
# 3A pour JavaEE 6

18 avril 2012

**A**ntoine Sabot-Durand
Ippon Technologies

y**A**nn Blazart
Ippon Technologies

**A**lexis Hassler
Sewatech

# Introduction

En 2006, JavaEE 5 a été conçu avec trois objectifs : la facilité de développement, la facilité de développement et la facilité de développement.La nouvelle version, JavaEE 6 se base sur ces améliorations pour faisant progresser des technologies existantes (nouvelles fonctionnalités, simplifications) et en ajoutant des nouvelles capacités comme BeanValidation, JAX-RS (Web Services RESTful), et la nouvelle et excitante spécification CDI (Context and Dependency Injection).

Cet atelier (ou Hands-on Lab - HOL) est une version allégée de l'exemple fourni en marge du livre “Beginning Java EE 6 with GlassFish 3, second Edition” d'Antonio Goncalves et disponible sur http://beginningee6.kenai.com. Ce site Web contient plus d'une douzaine d'exemples, partant d'une implémentation très basique jusqu'à une application Web complète couvrant les nouveautés de JavaEE 6. Le site Web fournit aussi tout ce dont vous aurez besoin pour progresser sur ces technologies : du code, des instructions, un forum et une mailing-list.

Pour être honnête, cet atelier ne rend pas justice à JavaEE 6, mais essaie de vous faire goûter à la simplicité et à la puissance auxquelles est arrivée la plateforme. Il va principalement couvrir JavaServer Faces 2.0, JAX-RS 1.1 et CDI 1.0 avec quelques références à ManagedBeans, EJB 3.1, JPA 2.0, Servlet 3.0 et plus encore. La section « Résumé des épisodes précédent... » a un paragraphe ou deux pour chacun d'eux.

Si vous êtes pressé, allez directement à l'exercice 1 (JSF) et commencez à coder. Sinon, lisez ce qui suit, avec les section d'instructions d'installation et de discussion du code.

# Prérequis et installation

Le code source peut être extrait de l'URL suivante :
https://github.com/antoinesd/Labs-Java-EE-6

L'ensemble du code source pour cet atelier est fourni sous la forme de modules Maven, lui procurant une bonne portabilité entre environnements de développement. Il n'y a donc aucun prérequis concernant les outils de développement en dehors de Maven lui-même.

Avec la proportion croissante de code Java par rapport aux descripteurs de déploiement XML, et avec la généralisation tu concept de CoC (Convention over Configuration), le support des APIs JavaEE 6 et d'assistants dans l'environnement de développement restent un plus mais ne sont plus une obligation. Ceci dit, Netbeans 7.x, Eclipse 3.7 (Indigo) avec certaines suites de plugins (JBoss Developper Tools, Oracle Enterprise Pack,...) ou IntelliJ offrent de bonnes possibilités.

Les serveurs d'applications cibles dans cet atelier seront Glassfish 3.1 et JBoss AS 7.1. Tous ont été sélectionnés parce qu'ils supportent JavaEE 6 Web Profile et font partie de la famille grandissante des serveurs d'applications modernes. L'atelier nécessite un serveur d'applications JavaEE 6 Web Profile, et vous noterez que les projets Maven dépendent d'un artéfact unique javax:javaee-web-api:jar:6.0 plutôt que d'un ensemble de librairies comme JSF, JPA, EJB,...

Toutes les combinaisons IDE / Serveur d'applications sont théoriquement possibles. Cependant, nous vous conseillons de conserver une certaine cohérence dans vos choix :

- Netbeans + Glassfish
- JBoss Developper Tools / JBoss AS
- IntelliJ / ce que vous voulez

Pour cette atelier, vous aurez besoin :

- d'un environnement de développement (Netbeans, Eclipse, InteliJ) avec Java 6,
- de Maven 3,
- d'un serveur d'applications (Glassfish 3.1, JBoss AS 7.1),
- d'un navigateur Web (Firefox, Chrome,...),
- d'un client HTTP en ligne de commande (wget, cURL,...).

## Résumé des épisodes précédents

Ceci est un résumé de ce qui a précédé l'exercice n°1 de cet atelier, partant de quelques classes, sans interface graphique. Nous vous encourageons à lire cette courte introduction et à étudier le code correspondant à chacun des paragraphes suivants.

