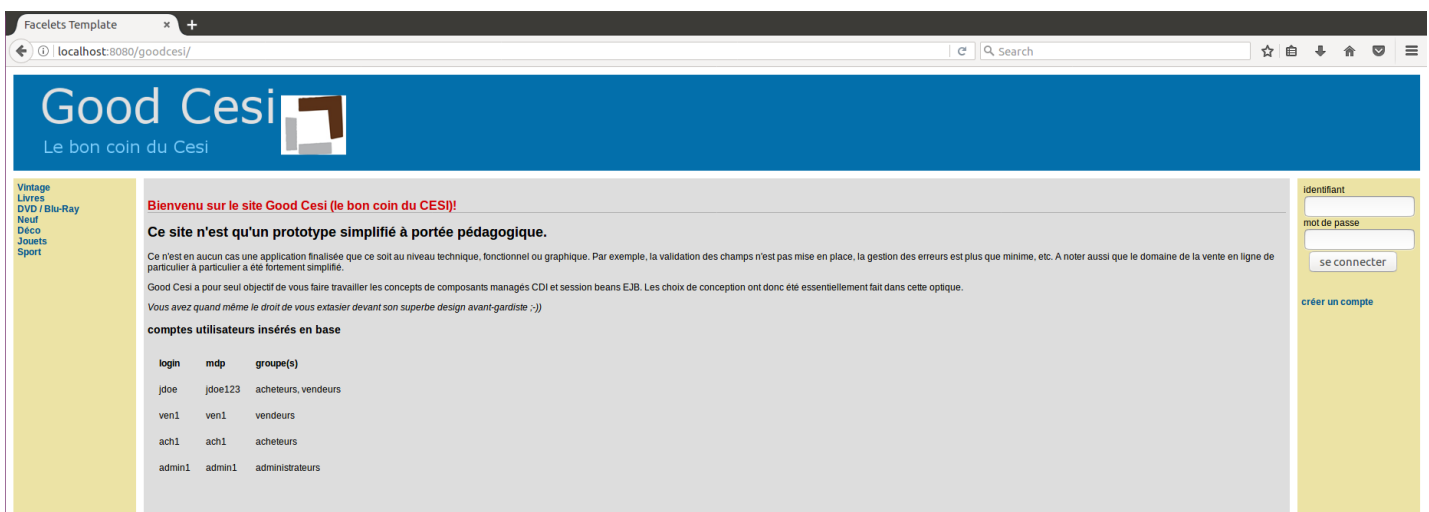
Dans ce fichier, mettez le mot de passe du compte root MySQL que vous avez défini. Si vous avez suivi le tutoriel d'installation de l'environnement ce mot de passe est **dbpwd**.

```
<jdbc-connection-pool allow-non-component-callers="false" associate-with
  <property name="URL" value="jdbc:mysql://localhost:3306/goodcesidb"/>
  <property name="User" value="root"/>
  <property name="Password" value="dbpwd"/>
  <property name="useSSL" value="false"/>
  <property name="serverTimezone" value="CET"/>
</jdbc-connection-pool>
```

Le code de l'application est commenté. De nombreux commentaires tiennent compte de la solution finalisée. C'est-à-dire qu'ils sont relatifs au code tel qu'il devrait être en fin de corbeille, une fois que vous aurez corrigé les erreurs. Ces commentaires devraient vous permettre de mieux appréhender la logique et vous guider dans la résolution des problèmes.

Cliquez du <droite> sur goodcesi-1.0-SNAPSHOT et exécutez Run.

Si Payara n'est pas démarré, alors NetBeans démarre le domaine serveur avant de déployer l'application et d'ouvrir la page d'accueil.



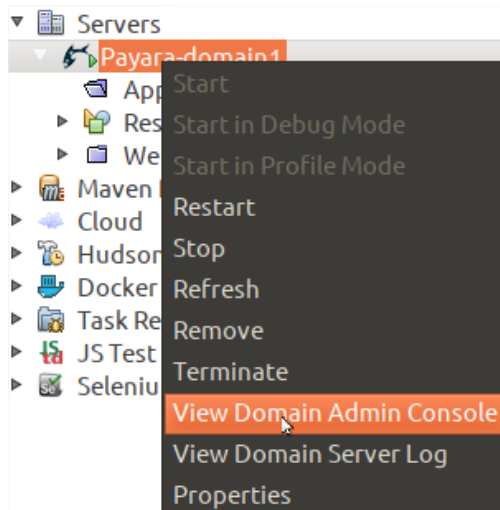
Si vous essayez de vous authentifier avec l'un des comptes indiqués, par exemple admin1/admin1, un message d'erreur s'affiche (*Erreur !! Veuillez vous authentifier*).

Notez que la liste des comptes présente dans la page d'accueil est un affichage statique recensant simplement les comptes initialement présents dans la base goodcesidb. Cette liste n'est donc pas mise à jour dynamiquement lorsqu'un nouveau compte est créé.

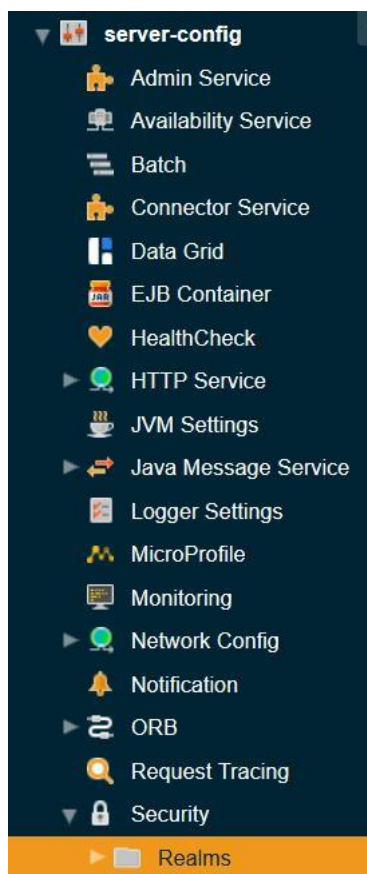
2.3. Configuration du domaine de sécurité Payara

Comme L'application s'appuie sur le service de sécurité fourni par Java EE, il faut donc configurer le domaine de sécurité au sein de votre serveur d'application Payara. Il ne vous est pas demandé de savoir sécuriser une application Java EE. Il faut par contre retenir que Java EE fournit des services de sécurité.

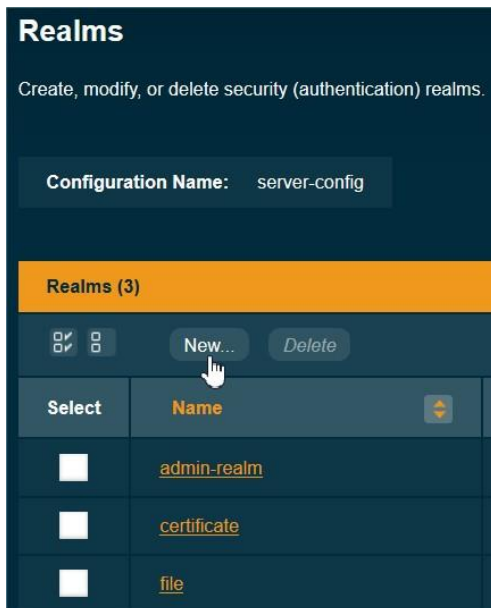
Pour cela ouvrez la console web d'administration du domaine (domain1) Payara. Vous pouvez l'ouvrir depuis l'onglet Services de NetBeans > Nœud Servers en cliquant du <droite> sur la référence du domaine domain1 et en sélectionnant View Domain Admin Console.



Dans la console web, naviguez dans le menu de gauche pour dérouler le nœud server-config > Security > Realms



Dans la page Realms, créez un nouveau Realm (domaine en français) nommé gcRealm.



gcRealm est le nom spécifié dans le fichier web.xml (Web Pages/WEB-INF) de l'application.

Il faut maintenant configurer gcRealm. Ce domaine de sécurité Payara est configuré pour que l'authentification s'appuie sur la base goodcesidb.

- Class Name : Sélectionnez `com.sun.enterprise.security.auth.jdbc.JDBCRealm` (on sécurise via une base par conséquent le realm est de type JDBC)
- Jaas Context : saisissez `jdbcRealm`
- JNDI : `jdbc/goodcesidS` (on utilise la source de donnée associée à goodcesidb)
- User Table : `users_groups_view` (l'authentification recherchera les utilisateurs inscrits dans la vue `users_groups_view`)
- User Name Column : `login` (on précise la colonne de la vue stockant les noms utilisateur)
- Password Column : `password` (nom de la colonne stockant les mots de passe)
- Group Table : `users_groups_view` (l'autorisation d'accès à des ressources sécurisées s'appuie sur des groupes stockés eux aussi dans la vue)
- Group Name Column : `groupname` (colonne de la vue stockant les groupes utilisateurs)
- Database user : `root` (compte utilisateur pour établir la connexion à la base)
- Database password : le mot de passe du compte utilisateur – c'est celui de votre compte root (`dbpwd`).
- Digest Algorithm : Le champ est laissé **vide** car on utilise l'algorithme par défaut SHA-256 (l'algorithme utilisé pour chiffrer le mot de passe lors de l'authentification afin de permettre la comparaison avec celui stocké en base).
- Charset : UTF-8 (le système de codage des caractères)

Name: *	gcRealm
Class Name:	<input checked="" type="radio"/> com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm <input type="radio"/> <input type="text"/>
Choose a realm class name from the drop-down list or specify a custom class	
Properties specific to this Class	
JAAS Context: *	jdbcRealm Identifier for the login module to use for this realm
JNDI: *	jdbc/goodcesiDS JNDI name of the JDBC resource used by this realm
User Table: *	users_group_view Name of the database table that contains the list of authorized users for this realm
User Name Column: *	login Name of the column in the user table that contains the list of user names
Password Column: *	password Name of the column in the user table that contains the user passwords
Group Table: *	users_groups_view Name of the database table that contains the list of groups for this realm
Group Table User Name Column:	<input type="text"/> Name of the column in the user group table that contains the list of groups for this realm
Group Name Column: *	groupname Name of the column in the group table that contains the list of group names
Assign Groups:	<input type="text"/> Comma-separated list of group names
Database User:	root Specify the database user name in the realm instead of the JDBC connection pool
Database Password:	dbpwd Specify the database password in the realm instead of the JDBC connection pool
Digest Algorithm:	<input type="text"/> Digest algorithm (default is SHA-256); note that the default was MD5 in GlassFish versions prior to 3.1
Encoding:	<input type="text"/> Encoding (allowed values are Hex and Base64)
Charset:	UTF-8 Character set for the digest algorithm

Cliquez sur OK en haut ou bas de page pour valider la création de gcRealm.

Vous pouvez désormais vous authentifier sur le site pour accéder aux espaces protégés.

Si vous n'avez pas réussi à faire cette manipulation. Vous pouvez utiliser la sauvegarde du domaine domain1 (dossier domainBckp) fournie en ressource. Cette sauvegarde vous permet de restaurer domain1 avec gcRealm configuré (la source de données jdbc/goodcesiDS est aussi configurée).

Pour lancer la restauration de domain1 en supposant que la sauvegarde se trouve dans le dossier /home/cesi/domainBckp :

```
asadmin restore-domain --backupdir /home/cesi/javaProjects/domainBckp domain1
```

Certaines versions de Payara tracent un grand nombre d'évènements lors du déploiement et lors de l'exécution d'une application. Une journalisation trop fine de certains évènements peut « polluer » la console de sortie NetBeans et vous rendre difficile l'accès aux informations qui vous intéressent. Cette corbeille fait souvent référence aux logs Payara affichés dans la console NetBeans. Par conséquent si vous avez trop de logs parasites, rendez-vous dans la console d'administration web de Payara et déroulez *Configurations | server-config | Logger Settings | onglet Log Levels*. Repérez les niveaux de journalisation positionnés sur **FINEST** et changez-les pour **INFO** ou **WARNING**. Par exemple, dans le cas de Payara 5.191, sélectionnez **INFO** pour le logger **java.util.logging.ConsoleHandler**. La sauvegarde de domain1 présentée ci-dessus, est configurée avec un niveau **INFO** pour ConsoleHandler.

Il est temps de passer aux exercices.

La première partie va vous faire travailler les services de base fournis par le standard CDI.

Elle est divisée en deux : une partie obligatoire et une partie optionnelle.

Les exercices optionnels sont consacrés à des concepts secondaires fournis par ce standard. Le caractère optionnel signifie que dans le cadre de ce module, il ne vous est pas demandé de savoir-faire, mais de quand même savoir que ces services existent.

La deuxième partie se focalise sur la mise en œuvre des session beans EJB. Cette partie est elle aussi divisée en deux. Les exercices obligatoires font le focus sur les caractéristiques essentielles des session beans. Les exercices optionnels comme précédemment vous permettront de travailler des concepts dont il faut avoir connaissance mais qu'il n'est obligatoire de creuser pour valider ce module.

Enfin *la troisième partie* est consacrée au standard JSF, standard de mise en œuvre de couche de présentation Web. JSF sera utilisé dans les 4 premiers prosits. Cette partie utilisera le projet goodcesi-2.0-SNAPSHOT situé dans *app2*.

Conseil : Réalisez en priorité les exercices obligatoires, puis la partie JSF. Enfin revenez aux exercices optionnels.

3. 1^{ère} partie - Contexts and Dependency Injection

3.1. Injection

Objectif : Utiliser l'injection de dépendance pour associer des composants managés

Introduction :

Les composants managés (beans CDI et session beans notamment) ne doivent pas être obtenus par instanciation. La raison principale est que ces composants (bien qu'ils aient les caractéristiques de base de simples objets java) ne sont pas utilisés pour simplement fournir des fonctionnalités implémentées dans leurs méthodes. Ces composants sont associés à des services tiers d'infrastructure (dit non fonctionnels) pour faciliter la mise en place d'application d'entreprise en plus d'implémenter des traitements. Ces services sont « associés » aux composants par un container qui s'interpose donc entre un client et une instance. C'est donc au container de gérer les instances et non au développeur de créer des instances.

- Lancez l'application (Run) & sélectionner une catégorie.

Une exception NullPointerException est levée. Cette exception traduit en gros que le container ne peut pas accéder au composant CatalogManager. C'est-à-dire que le container ne manage pas ce composant. Le composant n'est donc pas correctement référencé dans CatalogBean. Vous pouvez voir la pile d'appel avec la levée d'exception dans la console de sortie NetBeans du domaine Payara.

- Corrigez le problème pour que le bean CatalogManager de type CatalogManagerLocal soit correctement fourni au bean CDI CatalogBean.
- Sauvegardez les modifications du projet et lancez de nouveau le projet (Run). Les articles associés aux catégories seront désormais affichés si vous avez correctement corrigé.

➡ *Lors de la sauvegarde des modifications, NetBeans reconstruit automatiquement l'archive Web et la redéploie sur le serveur.*

➡ *Lors de la réalisation de cette corbeille, si le redéploiement automatique après sauvegarde ne semble pas prendre en compte vos modifications, faites un Clean & Build sur le projet puis un Run.*

3.2. Méthode productrice – fabrique d'instances contextuelles

Objectif : Utiliser une méthode productrice pour configurer des instances contextuelles avant injection.

- Authentifiez-vous avec jdoe/jdoe123 (vendeur & acheteur) et tentez d'accéder à l'espace vendeur

La console de sortie Payara indique qu'une exception est levée: currentUser.getUser() retourne une NullPointerException lors de l'invocation de : catalogManager.getItemsFromSeller(currentUser.getUser().getId()) car la propriété user de CurrentUser est nulle.

- De même si vous accédez à l'espace acheteur, le message de bienvenue n'indique pas l'identité de l'utilisateur.

Le message de bienvenue n'indique personne car `CurrentUser` n'est pas annoté avec `@Named`. Par conséquent `CurrentUser` ne peut être associé à la vue `buyer/buyerTemplate.xhtml` via le nom `authenticatedUser`.

```
</div>
<div id="content" class="left_content">
<h1>Bienvenue
#{authenticatedUser.user.firstname}#{authenticatedUser.user.lastname}</h1>
<ui:insert name="content">Content</ui:insert>
</div>
```

- Étudiez le code de `CurrentUser`. `CurrentUser` embarque l'entité `User` représentant l'utilisateur authentifié. La variable `User` n'a pas pu être instanciée par le container lors de l'injection.

`CurrentUser` représentant un utilisateur authentifié en session, Il faut donc pouvoir configurer `CurrentUser` de manière à ce que l'instance créée par le container :

- ✓ Embarque une instance `User` représentant l'utilisateur authentifié.
- ✓ Ait un scope de session. L'instance `CurrentUser` doit exister le temps de la session http.
- ✓ Ait un nom pour lier le bean aux vues JSF.
- ✓ Ait un qualificateur `@Authenticated` permettant de spécifier aux points d'injection de type `CurrentUser` qu'on injecte un bean correspondant à un utilisateur authentifié.
- Annotez `com.goodcesi.model.factory.CurrentUserFactory.createCurrentUser()` pour la transformer en méthode productrice
- Annotez la méthode pour que l'instance produite soit associée au contexte de la session http en cours. Autrement-dit, l'objet produit vivra toute la durée de la session utilisateur et ne sera visible qu'au sein de cette session.
- Annotez la méthode pour que l'instance contextuelle produite soit explicitement nommée `authenticatedUser` afin qu'elle soit référençable / utilisable dans les vues JSF via langage d'expressions unifié (`#{}).`
- Redéployez l'application pour tester de nouveau l'accès aux espaces.

Le redéploiement échoue. La console de sortie Payara indique une ambiguïté aux points d'injection. En effet, le container CDI ne sait pas quoi injecter (ambiguïté) aux points d'injection : doit-il injecter le bean annoté `@Dependent` ou le retour de la méthode productrice ?

- Créez un Qualificateur `@Authenticated` dans `com.cesi.qualifier` dans le but de documenter l'aspect utilisateur authentifié et de lever l'ambiguïté aux points d'injection.

Vous pouvez utiliser l'assistant NetBeans : cliquez du <droite> sur `com.goodcesi.qualifier` > New > Sélectionnez Other... > Sélectionnez *Qualifier Type* dans la catégorie Contexts and Dependency Injection

Remarquez l'annotation `@Qualifier` spécifiée sur le qualificateur que vous venez de créer.

- Annotez la méthode productrice avec le qualificateur `@Authenticated`
- Annotez les points d'injection de type `CurrentUser` avec `@Authenticated`. Reportez-vous au diagramme de composants pour déterminer quel bean utilise `CurrentUser`.

Attention SellerItemBean utilise une injection de constructeur. Il faut donc annoter le paramètre CurrentUser du constructeur.

- Sauvegardez l'application, relancez-la et testez de nouveau l'accès aux espaces protégés.

Vous pouvez désormais accéder aux informations dans les espaces protégés des vendeurs et des acheteurs. Vous pouvez même modifier un article existant dans l'espace vendeur.

Dans la console de sortie NetBeans de Payara l'invocation de la méthode productrice est tracée.

- Supprimez l'annotation @Dependent sur CurrentUser et relancez l'application après l'avoir sauvegardée.

Cela ne change rien car une méthode productrice sert aussi à produire des instances contextuelles qui ne sont pas issues de beans managés CDI. La classe CurrentUser n'étant plus annotée avec une annotation définissant un scope, elle n'est plus considérée par le container CDI comme un bean managé CDI.

Pour conclure cet exercice, revenons un peu sur le processus interne lié à la sécurité.

- 1) Concernant le système de sécurité :

Lorsqu'un utilisateur s'authentifie avec succès, c'est-à-dire qu'il fournit un login et un mot de passe valides (des « credentials » en anglais), le système d'authentification Java EE associe à la session http en cours un principal (représentation Java EE d'un utilisateur authentifié).

- 2) Concernant CurrentUser et la méthode productrice :

CurrentUser n'est utilisé qu'au sein d'espaces protégés. En effet CurrentUser est non seulement lié qu'à des pages JSF sécurisées (pages dans les dossiers buy et sell) mais n'est de surcroît invoqué que dans des beans utilisés au sein de ces pages sécurisées.

Par conséquent quand une page ou un bean a besoin d'une instance CurrentUser, le système d'authentification a déjà validé l'authentification. Aussi lorsque la méthode productrice est invoquée pour fournir une instance contextuelle, le contexte de sécurité contient le principal (ici le login). Dans la méthode productrice, on peut donc retrouver le principal via l'appel `FacesContext.getCurrentInstance().getExternalContext().getRemoteUser()` et utiliser ce dernier pour obtenir une instance d'User (`userManager.getRegisteredUser(« le principal »)`). On instancie enfin dans la méthode productrice CurrentUser avec l'utilisateur de type User avant de retourner l'objet contextuel.

- 3) Pour résumer :

La méthode productrice est invoquée par le container qu'une fois l'authentification passée. Elle peut donc fournir aux points d'injection ou aux vues JSF une instance contextuelle configurée avec les informations de l'utilisateur authentifié.



Pour les curieux, le système de sécurité de l'application est expliqué en annexe.

3.3. Scope des beans CDI

Objectif : Comprendre du concept de scope et donc d'objet contextuel

Vous devez définir les bons scopes pour les beans du paquetage **com.goodcesi.model** en vous appuyant sur le diagramme de composants.

Introduction :

Les beans CDI ont un scope (une portée). Cela signifie qu'il n'y a qu'une seule instance dans un contexte bien défini. C'est pour ça qu'on parle d'instance contextuelle, qu'on dit que les beans managés CDI sont contextuels. Les objets vivent donc dans un contexte d'exécution donné (requête http, session http...). De plus l'instance au sein du contexte accède à toutes les instances visibles dans ce contexte. Par exemple une instance ayant un scope de requête http pourra accéder à une instance de scope session, si la requête en question est exécutée au sein de la session. Inversement, une instance de session http pourra accéder aux instances de requêtes se déroulant au sein de la session.

C'est le container CDI qui est chargé de la gestion des objets contextuels, c'est-à-dire de la création des instances contextuelles, de leur maintien dans un contexte d'exécution donné et de leur destruction une fois le contexte terminé.

3.3.1. Scope de requête

Introduction :

Il faut configurer certains beans CDI de goodcesi pour que le container CDI « associe » à chaque requête http une instance de ces beans. En effet certains beans CDI exécutent des opérations ne nécessitant pas que les instances exécutant ces opérations vivent plus longtemps que le temps de la requête http.

L'avantage de limiter la vie d'une instance à la durée d'une requête http est l'aspect thread-safe. En effet comme une requête http = un thread, il n'y aura pas de risque que l'instance soit partagée par plusieurs threads. Les beans ayant un scope de requête sont donc thread-safe : le risque de corruption de l'état de l'instance est limité. Plus un bean reste en mémoire, plus il y a risque de corruption de ses données.

Notez que les JVM modernes et leurs ramasse-miettes sont optimisés pour que les cycles courts de construction /destruction d'objets (tels que ceux ayant une étendue de requête) aient peu d'impact en terme de performance. L'occupation de la mémoire par des objets peu utilisés a plus d'impact sur une application.

- Lancez l'application. Authentifiez-vous en jdoe/jdoe123 et rendez-vous sur l'espace acheteur et vendeur.

La liste des commandes n'est pas affichée dans l'espace acheteur ni dans l'espace vendeur. Il y a donc un problème avec OrderListBean.

- Si vous essayez de créer un compte, la liste des groupes n'est pas affichée dans le 3ème écran.

Il y a un problème avec GroupsBean.

- Connectez en tant qu'admin1/admin1 pour accéder à l'espace administrateur et tentez de créer une catégorie.

1 exception est levée indiquant que le bean `CategoryBean` n'est pas associé à la vue JSF.

- Pour résoudre les problèmes, annotez ces composants en vous basant sur les indications données dans diagramme des composants
- Les modifications enregistrées, redéployez l'application et testez l'affichage des commandes depuis l'espace acheteur ou vendeur, la création de compte et la création de catégorie pour vérifier que vous avez configuré comme il faut les composants posant problème.