### *ManagedBeans 1.0*

L'application développée ici gère des livres. Elle utilise un nouveau type de composants, les **managed beans**. C'est un type de composants légers définis dans JavaEE 6 et qui a été pensé comme la fondation des autres types de composants (EJB, Servlet,...) qui apportent des services additionnels (transactions, sécurité, HTTP,...). Dans notre cas, le composant est annoté avec @javax.annotation.ManagedBean et génère des numéros ISBN, avec un intercepteur qui trace tous les appels à ce service. Jetez un coup d'oeil à IsbnGenerator.java et LoggingInterceptor.java.

### *JPA 2.0*

Les livres sont des entités JPA qui sont stockés dans une base de données avec des attributs comme l'ISBN ?, le titre, le prix, la description, le nombre de pages, la présence ou non d'illustrations et un ensemble de tags (une collections de String, mappée à la table en utilisant un nouvelle fonctionnalité de JPA 2.0). Jetez un coup d'oeil à Book.java.

Comme toujours, l'unité de persistance (PU) est définie dans META-INF/persistence.xml.

### *Servlet 3.0*

L'implémentation initiale utilise une saisie depuis un formulaire HTML et passe les informations à une servlet 3.0 (un POJO annoté par @WebServlet, incluant les information de mapping et écartant de ce fait la nécessité d'avoir un descripteur de déploiement web.xml). Cette servlet est responsable de l'interaction avec l'entity manager de JPA.

### *EJB 3.1*

Afin d'offrir une meilleure séparation des préoccupations, un EJB a été ajouté. La servlet conserve sa responsabilité de front-end Web, de récolter les informations saisies par l'utilisateur alors que le composant transactionnel EJB fournit une interaction propre avec la persistance, retirant le code de gestion des transactions que la servlet devait embarquer. Jetez un coup d'oeil à ItemEJB.java et voyez comment cet EJB est empaqueté dans l'application Web (directement dans le war, un nouvelle fonctionnalité de JavaEE 6).

Un autre EJB a été ajouté, en tant que **singleton**, qui est instancié quand le conteneur démarre (encore un nouveauté de EJB 3.1), pour fournir les codes des langues et leur résolution. Voyez dans LanguageSingleton.java.

Ces deux EJBs peuvent être testés facilement depuis un environnement JavaSE (Junit, Maven,...) en utilisant la nouvelle API standard **EJBContainer**.

# Exercice 1 (JSF)

Comme nous utilisons JSF, il n'y a plus besoin d'écrire nos propres servlets (le framework JSF utilise une servlet *en sous-main*).

Pour commencer, étudiez le code de ItemBean.java. C'est un managed bean JSF (appelé parfois aussi backing bean) défini comme un POJO et annoté par @javax.faces.bean.ManagedBean et @javax.faces.bean.RequestScoped. Dans les versions précédentes de JSF, ces méta-données étaient définies dans le descripteur de déploiement faces-config.xml, qui est devenu optionnel. Ceci rend le répertoire WEB-INF vide, sans web.xml.

Le managed bean utilise de l'injection @EJB pour obtenir des références vers les implémentations ItemEJB et LanguageSingleton.

La méthode initList() (le nom est arbitraire) est annoté avec @PostConstruct qui peut être appliqué à n'importe quelle méthode du managed bean et où la liste des livres est peuplée avec l'aide de ItemEJB, chaque fois que le composant est créé par le conteneur.

Les méthodes doCreateBook() et doNewBookForm() seront référencées depuis les boutons dans la vue JSF que nous allons créer avec Facelets. L'expression de retour de ces méthodes indique la page suivante (dans notre cas, nous restons sur la page courante). C'est une nouvelle fonctionnalité appelée navigation implicite qui n'exige plus d'exprimer les règles de navigation dans le descripteur de déploiement faces-config.xml.

Parmi les nouvelles fonctionnalités de JSF 2.0, on a l'utilisation par défaut de Facelets, un framework utilisé couramment avec les prochaines les précédentes qui est maintenant intégré au standard. Facelets exprime les vues (pages) avec XHTML (HTML bien formé) et lie les composants UI comme les champs de texte vers les attributs des managed beans et les contrôle comme les boutons ou lien vers méthodes doXYZ() précitées. Une telle liaison est faite en utilisant le langage d'expression avec la notation #{...}.