3.3.2. Cycle de vie d'un bean CDI

Objectif : suivre le cycle de vie d'un bean managé.

Introduction :

Afin de mieux appréhender le cycle de vie d'un bean managé vous allez utiliser un intercepteur de cycle de vie pour tracer la construction et la destruction des instances de beans CDI. Il suffira de regarder les logs de la console de sortie NetBeans de Payara pour visualiser quand les instances sont construites et détruites par le container CDI.

- Annotez des classes de beans avec `@ScopeMonitor`. Annotez au minimum `CatalogBean`, `CategoriesCache` et `AccountBean`

Vous associez ainsi le bean à l'intercepteur `com.goodcesi.monitor.BeanScopeTracker` de cycle de vie. La méthode annotée `@PostConstruct` s'exécute lorsque l'instance est créée, la méthode annotée `@PreDestroy` s'exécute avant la destruction de l'instance.



Les intercepteurs sont approfondis dans la partie optionnelle.

- Relancez votre application et testez en naviguant dans les catégories. Pendant la navigation, suivez les logs de la console Payara.

Les instances de `CatalogBean` et `CategoriesCache` sont créées pour chaque requête (chaque clic) et ne vivent que le temps de celle-ci.

Une seule instance d'`AccountBean` est créée pour la durée de vie de l'application.

`CategoriesCache` est instanciée à chaque requête ce n'est donc pas un cache. `AccountBean` qui sert de point d'entrée dans le processus de création de compte n'a qu'une instance partagée pour toute l'application. Ce n'est pas vraiment ce que l'on désire. Ces beans n'ont sûrement pas le scope adapté.

3.3.3. Scope de session

Introduction :

`AccountBean` intervient, comme dit précédemment, dans le processus de création de compte. Il doit donc y avoir une instance d'`AccountBean` pour chaque processus de création de compte. Le processus de création de compte est fortement lié à la session http.

AccountBean ayant pour l'instant un scope d'application, la même instance sera utilisée par plusieurs internautes créant simultanément un compte. Ce qui veut dire que l'état de l'instance, représenté par ses propriétés, sera partagé par tous les utilisateurs créant un compte. Ce n'est évidemment pas la situation désirée. Il faut donc confiner chaque instance à la session http en cours.

- Modifiez AccountBean en vous basant sur le diagramme des composants pour que chaque instance de ce bean managé ne dépasse pas les bornes de la session http.
- Sauvegardez et réexécutez votre application puis testez la création d'un compte. Suivez les logs de Payara pour visualiser quand l'instance d'AccountBean est créée et quand elle est détruite.

L'instance ne vit que le temps du processus de création de compte.

- Optionnellement, vous pouvez tester la création deux 2 comptes en parallèle en ouvrant 2 navigateur.

Vous verrez dans les logs Payara que chaque navigateur (et donc chaque session) a une instance associée.

3.3.4. Scope d'application

Introduction :

Jusqu'ici une instance de CategoriesCache est créée pour chaque requête http. Or ça ne sert à rien de charger depuis la base de données la liste des catégories à chaque requête. En effet des nouvelles catégories sont rarement créées. CategoriesCache jouant le rôle de cache des catégories, l'application n'a besoin que d'une seule instance.

- En vous aidant du diagramme de composants, modifiez CategoriesCache pour qu'il y ait une seule instance dans l'application.
- Sauvegardez et réexécutez votre application puis testez en naviguant dans les catégories. Suivez les logs Payara dans la console de sortie de NetBeans pour vous assurer qu'il n'y a qu'une seule instance de CategoriesCache pour la durée de l'application.

Remarque :

Il ne faut pas abuser des singletons car plus une instance vie longtemps, plus ses données (son état) risquent de ne plus être à jour, c'est-à-dire de ne plus refléter la réalité des données.

CategoriesCache\$Proxy\$_\$\$_WeldSubclass vous donne aussi un indice sur le fonctionnement interne du container CDI. Le container CDI crée une sous-classe du type du bean qui fait office de proxy. Un point d'injection CDI référence donc un proxy et non directement l'instance. Par conséquent lorsque vous invoquez une méthode sur une variable injectée c'est le proxy qui entre en jeux et qui se chargera de déléguer l'invocation au container qui invoquera au final la méthode sur l'instance managée.

Dans le cas d'une instanciation classique une variable référence une instance.

3.3.5. Scope de conversation

Introduction :

Pour les beans ayant un scope de session, il y a une seule instance pour toute la durée de la session. C'est le cas d'OrderBean dans votre application. Aussi, si un utilisateur commande par exemple deux articles dans deux onglets distincts, il y a des risques qu'au moment de payer dans un onglet, il paie le prix de l'article en cours de commande dans l'autre onglet. Il ne sera pas heureux de payer 100 € un article à 2 balles.

Il faut donc limiter la vie d'un objet contextuel au contexte d'un onglet et plus précisément à un nombre d'étapes au sein de cet onglet. C'est pour cela que l'on peut définir des beans avec un scope de conversation, beans pour lesquels une instance par onglet sera créée. C'est le concept de conversation qui va démarquer la durée de vie et la portée de ce type de bean. Il faut donc que le développeur indique au sein de la classe du bean quand la conversation débute (quand le bean est promu en tant que composant de longue durée) et quand l'instance doit être détruite (quand la conversation se termine). Une instance contextuelle de scope conversationnel vit donc le temps d'un certain nombre d'étapes au sein d'un seul onglet.

- Annotez OrderBean avec @ScopeMonitor pour suivre le cycle de vie d'une instance.
- Modifiez OrderBean pour lui attribuer un scope de conversation (vous pouvez vous reporter au diagramme de composants).
- Injectez un objet de type javax.enterprise.context.Conversation.
- Implémentez le démarrage et l'arrêt de la conversation dans le bean.
 - ✓ La conversation « longue durée » est démarrée au début de beginOrdering().
 - ✓ La conversation « longue durée » est terminée à la fin de pay().
- Redéployez votre application et testez la commande d'un article. Vous pouvez aussi tester la commande de 2 articles dans 2 onglets différents. Pensez à regarder dans la console de sortie NetBeans les traces du cycle de vie pour OrderBean.

Vous en avez fini avec les exercices obligatoires relatifs à CDI. Les exercices suivants de cette partie CDI sont facultatifs. Ces exercices étant facultatifs, les codes à mettre en œuvre sont présentés.

3.4. Gestion des événements

Objectif : utiliser le service de gestion d'évènements CDI pour notifier un changement.

Introduction :

CategoriesCache est un cache applicatif. Problème, si vous créez une nouvelle catégorie, la liste des catégories du site n'est pas mise à jour. Cela est dû au fait que le bean est instancié lors du lancement du site, vu que la liste des catégories est présente dans la page d'accueil. La méthode @PostConstruct chargée de récupérer la liste des catégories est exécutée après instanciation du bean.

Pour rafraîchir le cache on va donc utiliser la gestion des événements. Lorsque CategoryBean créera une nouvelle catégorie un événement sera déclenché pour notifier CategoriesCache

qu'une nouvelle Catégorie a été ajoutée. CategoryBean est donc source d'évènements et le cache est un observateur de ce type d'évènements. CDI fournit donc une implémentation du pattern Observateur.

- Créez dans `com.goodcesi.qualifier` le qualificateur `@CategoryAdded`

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface CategoryAdded {}
```

- Injectez dans CategoryBean un évènement de type paramétrique Category. Annotez aussi ce point d'injection avec `@CategoryAdded` pour limiter la gestion des évènements.

```
@Inject @CategoryAdded private Event<Category> catEvent;
```

Attention, Event appartient à `javax.enterprise.event`.

- ✓ Dans CategoryBean, modifiez la méthode `createNewCategory(ActionEvent ae)` de la façon suivante :
Si `catalogManager.saveNewCategory(title)` réussit l'opération de sauvegarde, alors on peut déclencher l'évènement notifiant l'ajout d'une catégorie.

```
@Inject
private Event<Category> catEvent;

public void createNewCategory(javax.faces.event.ActionEvent ae){
    category = catalogManager.saveNewCategory(title);

    //si la catégorie a été sauvegardée
    if(category!=null){
        catEvent.fire(category); //on déclenche l'évènement
    }
}
```

Notez qu'`ActionEvent` est par contre un évènement JSF (`javax.faces.event`) et non un évènement CDI.

Pour information, cette méthode est invoquée par le moteur JSF depuis la vue `admin/categoryCreation.xhtml` quand on clique sur le bouton Créer.

```
<h:form>
<h:outputLabel value="titre"/>&nbsp;<h:inputText id="catTitleIn"
required="true" value="#{categoryModel.title}"/>
<br/>
<h:commandButton value="créer"
actionListener="#{categoryModel.createNewCategory}"/>
</h:form>
```

- Implémentez dans CategoriesCache la méthode observatrice :

Le qualificateur `@CategoryAdded` spécifie que seuls les évènements « annotés » `@CategoryAdded` seront gérés / pris en charge par la méthode observatrice.

```
public void addCategory(@Observes @CategoryAdded Category cat){
    categories.add(cat);
}
```

- Redéployez et testez l'application : connectez-vous en `admin1/admin1`, puis créez une nouvelle catégorie. Revenez à la page d'accueil pour visualiser le menu contenant la nouvelle catégorie.

3.5. Intercepteurs

Objectif : mise en place d'intercepteurs pour monitorer l'invocation des méthodes métier.

Introduction :

On a utilisé des intercepteurs pour journaliser/suivre le cycle de vie des beans. C'est-à-dire un code qui s'insère de façon transparente dans le cycle de vie d'un objet. Par transparent, on entend que vous n'avez pas eu à ajouter du code dans les classes des beans.

Vous allez créer un intercepteur pour suivre l'invocation des méthodes « métier » publiques. Ce type d'intercepteur s'insère dans l'invocation d'une méthode métier. Cela évitera de polluer le code de vos méthodes métiers avec des `System.out.println()`.

Les intercepteurs de méthode métier s'interposent entre le client et la cible. Ils interceptent donc l'invocation avant que l'objet cible exécute sa méthode. Ils enrobent l'invocation.

Les composants sous `com.goodcesi.business` sont pour l'instant annotés `@Transactional`.

Cette annotation lie le composant ou une de ses méthodes à un intercepteur standard CDI défini par le système de transaction Java EE (Java Transaction API) afin que chaque invocation à une méthode publique s'exécute dans une transaction.

Les intercepteurs Java EE permettent de mettre en œuvre ce qu'on appelle la programmation orientée aspect : « Embarquer dans des aspects du code redondant devant s'exécuter en plusieurs points / endroits de l'application. Et greffer ces aspects dans le programme sans polluer le code métier de celui-ci ».

- Créer une annotation `@MethodInvocationMonitor` de liaison d'intercepteur dans le paquetage `com.goodcesi.qualifier`. Vous pouvez utiliser l'assistant NetBeans :

<Clic-droite> sur le paquetage > New > Other ...> Contexts and dependency Injection > Interceptor Binding Type > Nommez le `MethodInvocationMonitor`.

Notez que cette annotation est associable à une classe ou une méthode. Notez aussi la présence de l'annotation `@interceptorBinding`.

- Créez l'intercepteur dans `com.goodcesi.monitor`. Pour cela :
 - ✓ Créez la classe `BeanInvocationTracker`
 - ✓ Annotez la classe avec `javax.interceptor.Interceptor` pour qu'elle soit considérée comme étant un intercepteur.
 - ✓ Annotez la classe de l'intercepteur avec `@MethodInvocationMonitor`
 - ✓ La classe doit implémenter `Serializable` pour que l'intercepteur puisse être associé aux bean ayant la capacité de passivation.
 - ✓ Annotez la classe du bean avec `@javax.annotation.Priority` (`Interceptor.Priority.APPLICATION+1`) pour activer cet intercepteur.

La priorité (nombre entier) entre parenthèse précise que cet intercepteur est déclenché après les intercepteurs du container et permet de spécifier l'ordre des intercepteurs si plusieurs intercepteurs sont chaînés.

- ✓ Créez et implémentez la méthode `public Object traceMethodCall(InvocationContext) throws Exception` et annotez-la avec `@javax.interceptor.AroundInvoke`. Cette Annotation déclare que l'intercepteur « s'exécute autour » d'une méthode métier.

javax.interceptor.InvocationContext représente le contexte d'invocation. Il permet d'accéder entre autre à l'objet cible, intercepté. Mais il permet surtout via la méthode `proceed()` d'invoquer l'intercepteur suivant dans la chaîne s'il y en a un, et en dernier d'invoquer la méthode ciblée.