## *Raccorder l'interface graphique au managed bean*

newBook.xhtml contient un formulaire Web pour ajouter des livres, ainsi qu'un tableau qui liste tous les livres qui ont été ajoutés précédemment.

A la fin du formulaire de livre (avant la liste des livres existants), remplacer le commentaire à la ligne #32 avec un bouton de commande et utiliser la langage d'expression (EL) pour l'associer avec la méthode doCreateBook(). Le code devrait être (n'oubliez pas de sauvegarder le fichier) :

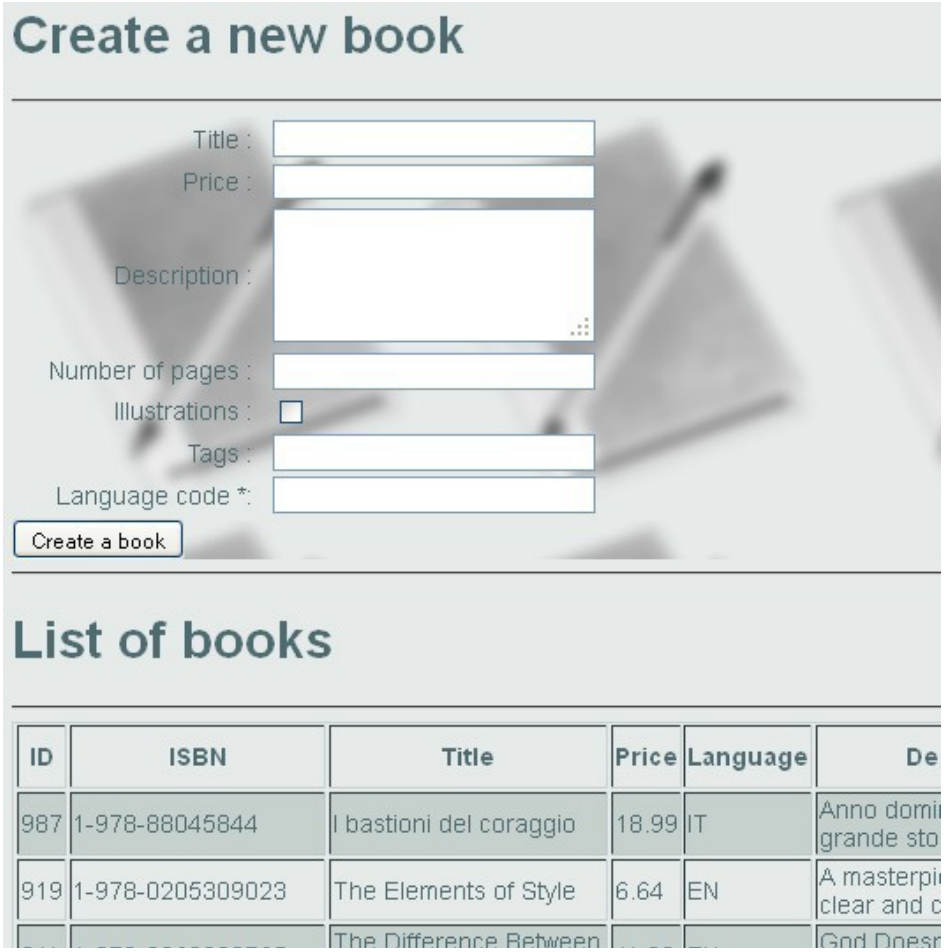
```
    ...
    </h:panelGrid>
    <h:commandButton value="Create a book"
                     action="#{itemBean.doCreateBook}"/>
</h:form>
```

Reconstruisez le projet dans l'IDE ou avec la commande mvn clean install.

Vérifiez que la base de données est démarrée (cf. annexe).

Démarrez le projet depuis l'IDE :

- le serveur d'applications sera démarré,
- l'application Web sera déployée dans le serveur d'applications,
- le navigateur Web sera ouvert avec l'URL http://localhost:8080/exercise1/newBook.faces




Testez l'application en ajoutant quelques livres. Vérifiez les logs du serveur d'applications.

### *Mise en forme et AJAX*

D'autres améliorations de JSF 2.0 concernent l'introduction de composants composites (vous pouvez écrire vos propres composants JSF dans un simple fichier XHTML), de nouvelles fonctionnalités de structuration (templating) et l'intégration complète d'AJAX sous la forme d'un fichier Javascript standardisé et d'une balise <f:ajax>.

Le prochain exercice intègre à la fois des composants composites et de la structuration, donc jetez un coup d'oeil au code source si vous voulez en savoir plus.

Si vous voulez comprendre comment Ajax peut être utilisé pour implémenter un champ texte pour filtrer le contenu du tableau de livres, étudiez newBook.xhtml. La ligne #42 utilise la balise <f:ajax> pour implémenter le rafraîchissement partiel (la liste des livres) de a page à chaque enfoncement de touche dans le champ « Filter ».

Par ailleurs, layout.xhtml illustre comment les nouvelles fonctionnalités de structuration de JSF 2.0 peuvent être utilisée.

Enfin, sous « Web Pages », resources/demo/newItem.xhtml, est un composant composite introduit pour gérer à la fois des livres et des CDs (cf. newCD.xhtml et newBook.xhtml)

### *Validation de la saisie*

Item.java et ItemBean.java contiennent des annotations BeanValidation (JSR 303, autre ajout à JavaEE 6) contraignant les valeurs du titre et du code de langue (avec un message d'erreur codé en dur).

Ces annotations @Size sont en commentaires ; retirez les marques de commentaires et testez à nouveau la saisie du titre et du code de langue.

## Exercice 2 (JAX-RS)

Avant que nous commencions la partie Web Serice RESTful de cet atelier, quelques mots sur le point de départ du second exercice.

Exercise2-JAX-RS va échouer son déploiement tel quel avec un message d'erreur « SEVERE: The ResourceConfig instance does not contain any root ressource classes ». Avec le prochain paragraphe, vous pourrez exposer les ressources RESTful avant d'exécuter le projet.

### *Activer JAX-RS*

Pour activer et configurer JAX-RS dans notre application, vous avez deux possibilités. Soit on configure une Servlet dans le fichier de configuration web.xml, soit vous pouvez utiliser une nouvelle fonctionnalité de JAX-RS 1.1, en créant une classe vide dans le package org.beginningee6.tutorial :

```
@javax.ws.rs.ApplicationPath("resources")
public class ApplicationConfig extends javax.ws.rs.core.Application
{}
```

Cela fait correspondre  la servlet JAX-RS avec le chemin "ressources" et permet en même temps de se passer de web.xml (nouvelle fonctionnalité de Servlet 3.0).

Vous pouvez à nouveau exécuter votre application pour vérifier que tout fonctionne toujours suite à cet ajout.

### *Transformer l'EJB en service REST*

JAX-RS est une technologie aussi simple et élégante que puissante pour développer des Web Service RESTful. La forme la plus simple est un POJO décoré d'une annotation @Path. Dans notre cas, nous allons exploiter une des nouvelles fonctionnalités de JAX-RS 1.1 qui permet de combiner des fonctionnalités d'EJB et de JAX-RS (la version 1.0 de JAX-RS est sortie plus tôt mais n'était pas intégrée à la plateforme Java EE).

Dans ItemEJB.java, ajoutez l'annotation @Path pour promouvoir la classe comme ressource JAX-RS (et ajustez les imports en incluant javax.ws.rs.Path) :

```
@Stateless
@Interceptors(LoggingInterceptor.class)
@Path("/items") // resource available from the /items path
public class ItemEJB {
    ...
}
```

Puis modifiez sa méthode getBook() en utilisant l'annotation JAX-RS suivante (vérifiez que dans les imports que vous utilisez les classes du package javax.ws.rs) :

```
@GET // HTTP's GET verb/operation
@Path("{bookKey}") // specializes the path with a parameter
@Produces(MediaType.APPLICATION_XML) // mime-type public Book
getBook(@PathParam("bookKey") Long bookKey) {
    ...
}
```

La variable bookKey sera initialisée avec la paramètre extrait de l'URL (par exemple, 123 pour l'URL http://localhost:8080/.../items/123).

Finalement, placez une annotation @XmlRootElement dans la classe Book.