```
@Interceptor
@MethodInvocationMonitor
@Priority(Interceptor.Priority.APPLICATION+1)
public class BeanInvocationTracker implements Serializable{

    @AroundInvoke
    public Object traceMethodCall(InvocationContext ctx) throws
Exception{
        String businessMethodName = ctx.getMethod().getName();
        String targetBeanName =
ctx.getTarget().getClass().getSimpleName();
        System.out.println("****méthode
"+targetBeanName+"#"+businessMethodName+ " invoquée****");
        return ctx.proceed(); //la méthode métier va être invoquée
    }
}
```

- Annotez des méthodes publiques de beans ou directement des classes de beans avec `@MethodInvocationMonitor` pour associer l'intercepteur. Vous pouvez par exemple annoter les classes `CatalogBean`, `OrderBean`.

Annotez la classe si vous voulez journaliser toutes les méthodes publiques ou annotez simplement les méthodes que vous voulez monitorer.

- Redéployez l'application et testez-la pour visualiser dans la console NetBeans de votre domaine Payara les traces des méthodes invoquées.

Si vous avez annoté des classes, vous devriez notamment voir des getters invoqués. Lors de l'affichage des vues (rendu de la réponse), les getters des beans CDI sont invoqués par le moteur JSF pour peupler les pages avec les données.

3.6. Les décorateurs

Objectif : mise en place de décorateurs pour enrichir la logique métier. Dans votre cas chiffrer/crypter un mot de passe avant l'insertion en base d'un nouveau compte utilisateur.

Introduction :

Pour l'instant le chiffrement du mot de passe est stocké dans la méthode privée `AccountBean.encrypt`. Il serait préférable de séparer ce code du traitement de création d'un compte. L'utilisation du pattern Décorateur est faite pour ça. CDI fournit une API pour faciliter la mise en œuvre de ce pattern. Les décorateurs enrichissent les méthodes métiers en implémentant une interface commune avec le bean intercepté. Par contre ils ne sont pas faits pour gérer des responsabilités transversales. Donc même si les décorateurs sont similaires aux intercepteurs du fait qu'ils interceptent des méthodes métiers, ces 2 concepts n'ont pas le

même but. D'ailleurs les décorateurs n'utilisent pas de type de liaison pour associer une méthode à un intercepteur.

- Supprimez l'instruction de chiffrement du mot de passe dans la méthode `createUser` d'`AccountBean` :

```
pwd= encrypt(pwd);
```

- Créez dans `com.goodcesi.model.decorator` une classe abstraite `PasswordEncoder` implémentant `AccountModel` et `Serializable`.

On utilise une classe abstraite car on veut simplement implémenter la méthode `AccountModel.createUser()`.

- Annotez la classe avec `@javax.decorator.Decorator` et `@Priority(Interceptor.Priority.APPLICATION + 10)`

@Priority est utilisée pour activer le décorateur. Elle permet aussi d'ordonner l'exécution des décorateurs (ou des intercepteurs) si plusieurs décorateurs (ou intercepteurs) sont présents.

- Injectez le type décoré `AccountModel` au point d'injection délégué. `AccountModel` implémenté par `AccountBean` est l'un des types du bean décoré `AccountBean`.

```
@Inject @Delegate private AccountModel accountBean;
```

@Delegate appartient au paquetage `javax.decorator`.

- Déplacez la méthode privée `AccountBean.encrypt` dans le décorateur `PasswordEncoder`.
- Implémentez la méthode `createUser` dans le décorateur `PasswordEncoder`.

```
public String createUser() {
    String pwd = accountBean.getPwd(); //récupération du mot de passe non chiffré
    pwd=encrypt(pwd); //on chiffre le mot de passe
    accountBean.setPwd(pwd); //on assigne le mot de passe chiffré
    return accountBean.createUser();
}
```

- Après avoir enregistré les modifications, exécutez de nouveau l'application et testez la création d'un compte. Vérifiez que le mot de passe inséré en base dans la table `user` et est bien chiffré. Vous pouvez aussi tester la validité de création du compte en vous authentifiant avec.

4. 2^{ème} partie - Session beans (EJB)

4.1. Session beans Stateless (SLSB)

Objectif : transformer les composants de la couche logique métier et d'intégration en SLSB

Introduction :

Ces couches sont critiques, il faut donc pouvoir bénéficier d'un support des transactions par défaut, du pooling pour la scalabilité, de la capacité à sécuriser facilement, voire la possibilité d'accéder à distance à cette couche, etc... Le modèle CDI léger et pourtant puissant n'est peut-être pas le plus adapté.

- Dans les paquetages `com.goodcesi.business.*`, Supprimer les annotations `@Transactional` et `@Dependent`

Pour démarquer une transaction avec CDI, il faut explicitement annoter les méthodes ou les classes avec `@Transactional`. Par contre les EJB et donc les session beans sont transactionnels par défaut. Cela signifie que dans le modèle EJB il n'y a pas besoin d'annotations pour que toute méthode métier s'exécute au sein d'une transaction. C'est le principe de conventions plutôt que configuration. Quand vous choisissez le modèle EJB c'est que vous avez sûrement besoin d'une couche applicative transactionnelle. Si vous désirez tuner le comportement transactionnel de vos composants EJB, vous avez entre autre des annotations qui le permettent.

Le comportement transactionnel par défaut pour une méthode EJB est le suivant :

- ✓ *Si un client (par exemple un bean CDI pour lequel `@Transactional` n'est pas spécifié) invoque un session bean dans un contexte non transactionnel, le container démarre une transaction avant d'exécuter la méthode du session bean.*
- ✓ *Si un client (par exemple un session bean) invoque un session bean dans un contexte transactionnel, alors la méthode du session bean invoqué s'exécute au sein de la transaction du client.*

`@Dependent` aurait pu être conservée car cette annotation définissant un bean CDI est compatible avec les SLSB. C'est la seule annotation définissant un scope qu'on peut appliquer à un session bean stateless.

- Vous pouvez aussi Supprimer l'implémentation de `Serializable`.

Les SLSB, même s'ils supportent `@Dependent` (pseudo-scope) ne sont pas des objets contextuels, par conséquent, il n'y a pas de dépendance avec le cycle de vie du client. Donc même si le bean CDI client invoquant un SLSB doit être « passivation capable », le SLSB n'a pas à se soucier de ça.

Les session beans stateful que vous allez voir plus loin intègrent un mécanisme de passivation qui ne nécessite pas l'implémentation de `Serializable` pour activer cette capacité.

- En vous appuyant sur le diagramme des composants, annotez les classes des paquetages ayant pour racine `com.goodcesi.business` afin de définir des session beans Stateless.
- Sauvegardez et réexécutez l'application puis testez que votre couche logique métier implémentée avec des session beans Stateless est fonctionnelle. Vous pouvez par exemple naviguer dans les catégories d'articles, commander un article, créer une nouvelle catégorie, modifier un article depuis l'espace vendeur etc.
- Supprimez les annotations `@TransactionScoped` et `Serializable` de `CrudService` situé dans le paquetage `com.goodcesi.integration.dao`.

L'annotation `@TransactionScoped` est une annotation CDI définie par Java Transaction API (JTA). Le scope d'un bean annoté avec cette annotation est la transaction. Il y a donc une instance créée pour chaque transaction. L'instance étant détruite en fin de transaction.

Toute méthode d'un SLSB de `com.goodcesi.business.` invoquant `CrudService` s'exécute dans une transaction. Pour chaque exécution de méthode il y aura donc création d'une*

instance de ce bean CrudService. Cette instance ne vivra que le temps de la transaction en cours et sera associée au contexte transactionnel du session bean appelant.

Notez aussi les méthodes « JPA » de modification doivent s'exécuter au sein d'une transaction. Java Persistence API sera le thème du prochain prosit. Ce n'est donc pas la peine de vous y attarder.

- Transformez CrudService en session bean stateless.
- Comme précédemment, testez les fonctionnalités de votre application pour vous assurer que la couche d'intégration est correctement configurée. En termes de fonctionnalité vous ne verrez pas de changements.
- Utilisez une autre annotation qu'`@Inject` pour injecter la vue locale CrudServiceLocal du session bean CrudService dans le session bean OrderManager.
- Redéployez l'application et naviguez dans le catalogue des articles pour vérifier que vous avez utilisé la bonne annotation pour injecter CrudService. Là encore vous ne verrez pas de changements au niveau fonctionnel car vous avez simplement utilisé une autre annotation pour injecter un session bean.

4.1.1. Vue locale d'un session bean

Introduction :

Les interfaces métier de vos session beans définissent une vue locale. Cela signifie que vos session beans ne sont accessibles qu'au sein de la même application (ici une ARchive Web – WAR) et doivent être accédés au travers de cette interface métier. Aucune annotation sur l'interface n'est nécessaire pour spécifier que les beans sont accessibles au travers d'une vue locale de type interface métier. Cependant pour faciliter la lecture votre code à des tiers, vous pouvez explicitement spécifier sur l'interface du session bean que c'est une vue locale.

- Annotez les interfaces métiers pour explicitement déclarer que sont des vues locales de SLSB. Ce n'est pas la peine d'annoter toutes les interfaces mais annotez-en au moins une.
- Testez votre application pour vérifier que vous avez positionné la bonne annotation sur l'interface métier.

4.1.2. Cycle de vie d'un session bean stateless

Objectif : Suivre le cycle de vie – et visualiser la mise en pool des instances d'un session bean Stateless.

- Afin de suivre le cycle de vie d'un SLSB, commencez par annoter CatalogManager avec `@javax.interceptor.Interceptors(com.goodcesi.monitor.BeanScopeTracker.class)`.

C'est la technique pour associer un intercepteur à un EJB. (L'autre méthode vue dans la partie intercepteurs marche aussi pour Payara). Un intercepteur « pour EJB » n'a pas besoin d'être défini avec `@Interceptor`.

Pour rappel, la mise en œuvre des intercepteurs est un objectif optionnel. Vous devez cependant savoir que le container EJB gère des pool d'instances de session beans Stateless.

Vous allez suivre le cycle de vie de CatalogManager car c'est le SLSB le plus fréquemment utilisé dans goodcesi.

- Redéployez l'application et parcourez le catalogue des articles pour visualiser dans les logs de votre domaine Payara, la construction des instances de ce SLSB CatalogManager.

Une seule instance est créée. Vous aurait-on menti ? Le container EJB ne crée donc pas un pool d'instance pour servir de façon concurrente un grand nombre de client ?

En fait, le container s'adapte à la charge de l'application. Vu que l'instance n'est pas trop sollicitée, il n'augmente pas le nombre d'instance du pool. Si la charge devenait importante le nombre d'instance mises en pool augmenterait.

- Vous allez forcer le container EJB à créer un pool avec un nombre minimum initial d'instances.
- Ouvrez la Console Web d'administration de Payara (<http://localhost:4848>) > server config > EJB Container > Pool Settings > champ *Initial et Minimum Pool Size* > mettez par exemple le nombre de 15 > relancez (Run) l'application.



Dans la console de sortie NetBeans pour Payara vous remarquerez que le container EJB a créé plusieurs instances de CatalogManager.

Même si cela est difficilement perceptible, sachez que les instances sont construites lors de la première utilisation.

- Le test effectué, remettez la valeur 0 pour éviter une occupation ici inutile de la mémoire.

4.2. Session beans Stateful (SFSB)

Objectif : fournir la capacité session bean Stateful au bean CDI OrderBean

Introduction :

OrderBean est critique en termes d'accès concurrents. Il ne faudrait pas que des requêtes simultanées même au sein d'un même onglet risquent de polluer les données (il y a peu de risques), qu'un utilisateur non autorisé (n'ayant pas le rôle SELLER) puisse accéder au paiement... En terme de transactions c'est aussi critique : la plupart des méthodes doivent réussir ou échouer d'un bloc, les données manipulées doivent être « isolées », etc. OrderBean étant le point d'entrée dans le processus de paiement le modèle EJB @Stateful est donc utilisé pour intégrer des services que ne fournit pas par défaut le modèle CDI.

- Ajoutez l'annotation pour qu'OrderBean soit aussi considéré comme un session bean Stateful.

Les SFSB sont les seuls types d'EJB à supporter tous les scopes CDI.

- Un SFSB à un cycle de vie intégrant naturellement la passivation vous pouvez supprimer aussi l'implémentation de l'interface marqueur Serializable
- Annotez aussi le composant pour spécifier explicitement que la seule vue disponible est une vue locale sans interface.

Depuis Java EE 6 un session bean peut être exposé au travers d'une vue locale sans interface.

- Optionnellement vous pouvez associer un intercepteur de cycle de vie à OrderBean. Redéployez l'application et testez le processus de commande si vous le désirez mais à ce niveau il n'y aura pas de différence fonctionnelle si vous avez correctement configuré OrderBean.
- Pour conclure sur les session beans Stateful prenez connaissance du commentaire avant la définition de la classe OrderBean en ce qui concerne l'annotation @Remove.

4.3. Session beans Singleton

Objectif : Mettre en place un singleton pour mettre en œuvre un mécanisme déclaratif (par annotation) de gestion de la concurrence.

Introduction :

Le cache est déjà « un singleton » par contre il n'est pas thread-safe. La liste des catégories pourrait être simultanément lue pendant qu'elle est mise à jour.

De plus le fait d'ajouter la capacité Session Bean Singleton permettra de déclarer un service de temps dans ce composant (exercice optionnel).

- Annotez le bean CDI CategoriesCache pour qu'il soit aussi considéré comme un session bean singleton (ATTENTION : le packaging n'est pas javax.inject).

Un EJB Singleton peut être annoté @ApplicationScoped ou @Dependent.

Désormais toutes les méthodes métiers sont verrouillées en écriture Aucune méthode ne peut s'exécuter de façon concurrente. Il n'y a plus de risque d'accès concurrent.

- Là encore relancez votre application pour vérifier que vous avez annoté correctement CategoriesCache. Fonctionnellement il n'y aura pas de changement de changement perceptible.

Vous en avez fini avec les exercices obligatoires relatifs aux session beans. Les exercices suivants de cette partie session beans sont facultatifs. Ces exercices étant facultatifs, les codes à mettre en œuvre sont présentés. Ces exercices vont vous présenter un échantillon relativement complet des services offerts par le modèle EJB.

4.3.1. Définir explicitement le mode de verrouillage des méthodes d'un Singleton

Introduction :

C'est bien, le cache n'est plus accessible de façon concurrente par des clients (ici des vues JSF). Par contre cela pose un problème de performance car l'exécution des méthodes se fait de façon sérialisable : Lorsque plusieurs utilisateurs accèdent au site simultanément et donc aux catégories d'article, la méthode `getCategories()` devrait donc être invoquée simultanément sur l'instance par JSF. Du fait du mode de verrouillage, `getCategories()` ne peut pas être invoquée de façon concurrente. Cela crée un goulot d'étranglement et réduit donc les performances.

En fait il faut que la méthode `getCategories()` puisse être invoquée de façon concurrente mais pas la méthode `addCategories()`. Par contre, pendant qu'`addCategories()` s'exécute pour ajouter une catégorie à la liste en cache aucune autre méthode doit être invoquée sur l'instance du singleton.

Comme dit plus haut, par défaut toutes les méthodes métier d'un singleton sont verrouillées en écriture (mode WRITE). Si vous préférez un verrouillage non exclusif, vous pouvez spécifier explicitement un verrou READ. Là encore c'est le principe Convention Over Configuration : on part du principe que pour une instance accessible de façon concurrente, la protection, la consistance des données est plus importante que les performances. On peut ensuite décider de spécifier un autre verrou partagé (READ) pour autoriser un accès non sérialisable (c'est-à-dire simultané).

- Annotez la méthode `getCategories()` de `CategoriesCache` avec `@javax.ejb.Lock(LockType.READ)` pour positionner un verrou de lecture (verrou partagé). Sur cette méthode. Profitez-en pour annoter la classe `CategoriesCache` avec `@javax.ejb.Lock(LockType.WRITE)` dans un but simplement documentaire.

Le mode d'accès concurrent a donc été redéfini uniquement pour la méthode métier `getCategories()`.