Maintenant essayons tout ceci et exécutons le projet pour déclencher le redéploiement. Une fois que la page apparaît dans le navigateur, mémorisez un identifiant de livre et tapez l'adresse suivante dans la barre d'adresse du navigateur : http://localhost:8080/exercise2/resources/items/{id} ({id} est un identifiant de livre), et observez la réponse XML. Le chemin /resources est défini par l'annotation @ApplicationPath.

Notez que certains navigateurs comme Safari ou Chrome affichent de l'XML non formaté. Utilisez le menu de visualisation du code source de la page pour voir la version formatée.

Pour le moment, quelque soit le type MIME demandé, nous obtiendrons de l'XML. Notez aussi que seule la méthode GET est implémentée dans cet atelier ; les autres causeront une erreur HTTP 405 – Method not allowed.

### *Support de plusieurs media types*

Utiliser un navigateur pour accéder à une ressource RESTful est d'un usage limité, donc nous allons migrer vers l'utilisation d'un client HTTP en ligne de commande pour illustrer les autres fonctionnalités de JAX-RS.

Dans ItemEJB.java, modifiez l'annotation @Produces pour proposer json comme format de sortie supplémentaire ; grâce à JAXB et à @XmlRootElement, c'est la seule chose à faire.

```
@Produces({MediaType.APPLICATION_XML,MediaType.APPLICATION_JSON})
```

Redéployez l'application. Vous pouvez ouvrir un terminal et utiliser cURL pour appeler le commandes suivantes :

```
curl http://localhost:8080/exercise/resources/items/{id}
curl -H "Accept:application/json"
          http://localhost:8080/exercise/resources/items/{id}
curl -H "Accept:application/xml"
          http://localhost:8080/exercise/resources/items/{id}
```

- La première commande va retourner une représentation XML du livre dont l'identifiant est 123.
- La deuxième commande utilise un en-tête HTTP "Accept :" et va retourner une représentation JSON du même livre.
- La troisième commande demande explicitement une représentation XML.

# Exercice 3 (CDI)

## *Introduction*

CDI, ou Context and Dependency Injection (JSR 299) est une nouveauté de JavaEE. Vu d'un haut niveau, il peut être décrit comme une solution d'injection de dépendance fortement typée et faiblement couplée, avec une gestion automatique du contexte (comme la portée « conversation ») et d'autres fonctionnalités (événements, stéréotypes, etc...). Cet exercice est une introduction à CDI, centrée sur la partie injection.

CDI est construit au dessus des annotations de la JSR 330 (Dependency Injection for Java), qui sont @Inject, @Qualifier et @Named. Nous utiliserons ces trois annotations dans le contexte de notre application Web. Notez bien que CDI est un composant de JavaEE 6 Web Profile ce qui signifie que tout conteneur qui supportee ce standard pourra exécuter le code qui suit.

## *Activer CDI*

Pour activer CDI, les artéfacts déployés dans le runtime doivent contenir un fichier beans.xml (localisé dans WEB-INF dans notre cas). Au cours de cet atelier, ce fichier restera vide mais il doit être présent pour que CDI fonctionne (ce fichier peut être utilisé pour des cas plus avancés comme les alternative quand la séparation des responsabilités est une exigence clé).

Avec l'IDE ou une autre méthode (la commande Unix touch par exemple), créez un fichier vide WEB-INF/beans.xml dans votre projet.

## *Standardisation avec @Inject*

Dans ItemEJB.java, remplacez les occurrences de @Resource avec @Inject et ajustez les imports. Cela délègue à CDI l'injection des ressources. Cette résolution de l'injection sera réalisée au démarrage, ainsi une quelconque erreur sera provoquée tôt plutôt que d'avoir des injections causant des NullPointerExceptions à l'exécution.

De la même façon, dans ItemBean.java, remplacez @EJB par @Inject et ajustez les imports.

Vous pouvez aussi supprimer l'annotation @ManagedBean de la classe IsbnGenerator. Elle était nécessaire pour l'injection par @Resource, mais est inutile avec CDI.

Vous pouvez maintenant exécuter à nouveau l'application et comprendre que c'est juste un refactoring qui ne modifie pas le comportement.