```
...//autres annotations sur la classe
@Lock(LockType.WRITE)//mode de verrouillage par défaut
public class CategoriesCache {
    ...
    //redéfinition du mode de verrouillage par défaut.
    //la méthode peut être invoquée de façon concurrente
    @Lock(LockType.READ)
    public List<Category> getCategories() {
        return categories;
    }
    ...
}
```

- Une fois de plus relancez votre application pour vérifier que vous avez annoté correctement `CategoriesCache`.

4.4. Méthode asynchrone

Objectif : exécuter une méthode de manière asynchrone pour éviter que le client soit bloqué en attendant la fin de l'exécution de la méthode.

Introduction :

La création d'une commande peut être une opération longue, il faut valider la carte bleue, donc interroger des banques.... Il ne faut pas bloquer le client durant cette opération.

Il faudrait donc que la méthode `OrderManager.createOrder` s'exécute de façon asynchrone. C'est-à-dire qu'elle ne s'exécute pas dans le thread client.

- Dans la méthode du session bean `OrderManager.createOrder`, décommentez le bloc de code simulant un traitement long de 10 secondes.

```
public void createOrder(Order order) {
    dao.create(order);
    Item i = order.getOrderedItem();
    dao.update(i);
    //ne pas utiliser l'API java.lang.Thread dans un environnement Java EE. Ici
    //c'est pour simuler un traitement long et visualiser l'exécution asynchrone
    try {
        System.out.println("démarrage d'un traitement long de paiement");
        Thread.sleep(10000L);
        System.out.println("fin du traitement de paiement");
    } catch (InterruptedException ex) {
        Logger.getLogger(OrderManager.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

- Redéployez l'application et commander un article.

Lorsque vous effectuez le paiement, vous êtes bloqué pendant 10 secondes et encore ce n'est qu'un test. Pas terrible en termes d'expérience utilisateur.

👉 Si vous avez déjà épuisé tous les articles du catalogue, relancez le script sql.

- Convertissez la méthode `OrderManager.createOrder` en méthode asynchrone. Il suffit d'annoter la méthode avec `@javax.ejb.Asynchronous`.

```
@Asynchronous
public void createOrder(Order order){...}
```

- La modification enregistrée, lancez l'application (Run) et testez de nouveau la commande d'un produit.

La réponse est affichée de façon quasi-instantanée. Par contre dans les logs du serveur vous voyez le traitement long asynchrone qui s'exécute alors que l'utilisateur a repris la main.

4.5. Service d'horloge

Objectif : mettre en place des rappels temporels qui permettront l'exécution de méthodes à intervalle de temps régulier dans notre cas.

4.5.1. Horloge automatique

Introduction :

Le cache de catégorie est mis à jour automatiquement quand une nouvelle catégorie est créée. Cependant si une catégorie est supprimée ou le nom modifié, la mise à jour de la liste ne sera pas reflétée. Pour simplifier, disons que si vous supprimez une catégorie dans la base ou que vous en changez le nom, ces mises à jour ne seront pas répercutées dans le logiciel.

Pour pallier à cela, vous allez mettre en place dans le singleton CategoriesCache un rappel temporel dit automatique qui mettra à jour à intervalle de temps régulier la liste des catégories. Ce service de rappel temporel invoquera toutes les 15 secondes (pour le test) CatalogManager.getAllCategories().

- Dans CategoriesCache, implémentez la méthode refreshCategories et annotez-la avec @javax.ejb.Schedule.

```
@Schedule(hour = "*", minute = "*", second="*/15")
public void refreshCategories(){
    System.out.println("timer");
    categories = catalogManager.getAllCategories();
}
```

@Schedule permet de déclarer un timer qui sera inscrit automatiquement dans le service d'horloge lors du déploiement de l'application. Toutes les 15 secondes cette horloge envoie un signal au container EJB pour qu'il exécute la méthode sur laquelle le timer a été déclaré.

Les services d'horloge ne peuvent être utilisés que dans les session beans Stateless et les Singletons.

Notez que la méthode a un verrou exclusif car @Lock(LockType.WRITE) est spécifiée sur la classe CategoriesCache. Par conséquent lors de son exécution aucune autre méthode ne s'exécutera.

- Ouvrez dans MySQL WorkBench la table category
- Lancez l'application.
- Modifiez dans la table category le nom d'une catégorie
- Naviguez dans les catégories du site.

Le délai écoulé, vous verrez dans la page la modification dans le menu des catégories. Dans la console Payara, vous verrez timer s'afficher toutes les 15 secondes.

4.5.2. Horloges programmées

Introduction :

Lorsqu'une commande a été validée, le vendeur se doit d'envoyer la commande. Pour le suivi, il doit marquer la commande comme envoyée (SENT) dans son espace vendeur.

Lors de la création d'une commande, il faut associer un timer qui se déclenchera à intervalle régulier tant que la commande n'aura pas le statut SENT. Ce rappel temporel permettrait

d'envoyer « un mail » au vendeur pour lui rappeler qu'il a une commande non envoyée. On se contentera d'un affichage de message dans la console de sortie.

- Dans le session bean `OrderManager`, injectez un service d'horloge permettant de créer des timers (`javax.ejb.TimerService`)

```
@Resource//injection d'un service d'horloge
private TimerService timerService;
```

@javax.annotation.Resource est une annotation permettant d'injecter des ressources du type timer, mais aussi des sources de données (javax.sql.DataSource) pour des manipulations avec JDBC, et pleins d'autres services... Cette annotation ne permet cependant pas d'injecter un bean CDI ou un session bean.

- En fin de méthode `OrderManager.createOrder`, créez le timer.

```
//le service d'horloge sera invoqué une première fois 30 sec après sa
//création puis toutes les 30 sec.
timerService.createTimer(30000L, 30000L, order.getOrderedItem().getId());
```

Ce timer embarque comme information l'id de l'article commandé.

Le timer est enregistré dans le service d'horloge et se déclenchera une première fois 30 secondes après sa création puis toutes les 30 secondes.

- Toujours dans `OrderManager`, créez la méthode `checkOrderStatus` annotée `@javax.ejb.Timeout` qui sera déclenchée / invoquée par le container EJB chaque fois que le délai sera écoulé. Cette méthode a un paramètre de type `javax.ejb.Timer` représentant le timer créé.

```
@Timeout
void checkOrderStatus(Timer timer){
    Long id = (Long) timer.getInfo();//on récupère l'id stocké comme info dans le
    //timer
    Order retrievedOrder = dao.find(Order.class, id);//récupération de la commande.
    if(retrievedOrder.getStatus()==OrderStatus.SENT){//si la commande a été
    //envoyée
        timer.cancel();//on annule les rappels temporels
    }else{//sinon le système envoie un "mail" au vendeur.
        //ici on se contente d'un affichage console
        System.out.println("Bonjour
        "+retrievedOrder.getOrderedItem().getSeller().getFirstname()+
        " n'oubliez pas d'envoyer la commande
        "+retrievedOrder.getOrderedItem().getName());
    }
}
```

Cette méthode envoie un « mail » au vendeur si la commande pour laquelle le timer a été créé n'est pas marquée comme envoyée. Si la commande est envoyée. Le rappel de temps est annulé.

- Après avoir enregistré les modifications, exécutez l'application puis effectuez une commande.

Regardez les logs Payara pour voir le message destiné au vendeur qui s'affiche toutes les 30 secondes.

- Identifiez-vous avec un compte vendeur et marquez la commande comme envoyée.