## *Accès depuis les facelets JSF*

Ajoutez @Named("languages") à la classe définie dans LanguageSingleton.java pour rendre tous ses attributs disponibles depuis l'EL (langage d'expression) dans les vue JSF XHTML. Notez aussi la nouvelle méthode suivante :

```
public String getListOfValues() {
  return languages.keySet().toString();
}
```

Dans newBook.xhtml, après le code de langue <h:inputText> à la ligne #30, ajoutez le code suivant (idéalement, utilisez la complétion de code pour éviter les fautes de frappe) :

```
<h:outputLabel value="*" styleClass="alabel"/>
<h:outputLabel value="#{languages.listOfValues}"
               styleClass="ainput"/>
```

Ceci utilise un bean appelé « languages » (qui a été défini ci-dessus par @Named) pour énumérer les options valides pour les codes de languages sur deux lettres. Notez que l'annotation @Named peut être utilisée sans attribut auquel cas le nom du bean est dérivé du nom de la classe Java.

### *Injection fortement typée avec les qualifiers*

Regardez maintenant le fichier source Customer.java. Il définit une simple interface Java avec deux méthodes implémentées dans les classes concrètes définies dans DefaultCustomer.java et SpecialCustomer.java. Comme vous pouvez le voir dans le code, le client « default » ne pas acheter quoi que ce soit alors que le client « special » peut le faire.

Dans ItemBean.java, sous l'annotation @ManagedBean, ajoutez une annotation @Named("itemBean"). Changez aussi l'import de @RequestScoped, javax.enterprise.context.RequestScoped au lieu de l'import javax.faces.bean.RequestScoped;. (Vous pouvez éventuellement retirer l'annotation @ManagedBean qui n'est plus utile)
Pour commencer à travailler avec un client, nous avons besoin d'injecter l'attribut suivant dans la classe ItemBean :

```
@Inject Customer cust;
```

… et observez les méthodes suivantes :

```
getForbiddenToBuy()
getBuyButtonLabel()
doBuy()
```

Les deux premières méthodes sont liées à la page JSF newBook.xhtml (lignes #120 et #121) alors que la méthode doBuy() va effectivement placer la commande pour un numéro ISBN donné (ligne #119). Les numéros de lignes peuvent varier d'une ou deux unités selon la mise en forme des exercices précédents.

Essayez d'exécuter l'application.

Le déploiement devrait échouer et les logs du serveur d'applications devraient montrer un message similaire à :

```
Exception while loading the app : WELD-001409 Ambiguous
dependencies for type [Customer] with qualifiers [@Default] at
injection point [[field] @Inject
org.beginningee6.tutorial.ItemBean.cust]. Possible dependencies
[[Managed Bean [class org.beginningee6.tutorial.DefaultCustomer]
with qualifiers [@Any @Default], Managed Bean [class
org.beginningee6.tutorial.SpecialCustomer] with qualifiers [@Any
@Default]]]
```

Comme vous pouvez le voir, le serveur d'applications n'a pas assez d'informations pour savoir quelle implémentation de Customer il doit injecter puisque SpecialCustomer et DefaultCustomer implémentent cette interface.

Pour résoudre ça, nous allons utiliser un Qualifier CDI défini dans Premium.java. Ceci est une annotation personnalisée que vous pouvez utiliser pour marquer les implémentations et qualifier les injections. Dans notre cas, nous devons ajouter @Premium à la classe

SpecialCustomer :

```
@Premium
public class SpecialCustomer implements Customer {

    @Override
    public OrderItem buy(String id) {
        ...
    }
}
```

Maintenant, ré-exécutez l'application.

Vous devriez voir que le serveur d'applications résout l'injection en fournissant une instance de DefaultCustomer. Ceci veut dire que l'utilisateur n'est pas capable d'acheter quoi que ce soit, de sorte que l'interface graphique désactive les contrôles : bouton « Sorry, can't buy ».

A présent, modifions l'injection dans ItemBean.java comme ceci :

```
@Inject @Premium Customer cust;
```

Exécutez à nouveau l'application et vérifiez que, cette fois-ci, que SpecialCustomer est l'implémentation utilisée et que l'utilisateur est capable d'acheter (bouton « Buy me ! »). Aucune fonctionnalité de réservation n'est implémentée ici, par conséquent vous devrez consulter les logs pour voir les messages de passation de commande.

Ceci conclut la partie injection CDI de cet atelier. La fonction clé est la capacité à changer d'implémentation sans avoir à changer le code d'appel/injection. C'est la raison pour laquelle CDI est considéré comme faiblement couplé alors que la partie qualifier est celle pour laquelle CDI est considéré comme fortement typé (pas de chaîne de caractères).