Tant que le vendeur n'a pas marqué SENT la commande, alors un message s'affiche dans la console de NetBeans.

4.6. Gestion des transactions

4.6.1. Attributs transactionnels

Objectif : Modifier le comportement transactionnel par défaut des EJB.

Introduction :

Les justifications concernant les choix transactionnels fait ci-après peuvent être discutées. L'idée est de vous faire travailler sur les attributs transactionnels. Sans compter que le choix des attributs dépend bien évidemment des choix de conception.

L'invocation systématique de méthodes au sein de transactions peut avoir un effet sur les performances. L'entrée en action du service transactionnel n'est pas une petite affaire.

Quand une méthode n'est qu'en lecture, c-à-dire qu'elle ne modifie aucune donnée, ce n'est pas obligatoirement la peine que cette méthode s'exécute dans une transaction, surtout si celle-ci est invoquée par un client non transactionnel.

Pour rappel, il vous est demandé de connaître que le comportement transactionnel par défaut des session beans. Ce comportement transactionnel correspond au mode REQUIRED.

Pour spécifier explicitement le comportement transactionnel des méthodes d'un EJB, il faut utiliser l'annotation `@javax.ejb.TransactionAttribute`.

- Annotez la classe du session bean `ItemsSeeker` (paquetage `com.goodcesi.business.catalogmgmt`) pour que ses méthodes, chargées de retourner des listes d'articles, ne s'exécutent jamais au sein d'une transaction.

```
...
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class ItemsSeeker implements ItemsSeekerLocal {...}
```

On veut s'assurer que toute méthode du gestionnaire de commande, s'exécute au sein d'une transaction démarrée spécialement pour la méthode. L'idée est de s'assurer que ce session bean critique s'exécute toujours au sein d'une nouvelle transaction, et ce, que le client invoquant ce session bean soit transactionnel ou pas.

- Annotez `OrderManager` (paquetage `com.goodcesi.business.ordermgmt`) pour spécifier qu'une nouvelle transaction est démarrée pour chaque méthode invoquée quel que soit le comportement transactionnel du client d'`OrderManager`.

```
...
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public class OrderManager implements OrderManagerLocal {...}
```

- Cependant la méthode `retrieveOrders` ne fait que retourner une liste. Annotez donc la méthode `OrderManager.retrieveOrders` pour redéfinir le comportement transactionnel de cette méthode afin que celle-ci ne s'exécute qu'au sein d'une transaction démarrée par le client invoquant la méthode. Si le client n'est pas transactionnel, le container n'a pas à associer une transaction à cette méthode.

```
@Override
@Transactional(TransactionalAttributeType.SUPPORTS)
public List<Order> retrieveOrders(User user, CallerContext context) {...}
```

Une méthode `@Timeout` est invoquée par le container EJB. Par conséquent si on ne spécifie pas volontairement `NOT_SUPPORTED` sur cette méthode, celle-ci s'exécutera toujours au sein d'une nouvelle transaction. Elle aura toujours le comportement `REQUIRES_NEW`, même si aucun attribut n'est spécifié ou si `REQUIRED` l'est explicitement.

Dans un même ordre d'idée, une méthode asynchrone s'exécute dans son propre thread. Le client ayant invoqué une méthode asynchrone aura peut-être retourné quand la méthode asynchrone achèvera ses opérations. La méthode asynchrone ne peut donc pas joindre la transaction cliente. On est dans un cas similaire aux méthodes `@Timeout` ou seuls les attributs `REQUIRED` (défaut) / `REQUIRES_NEW` et `NOT_SUPPORTED` peuvent être spécifiés.

Introduction :

Le session bean `CrudService` (`com.goodcesi.business.integration.dao`) dans la couche intégration doit obligatoirement s'exécuter au sein d'une transaction démarrée par l'appelant.

Sauf dans le cas de la méthode `findAll` qui retourne potentiellement un grand nombre d'objets, par exemple l'ensemble des articles sauvegardés en base. Déjà que ce type d'opération de lecture peut être assez couteux, ce n'est pas la peine de faire entrer en plus le système transactionnel dans la partie. Il faut donc que toute transaction éventuellement en cours soit interrompue le temps que `findAll` s'exécute.

Il est vrai que ce type d'implémentation retournant tout d'un bloc n'est pas une bonne pratique.

- Annotez la classe `CrudService` et sa méthode `findAll` pour répondre à ces exigences.

```
...
@Transactional(TransactionalAttributeType.MANDATORY)
public class CrudService implements CrudServiceLocal{
...
@Override
@Transactional(TransactionalAttributeType.NOT_SUPPORTED)
public <T> List<T> findAll(Class<T> type){...}
...
}
```

Enfin `CategoriesCache` est un Singleton ayant le rôle de cache applicatif, son but est juste de cacher des données stockées en base, en aucun cas de les modifier. Il ne faut donc pas que ces méthodes puissent s'exécuter au sein d'une transaction. Autrement dit toute transaction invoquant ce bean doit échouée.

- Annotez `CategoriesCache` pour spécifier ce comportement.

```
...
@Transactional(TransactionAttributeType.NEVER)
public class CategoriesCache{...}
```

La méthode `@PostConstruct` n'hérite pas de l'attribut spécifié sur la classe. De plus sur la méthode `@PostConstruct NEVER` n'a pas de sens car la méthode n'est jamais invoquée par un client. Cette méthode de callback supporte `REQUIRES_NEW` / `REQUIRED` et `NOT_SUPPORTED`. Donc si on applique `NEVER` sur la classe, la seule façon d'empêcher la méthode `@PostConstruct` de s'exécuter dans une transaction est de spécifier explicitement `NOT_SUPPORTED` sur cette méthode.

- Testez votre application. Vous ne verrez pas de différence d'un point de vue fonctionnel.
- Vous pouvez tester aussi le comportement d'une méthode annotée `MANDATORY` lorsque l'invocation a lieu au sein d'un contexte non transactionnel. Pour cela :

- ✓ Injectez le DAO `CrudService` dans `CacheCategories`.

```
@Inject CrudServiceLocal crudService
```

- ✓ Invoquez dans la méthode `CategoriesCache.getAllCategories` la méthode `find` de `CrudService`.

```
...
public List<Category> getCategories() {
    crudService.find(Category.class, 1L);
    return categories;
}
```

- Exécutez votre application.

Dans la console Netbeans des logs Payara et dans la page d'accueil, il est indiqué que le container a levé l'exception `EJBTransactionRequiredException`. Cette exception indique que `find` doit être invoquée au sein d'une transaction.

- Le test effectué, supprimez l'invocation et l'injection.

4.6.2. Exceptions

Objectif : Tester l'impact d'une exception déclenchée au sein d'une transaction

Introduction :

Les exceptions ont une incidence sur l'issue d'une transaction.

Les exceptions d'application (pour simplifier les exceptions contrôlées ou les exceptions étendant `RuntimeException` annotées avec `@javax.ejb.ApplicationException`) n'annulent pas la transaction en cours lorsqu'elles sont déclenchées au sein d'une méthode transactionnelle.

Les exceptions système (en gros les exceptions dérivées de `RuntimeException` non annotées `@ApplicationException`) entraîne automatiquement l'avortement de la transaction lorsqu'elles sont déclenchées au sein d'une méthode transactionnelle. La seule issue pour la transaction est le rollback (annulation). Toutes les opérations effectuées au sein de cette transaction seront annulées.

- Dans `CatalogManager.saveNewItem`, déclenchez une exception Système (de type `RuntimeException`) en fin de méthode. Commentez le retour de la méthode.


```
@Override
public Item saveNewItem(Item item, User owner, Set<Long> categories) {
    ...
    throw new RuntimeException();
    // return managedItem;
}
```

- Exécutez l'application et testez la création d'un article. Pour cela connectez-vous avec un compte vendeur (par exemple, jdoe ou ven1).

Les logs Payara indiquent que le container a intercepté l'exception levée et a renvoyé au client une exception de type `javax.ejb.EJBException`.

Dans MySQL workbench, aucune donnée n'a été sauvegardée en base. La transaction démarrée par le container au sein de laquelle l'exception a été levée est annulée (rollback). Par conséquent la modification (ici l'insertion d'un article) n'est pas appliquée.

Le container a aussi détruit l'instance du session bean, celle-ci étant considérée comme « polluée ». En effet les exceptions système servent à notifier un problème d'infrastructure difficilement récupérable.

Une application d'exception notifie un comportement inattendu au niveau applicatif. C'est pour cela que la levée de ce type d'exception au sein d'une transaction n'entraîne pas par défaut l'annulation de cette transaction. C'est au développeur de décider si la transaction doit être annulée ou pas.

- Supprimez le déclenchement de l'exception et décommentez le retour.

4.7. Sécurité des EJB

Objectif : sécuriser l'accès aux session beans en utilisant le modèle de sécurité déclaratif des EJB.

Introduction :

L'accès aux ressources web est sécurisé. On peut pousser encore plus loin la sécurité en sécurisant aussi l'accès aux session beans de la couche métier.

En effet si un développeur de l'équipe injecte par exemple un session bean dans un espace « non sécurisé », alors sans sécurité associée au session bean, une faille est créée. De plus si certains session beans étaient exposés à des applications distantes, un man in the middle pour accéder à un session bean alors qu'il n'est pas autorisé... Les session beans représentant souvent des briques critiques dans l'application, il faut autant que possible gérer les autorisations d'exécution.

EJB fournit un modèle de sécurité déclaratif s'appuyant sur les annotations. Ce système permet d'autoriser l'accès aux méthodes en fonction des rôles déclarés.

L'annotation la plus utilisée est `@javax.annotation.security.RolesAllowed({« role1 », « role2 », ...})` qui peut être spécifiée sur une classe ou sur une méthode. Lorsqu'une méthode est invoquée le container examine le principal (identité de l'appelant authentifié) associé à l'invocation. Il contacte le service de sécurité pour savoir si le principal a un rôle l'autorisant à exécuter la méthode. Si c'est le cas le container permet l'exécution de la méthode (l'invocation est dispatchée à la méthode métier). Sinon l'invocation est rejetée.

Pour rappel les rôles logiques de sécurité sont définis dans le fichier web.xml et mappés avec des groupes physiques dans glassfish-web.xml (ou payara-web.xml).

Vous allez donc sécuriser quelques méthodes.

- Annotez la méthode `OrderManager.createOrder` avec `@RolesAllowed("BUYER")` pour que seuls les acheteurs puissent créer une commande.

```
@Override
@RolesAllowed("BUYER")
@Asynchronous
public void createOrder(Order order){...}
```

- De la même manière annotez `OrderManager.retrieveOrders` avec `@RolesAllowed({"BUYER","SELLER"})` pour que les vendeurs et acheteurs soient les seuls autorisés à récupérer une liste de commande.

```
@Override
@Transactional(TransactionalTypeType.SUPPORTS)
@RolesAllowed({"BUYER","SELLER"})
public List<Order> retrieveOrders(User user,CallerContext context) {...}
```

- Annotez `OrderManager.putOrderIntoSentState` pour n'autoriser que les vendeurs à accéder à cette méthode permettant de donner le statut « envoyé » à une commande.

Vous devriez savoir comment faire.

- Annotez aussi la méthode `CatalogManager.saveNewCategory` pour que seul un utilisateur ayant le rôle ADMIN soit autorisé à créer une nouvelle catégorie.

Vous devriez savoir comment faire.

- Enfin annotez la méthode `pay()` d'`OrderBean` pour autoriser uniquement les acheteurs à exécuter un paiement.

Pour rappel OrderBean est un bean CDI et un session bean Stateful. Il peut donc utiliser la sécurité déclarative EJB.

Vous devriez savoir comment faire.

- Testez que l'application fonctionne normalement.
- Pour mettre en évidence le système de sécurité, invoquez la méthode sécurisée `CatalogManager.saveNewItem` accessible uniquement aux administrateurs dans `ItemBean.initConversation`.

```
public String initConversation(){
    catalogManager.saveNewCategory("test");
    if(conversation.isTransient()){
        conversation.begin();
    }
    return "/seller/itemCreation";
}
```

- Connectez-vous avec un utilisateur ayant le rôle de vendeur et essayez de créer un nouvel article. La méthode `initConversation` est invoquée quand vous cliquez sur Ajouter un article depuis l'espace vendeur.

Dans la vue JSF et dans les logs Payara, vous pouvez voir qu'une exception de type `javax.ejb.EJBAccessException` est levée. Elle indique que l'utilisateur ayant un rôle `SELLER` n'est pas autorisé à invoquer `saveNewCategory` qui nécessite d'avoir le rôle `ADMIN`.

- Une fois le test effectué, vous pouvez supprimer cette invocation de test.

5. 3^{ème} partie - JSF

Maintenant que vous avez vu comment implémenter des couches logicielles avec les standards EJB & CDI, vous allez aborder comment associer vos vues JSF avec les beans CDI dont le rôle dans `goodcesi` est essentiellement de lier les vues et la logique métier.

Ouvrez le projet `goodcesi-2.0-SNAPSHOT`

Ce projet est une version corrigée de `goodcesi`. Dans ce projet les services de rappels temporels ont été « commentés » pour ne pas polluer le suivi le cycle de vie JSF.

Lancez éventuellement le script sql pour recréer la base de données.

5.1. Utilisation du langage d'expressions unifié

Objectif : associer une bean CDI à une vue afin de déclencher des actions et pour afficher des données

Introduction :

Pour qu'un bean CDI (voire un session bean) puisse être associé à une vue, il faut tout d'abord lui attribuer un nom afin que le moteur JSF puisse faire la liaison.

- Exécutez l'application `goodcesi-2.0-SNAPSHOT` et cliquez sur une catégorie.

Lorsque vous cliquez sur une catégorie il ne se passe rien. Aucune action n'est déclenchée.

- Assigner le nom `catalogModel` au bean `CatalogBean`

Vous pouvez regarder les autres beans pour savoir comment faire.

- Vous devez ensuite associer la méthode d'action `public String retrieveItemsFromCategory()` du bean `CatalogBean` à la balise JSF `commandLink` de la vue `catMenu.xhtml`

Une méthode d'action à la signature suivante : `public String <nom_méthode>([paramètre])`

Depuis JSF 2, une méthode d'action peut accepter un paramètre.

Les méthodes d'action retournent une chaîne qui permet de déclencher la navigation vers une autre vue.

Pour associer une action à une balise JSF, il faut utiliser le langage d'expressions unifié : `#{}.`

- Prenez le temps de regarder l'implémentation de la méthode `retrieveItemsFromCategory()`
- Lancez un Build puis déployez l'application via Run. Restez l'accès aux catégories.

Vous naviguez bien mais vers une page vide (vue itemList). Aucune valeur n'est associée à la vue itemList.xhtml.

- Associez la liste des articles de la catégorie sélectionnée à la balise JSF dataTable d'itemList.

dataTable est un composant JSF qui a pour valeur une liste en vue d'afficher des éléments de cette liste. Il faut donc utiliser la balise value de dataTable pour lier grâce au langage d'expressions la liste.

- Exécutez l'application (Run).

Vous pouvez désormais naviguer dans les catégories d'articles.



Si vous avez un message d'erreur PropertyNotFoundException lorsque vous accédez aux articles, lancez un Build puis de nouveau Run

5.2. Utilisation de balises de composant JSF

Objectif : mettre en place des vues JSF en utilisant des balises de composants JSF.

5.2.1. Création d'un formulaire

- Créez dans la vue personnalInfos.xhtml, le formulaire JSF permettant de saisir les informations personnelles lors de la création d'un compte utilisateur.
 - ✓ Ce formulaire est associé au bean CatalogBean nommé catalogModel.
 - ✓ L'utilisateur doit saisir dans des champs textes (inputText) son nom, son prénom et son email
 - ✓ Et dans une zone de texte (textArea) son adresse.
 - ✓ Le nom saisi est associé à la propriété *lname* du bean.
 - ✓ Le prénom saisi est associé à la propriété *fname* du bean.
 - ✓ L'email est associé à la propriété *email* du bean.
 - ✓ L'adresse est associée à la propriété *address* du bean.
 - ✓ Un bouton permet de passer à l'écran suivant groupsSelection. On est dans un cas de navigation statique similaire au passage de la vue credInfos vers la vue personnalInfos.

Voici le résultat attendu : c'est le formulaire que vous avez rencontré dans la version goodcesi-1.0-SNAPSHOT

Saisissez vos informations personnelles

prénom


nom

adresse

email

- Exécutez l'application et créez un nouveau compte pour vérifier l'implémentation de votre formulaire de saisie des informations personnelles.

5.2.2. Création d'une dataTable

- Dans la vue buyer/orderList.xhtml, créez la dataTable listant l'ensemble des commandes de l'acheteur authentifié.
 - ✓ La liste affiche simplement la date de commande (orderDate) au format dd/MM/YYYY, le nom de l'article (name) et le nom du vendeur (lastname).
 - ✓ La dataTable utilise le bean OrderListBean pour accéder à la liste des commandes (orders).
-  *NetBeans va indiquer un warning spécifiant que la propriété lastname de seller est introuvable. C'est juste l'IDE qui ne repère pas une propriété au-delà d'un certain niveau de profondeur de navigation dans une relation.*
- Exécutez l'application pour visualiser la liste des commandes connectez-vous en tant qu'acheteur (jdoe par exemple) puis accédez à l'espace acheteur.

5.3. Cycle de vie JSF

Objectif : comprendre le cycle de vie d'une requête JSF

Introduction :

Quand les getters et setters sont-ils invoqués, quand les actions sont-elles déclenchées ?

La création d'un bean (si nécessaire), l'invocation des setters des propriétés de ce bean, de ses getters et des méthodes d'action ont lieu dans des phases bien spécifiques du cycle de vie d'une requête JSF.

➤ Vous allez suivre le cycle de vie d'une requête JSF. Pour cela :

- ✓ Activez le listener en décommentant l'élément **lifecycle** dans le fichier WEB-INF/faces-config.xml.

Ce fichier permet entre autre de spécifier les règles de navigation entre vues au lieu de les établir directement dans les méthodes d'action. Les prochains prosits utiliseront ce système.

Le listener LifecycleTracker écrit dans la console de sortie NetBeans du domaine Payara le nom de la phase avant chaque phase du cycle de vie JSF. En fin de requête, quand la réponse est envoyée au client, ce listener affiche un message indiquant la fin de la requête avec un n° pour mieux suivre. Le listener se trouve dans le paquetage com.goodcesi.monitor.

- ✓ Associez CatalogBean aux intercepteurs BeanScopeTracker et BeanInvocationTracker, en annotant la classe avec respectivement @ScopeMonitor et @MethodInvocationMonitor afin de suivre le cycle de vie et l'invocation des méthodes d'une instance.
- Redéployez l'application goodcesi-2.0-SNAPSHOT et naviguez dans les catégories d'articles. Pendant que vous naviguez, regardez les logs de Payara pour suivre le cycle de vie JSF.
- Répondez aux questions suivantes :
 - ✓ Quelles sont les phases indiquées lors de la première requête (requête initiale JSF) ? Autrement dit lors du premier accès à la page d'accueil ?
 - ✓ Lorsque vous cliquez sur une catégorie, quelles sont les phases indiquées ?
 - ✓ Dans quelle phase est invoquée la méthode d'action retrievalItemFromCategory ?
 - ✓ Dans quelle phase est invoqué le getter getItemsFromCategory ?

- ✓ Si vous sélectionnez un article, dans quelle phase sont invoquées les setters ?

- ✓ Dans quelles phases sont invoqués les getters ?

Vous trouverez en annexe une explication des différentes phases du cycle de vie d'une requête JSF.

Annexes

1. Système de sécurité de goodcesi

La sécurité physique

En début de document vous avez configuré dans le domaine domain1 un domaine de sécurité (realm) s'appuyant sur la base de données goodcesidb. Cette base stocke les comptes utilisateurs ainsi que les groupes auxquels les utilisateurs appartiennent. Le système de sécurité physique est donc mis en place au sein d'un serveur, dans notre cas au sein d'un domaine Payara.

La plateforme Java EE fait la distinction entre sécurité physique configurée dans un domaine Payara et sécurité logique définie au sein d'une application. Cette distinction permet de changer de système de sécurité physique sans impacter le code de l'application.

La sécurité logique

On définit donc la sécurité logique au sein d'une application. Pour cela on va utiliser le descripteur standard WEB-INF/web.xml

Dans ce fichier, on définit des contraintes de sécurité indiquant quelles ressources web ont un accès protégé. L'extrait ci-dessous spécifie que toutes les vues JSF et autres pages web situées sous le dossier **seller** sont accessibles uniquement aux utilisateurs associé au rôle logique de sécurité **SELLER**

```
<security-constraint>
  <display-name>selling constraint</display-name>
  <web-resource-collection>
    <web-resource-name>seller resources</web-resource-name>
    <description/>
    <url-pattern>/faces/seller/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <description>for authorized sellers</description>
    <role-name>SELLER</role-name>
  </auth-constraint>
</security-constraint>
```

Un rôle logique définit donc à quelle ressource un utilisateur authentifié est autorisé d'accéder.

Les rôles logiques de sécurité sont définis via les éléments <security-role>. Ci-dessous il s'agit de la définition du rôle SELLER.

```
<security-role>
  <description/>
  <role-name>SELLER</role-name>
</security-role>
```

Enfin le fichier web.xml spécifie quel mécanisme d'authentification est utilisé. Dans le cas de goodcesi, on utilise un formulaire d'authentification (**FORM**).

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>gcRealm</realm-name>
  <form-login-config>
    <form-login-page>/faces/authMenu.xhtml</form-login-page>
    <form-error-page>/faces/error.xhtml</form-error-page>
  </form-login-config>
</login-config>
```

Cette section indique aussi le domaine de sécurité physique (**gcRealm**) sous-jacent.

La page `authMenu.xhtml` contient le formulaire standard html d'authentification

```
<form action="j_security_check" method="post">
  <!--utilisation de l'objet implicite JSF facesContext-->
  <h:panelGrid
    rendered="#{facesContext.externalContext.userPrincipal==null}">
    identifiant
    <input type="text" name="j_username" class="authInput"/>