## *Injection contextuelle : loggers*

Dans notre application, les traces sont gérées par l'intercepteur LoggingInterceptor, utilisé avec ItemEJB. Depuis JavaEE 6, les intercepteurs ne sont plus uniquement associés à des EJBs mais peuvent intercepter les invocations d'autres composants comme des Beans CDI.

Plutôt que d'utiliser cet intercepteur, nous allons injecter un logger dans nos beans. Par exemple, dans ItemEJB, nous allons créer un champ de type java.util.logging.Logger :

```
    @Inject
    private Logger logger;
```

Evidemment, Logger n'est pas un bean CDI, donc si on essaie d'exécuter l'application ainsi, l'injection sera en échec. Nous allons donc produire des instances de Logger. Pour cela, nous allons créer une classe LoggerProducer avec une méthode annotée par @javax.enterprise.inject.Produces :

```
public class LoggerProducer {
    @Produces
    public Logger getLogger() {
        return Logger.getLogger(LoggerProducer.class.getName());
    }
}
```

Dans ces conditions, l'injection fonctionnera et nous pourrons utiliser le champ logger dans nos méthodes. Ceci dit, c'est encore insuffisant car toutes les traces sortiront depuis un Logger unique, ce qui limite grandement l'intérêt d'un API de traces.

Pour affiner le fonctionnement de notre producteur, nous pouvons le rendre contextuelle en demandant à CDI de lui passe une information appelée point d'injection.

```
public class LoggerProducer {
    @Produces
    public Logger getLogger(InjectionPoint injectionPoint) {
        return Logger.getLogger(
            injectionPoint.getBean().getBeanClass().getName());
    }
}
```

Vous constaterez que le cas d'ItemBean est un peu délicat du fait de son rôle de bean JSF et CDI. Pour que le point d'injection ne soit pas null, il ne faut pas que le bean soit annoté avec @javax.faces.bean.ManagedBean.

Dans notre exemple, nous utilisons java.util.logging, mais cette façon de faire fonctionne aussi avec Log4J, SLF4J ou d'autres APIs.

## Exercice 4 (Tests)

Dans cet exercice, nous vous fournissons des tests unitaires pour ItemEJB et ItemBean. Pour faire ces tests, nous devons préparer des objets Mock. Nous testons ainsi le code de nos méthodes, mais pas la façon dont nous avons assemblé nos composants.

Pour faire des tests plus pertinents, nous allons faire des tests d'intégration, avec Arquillian. Ce n'est pas dans JavaEE, mais c'est un outil qui permet de tester des composants JavaEE en les déployant dans leur conteneur.

La structure d'un test Arquillian est la suivante :

– Utiliser le runner Arquillian

```
@RunWith(Arquillian.class)
```

– Préparer une archive à déployer (oui oui, on déploie à partir du code de test), en y ajoutant des classes, un descripteur JPA et un descripteur CDI

```
@Deployment
public JavaArchive deploy() {
  return ShrinkWrap.create(JavaArchive.class)
                   .addPackage(ItemEJB.class.getPackage())
                   .addAsResource("META-INF/persistence.xml",
                                  "META-INF/persistence.xml")
                   .addAsManifestResource("META-INF/beans.xml",
                                          "beans.xml");
}
```

– Injecter les objets à tester (le test s'exécutant dans le conteneur, il bénéficie de ses capacités)

```
@EJB ItemEJB itemEJB;
```

– Tester

```
@Test
public void shouldGetBookSimplyReturnFoundBook() {
  ...
}
```

Evidemment, nous testons avec une base de données, sans avoir la main sur les données contenues. Un équivalent de DbUnit serait pratique ; il y a une extension pour ça... A défaut, on peut déployer un EJB qu'on injecte dans notre test et qu'on appelle avant et après les tests.

```
@EJB DBInit dBInit;

@Before
public void initData() {
  dBInit.initDatabase();
}
```

Les méthodes annotées @Before sont exécutées dans le conteneur, elles bénéficient donc des composants injectés.

## Complément : CDI et événements

CDI apporte un mécanisme très puissant de gestion des événements au sein des applications.