    mot de passe
    <input type="password" name="j_password" class="authInput"/>

    &nbsp;
    <input type="submit" value="se connecter"/>
  </h:panelGrid>
```

Les attributs préfixés **j_** sont des attributs standards Java EE liés au mécanisme d'authentification.

Chaque fois qu'un utilisateur tente d'accéder à une ressource web protégée, ce formulaire d'authentification s'interpose. Si l'authentification réussie et que l'utilisateur est associé au rôle de sécurité l'autorisant à accéder à la ressource, la ressource est affichée sinon un message d'erreur est retourné.

Mapping des rôles de sécurité logiques et des groupes physiques

Le système d'autorisation s'appuie sur les rôles logiques définis dans l'application. Il faut donc mettre en correspondance ces rôles logiques avec les groupes physiques définis dans le domaine de sécurité serveur. Pour cela on utilise un descripteur de déploiement propre au serveur d'application. Dans le cas de Payara (fork de GlassFish), il s'agit du fichier de WEB-INF/glassfish-web.xml (ou payara-web.xml).

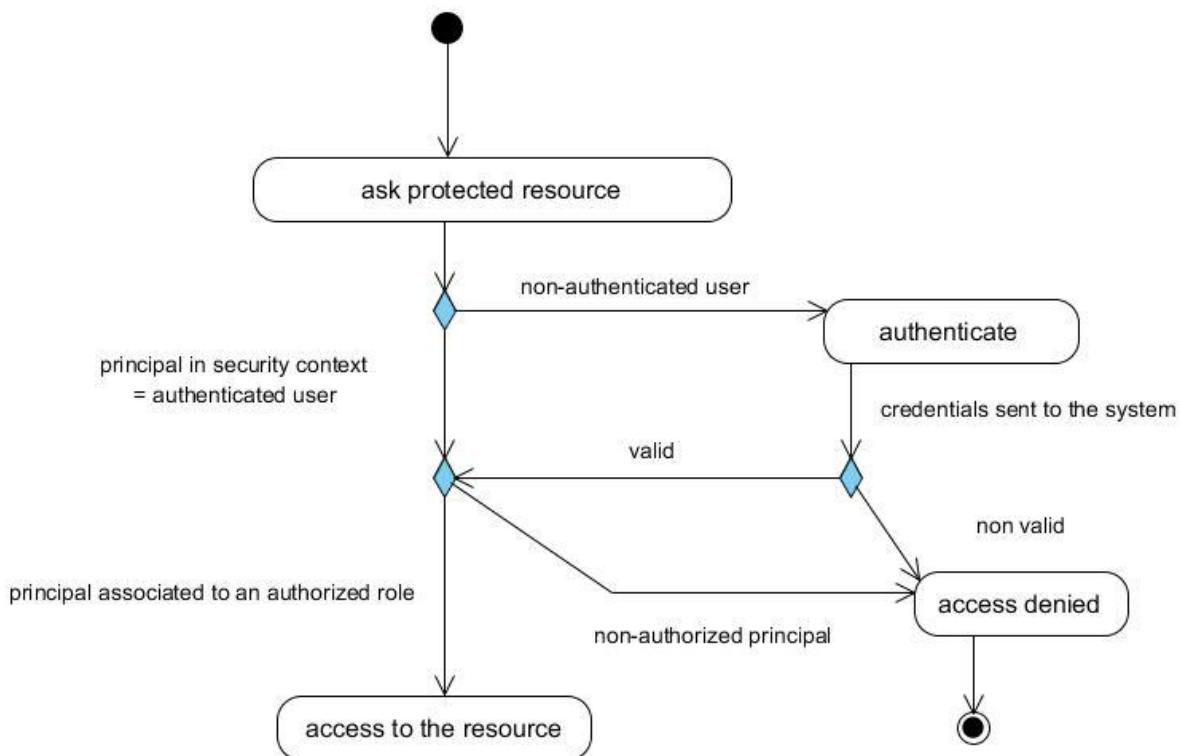
Dans ce fichier, on met donc en correspondance les rôles logiques définis dans `web.xml` avec les groupes physiques stockés dans la base `goodcesidb`. L'extrait ci-dessous montre le mapping entre **SELLER** et **vendeurs**, ce dernier étant stocké en base.

```
</security-role-mapping>
<security-role-mapping>
  <role-name>SELLER</role-name>
  <group-name>vendeurs</group-name>
</security-role-mapping>
```

L'avantage de ce système séparant la sécurité logique de la sécurité physique est qu'on peut changer de système de sécurité physique (utilisation d'un annuaire LDAP, d'un fichier plat, d'un système propriétaire...) en ayant juste à modifier des fichiers XML et sans avoir à modifier le code.

Authentification et autorisation

Le diagramme ci-dessous représente le fonctionnement du mécanisme d'authentification (s'identifier auprès d'un système) et d'autorisation (droit d'accès à une ressource)

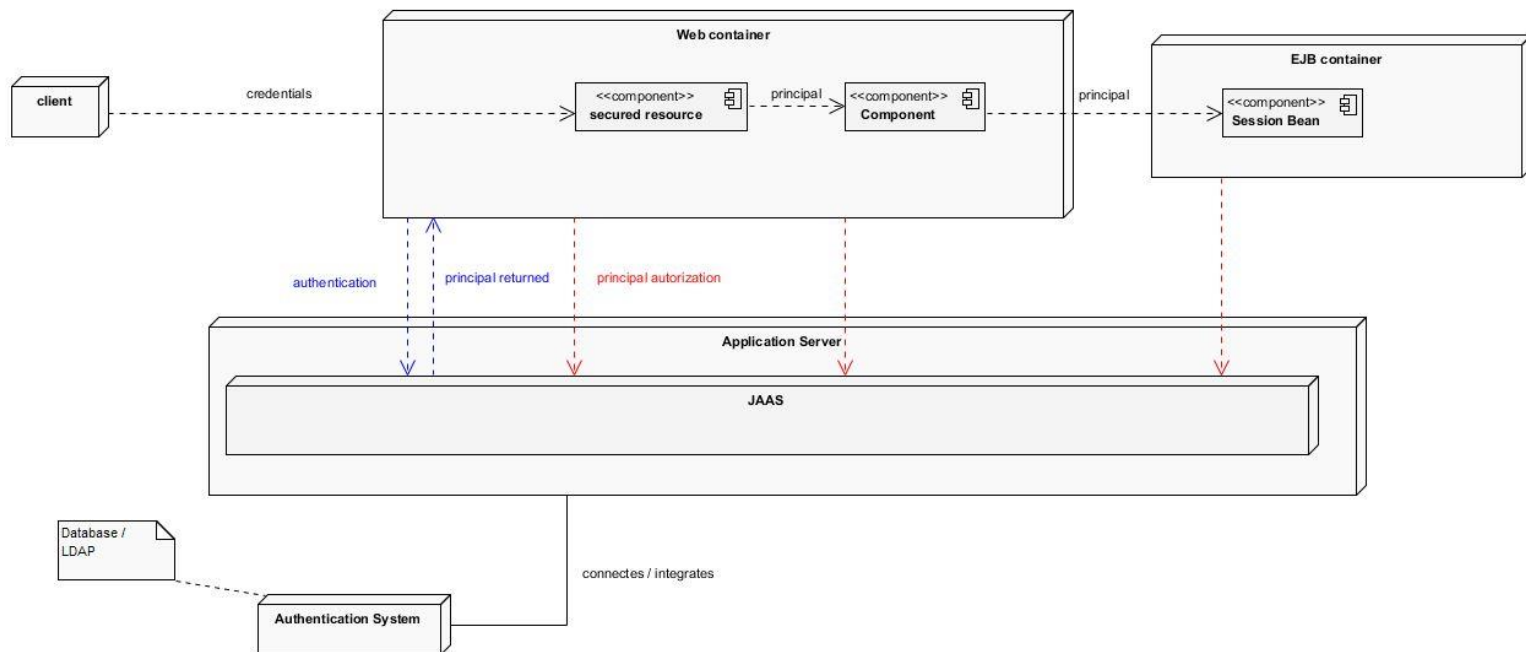


gc_authProc.jpg

Comme dit précédemment toute demande initiale d'accès à une ressource protégée nécessite de s'authentifier. Pour goodcesi, c'est un formulaire qui va permettre à l'utilisateur de saisir son login et mot de passe. Si ces crédeniels (anglicisme qu'on pourrait traduire par référence) sont validés par le système de sécurité Java EE, ce dernier crée une représentation de l'utilisateur authentifié (principal). Le système valide ensuite que le principal est associé à un rôle ayant le droit d'accéder à la ressource. Evidemment si l'authentification ou l'autorisation échouent l'utilisateur ne peut accéder à la ressource.

Infrastructure de sécurité

Le diagramme ci-après est une représentation très schématique de l'infrastructure standard de Java EE.



gc_infraSecu.jpg

L'utilisateur fournit ses credentials à l'application (par l'intermédiaire d'un formulaire dans le cas de goodcesi). Le serveur d'application (Payara en ce qui nous concerne) authentifie l'utilisateur auprès de son service de sécurité¹. Un principal pour cet utilisateur est désormais associé au contexte de sécurité de l'application. Chaque fois qu'un utilisateur désire accéder à une ressource web (une page JSF par exemple) sécurisée le principal est examiné par le service de sécurité pour vérifier que ce dernier est associé à un rôle ayant l'autorisation d'accès. Chaque invocation à un composant de l'application contient le principal. Ainsi lorsqu'un utilisateur tente d'invoquer une méthode sur un session bean EJB par exemple, cette invocation contient le principal. Si la méthode du session bean est sécurisée alors l'invocation sera autorisée uniquement si le service de sécurité détermine que le principal associé à l'invocation est autorisé à exécuter la méthode. Les EJB qu'on pourrait qualifier de composants « plus lourds » que les beans CDI offrent des services de sécurité des appels. Ce mécanisme est mis en œuvre dans un exercice optionnel de la partie session beans.

Le tutoriel Java EE détaille bien plus que cette annexe la sécurité :

<https://docs.oracle.com/javaee/7/tutorial/security-intro001.htm#BNBWK>

¹ Le mécanisme de sécurité Java EE s'appuie sur JAAS ((Java Authentication and Authorization Service) qui est une technologie Java SE. Sur le schéma, le rectangle « JAAS » représente un fournisseur JAAS, c'est-à-dire un module qui intègre un système d'authentification (une base, un annuaire LDAP...) au serveur d'application.

2. Explication du cycle de vie JSF

2.1. *Requête initiale*

Une page / vue JSF (ici la page d'accueil) est accédée pour la première fois au sein d'un navigateur.

RESTORE VIEW :

JSF crée un arbre vide de composants. C'est-à-dire un arbre limité à une racine (UIViewRoot) puis il y a transition vers la phase RENDER RESPONSE.

RENDER RESPONSE :

L'arbre est peuplé avec des objets correspondants aux balises de la page. Puis il est parcouru en vue de générer la réponse au client. L'état de l'arbre est aussi sauvegardé.

Durant cette phase l'instance CategoriesCache est créée car elle est liée à la page. **Les instances de beans CDI sont créées lorsqu'elles sont utilisées.** A noter que le container CDI va déléguer au container EJB la création de l'instance, ou du moins l'orchestrer avec lui car CategoriesCache est aussi un singleton EJB.

2.2. *Requête postback*

Une action est déclenchée dans la vue affichée dans le navigateur (par exemple, accès à une liste d'articles d'une catégorie).

RESTORE VIEW :

L'arbre des composants précédemment sauvegardé est restauré en mémoire.

APPLY REQUEST VALUES :

Les composants de l'arbre sont modifiés par assignation des valeurs stockées dans la requête.

PROCESS VALIDATIONS :

Les valeurs des composants (provenant de la requête cliente) sont converties (conversion du type des données) puis validées. Si la conversion ou la validation échouent alors des exceptions sont levées et les messages d'erreur sont placés dans une queue. Si la conversion ou la validation échouent, le cycle est avorté : JSF passe directement à l'étape Render Response.

UPDATES MODEL VALUES :

Dans le cas d'un clic sur une catégorie, on ne voit rien. Mais c'est durant cette phase que les setters des beans associés à l'attribut *value* des champs de saisie JSF via le langage d'expressions unifié sont invoqués.

Les propriétés des beans sont assignées avec les valeurs des composants JSF converties et validées : c'est-à-dire `unbean.setProperty(composantJSF.getValue())`

INVOKE APPLICATION :

C'est durant cette phase que la logique métier est invoquée.

La méthode d'action « liée » via langage d'expressions à l'attribut *action* d'une balise JSF est déclenchée. La chaîne retournée par la méthode va déterminer la navigation vers une page. Une méthode liée à l'attribut *actionListener* d'une balise (*méthodes actionListener*) est invoquée en fin de phase.

Une instance *CatalogBean* (bean ayant un scope de requête) pour la requête en cours est créée car elle est pour la première fois utilisée pour déclencher une action.

RENDER RESPONSE :

JSF met à jour l'arbre des composants. Dans le cas où il y a navigation vers une autre vue, l'arbre est mis à jour avec les composants exposés dans cette vue. C'est une navigation interne. Il n'y pas de redirection d'URL. C'est pour ça que l'URL de la vue reste inchangé dans la barre d'adresse du navigateur. Pour déclencher une navigation de type requête initiale vers une page il faut suffixer la chaîne de navigation avec *faces-redirect=true*.

Durant cette phase, les getters associés à la vue sont invoqués pour peupler les valeurs des composants rendus au client.

Le cycle de vie de la requête arrive à son terme, l'instance *CatalogBean* associée à cette requête est détruite.

Vous voyez que le même getter d'une instance de bean CDI peut être invoqué par JSF plusieurs fois pour bâtir la réponse. Il ne faut donc pas invoquer des traitements métier dans des getters pour éviter des invocations inutiles de la couche métier.