Dans l'exercice, la classe SpecialCustomer comporte le code suivant :

```
    @Inject @ImportantOrder Event<OrderItem> order;

    @Override
    public OrderItem buy(String id) {
        OrderItem theOrder = new OrderItem(id);
        order.fire(theOrder);
        return theOrder;
    }
```

En clair, CDI lui injecte un gestionnaire d'événement et permet ainsi à SpecialCustomer « d'avertir » d'autres composants de l'application d'un achat.

Pour vérifier le fonctionnement, créez deux EJB Singleton StockManager et StatsManager par exemple et ajoutez une trace de la manière suivante :

```
@Singleton  // de ejb
@Startup
public class StatsManager {
    @Inject
    private Logger logger;

    public void prepareOrder(@Observes OrderItem orderItem) {
        logger.info("Stats new order :"+orderItem.toString());
    }
```

Quand vous cliquerez sur « Buy me ! » vous verrez les traces apparaître dans la console.

Maintenant, dans un de vos deux EJB, qualifiez le paramètre orderItem de la méthode prepareOrder avec «  @NormalOrder » :

```
public void prepareOrder(@Observes @NormalOrder OrderItem orderItem)
{
    logger.info("Stats new order :"+orderItem.toString());
}
```

En retentant, vous verrez que la méthode ne recevra plus le message.

En effet, dans SpecialCustomer, l'EventOrder est qualifié avec @ImportantOrder, les « message » qu'il envoie donc le sont aussi. Par défaut un @Observes prends tous les messages correspondant au type java émis, sauf si l'on spécifie le qualifier.

### *Quelques infos en plus*

CDI crée automatiquement le bean qui a la méthode d'écoute de l'événement, sauf si on lui précise le contraire :

```
    @Observes(notifyObserver= Reception.IF_EXISTS)
```

Dans ce cas, la méthode ne sera appelée qui si un instance du bean encapsulant existe déjà.

On peut aussi paramétrer la réaction à un événement à des moments particulier des transactions :

```
    @Observes(during= TransactionPhase.AFTER_SUCCESS)
```

→ AFTER_COMPLETION : une fois que la transaction est finie (succès ou échec).

→ AFTER_FAILURE : une fois que la transaction a échouée. Cela peut aidé par exemple pour contrôler des systèmes ne pouvant être attachés au moniteur transactionnel du server (complétude).

→ AFTER_SUCCESS : une fois la transaction réussie.

→ BEFORE_COMPLETION : avant que la transaction n'essaie d'être validée. Cela permet par exemple de l'interrompre.

→ IN_PROGRESS : l'événement est traité au moment ou il est déclenché.

## Conclusion

Nous espérons que cet atelier a éveillé suffisamment de curiosité pour que vous ayez envie de télécharger un serveur d'applications moderne et de développer avec JavaEE 6.

# Aide Netbeans

## *Installation*

Lorqu'on installe Netbeans, on installe en même temps Glassfish et JavaDB (ie. Derby).

Au premier démarrage, il demande si on souhaite télécharger jUnit. Faites-le !

## *Ajouter un serveur*

A priori, il ne devrait pas être nécessaire d'ajouter un serveur à Netbeans. En effet, Netbeans 7.1 intègre Glassfish 3.1, mais n'est pas compatible avec JBoss AS 7.

## *Gérer les imports*

Clic droit sur le code source et sélection de « Fix Imports » ou CTRL + SHIFT + I

## *Tests JAX-RS*

Comme alternative au navigateur, vous pouvez utiliser la fonction de test RESTful de Netbeans. Pour cela, il faut faire un clic droit sur “RESTful Web Services”, dans notre projet, puis sélectionner “Test RESTful Web Services”.

## *Créer le fichier bean.xml*

Clic droit sur WEB-INF et choisissez New > XML Document..., écrivez « beans » dans le nom (l'extension sera ajoutée automatiquement) ou clic droit sur le projet et choisissez New > Other... > Context and Dependency Injection > beans.xml.

## *Erreur en test unitaire*

Si les tests unitaires se lancent depuis Maven mais pas dans Netbeans, c'est à cause de la fonctionnalité « Compile On Save ». Dans ce cas, il faut aller dans File → Project Properties → Build / Compile et passer l'option « Compile On Save » de « for both application and test execution » à « For application only »