Weld - Implémentation de
Référence de la JSR-299

JSR-299 : Le nouveau standard Java pour l'injection de dépendances et la gestion contextuelle du cycle de vie.

Gavin King

Pete Muir

Dan Allen

David Allen

Traduction italienne: Nicola Benaglia, Francesco Milesi

Traduction espagnole: Gladys Guerrero
Traduction coréenne: Eun-Ju Ki,
Traduction chinoise traditionnelle: Terry Chuang
Traduction chinoise simplifiée: Sean Wu

Note sur la nomenclature et les règles de nommage

Peu de temps avant la publication de la version finale, la spécification JSR-299 a été renommée en passant de "Composants Web" (Web Beans) à "Injection de Contextes et de Dépendances pour la plateforme Java EE" (Java Contexts and Dependency Injection for the Java EE platform : CDI). Cette mise à jour a amené plus de confusion que de clarté et finalement Red Hat a choisi de renommer l'implémentation de référence "Weld". Si vous rencontrez des références documentaires (forums, blogs, articles ...) utilisant l'ancienne nomenclature, mettez-les à jour s'il vous plaît. Le jeu de renommage est fini.

Vous pourrez constater que certaines fonctionnalités telles que la définition XML des composants ont disparu. Ces fonctionnalités seront proposées en tant qu'extensions du projet Weld et potentiellement dans autres implémentations de CDI.

Veuillez noter que la rédaction de ce guide de référence a commencé alors que la spécification était encore en cours d'évolution. Nous avons fait de notre mieux pour la maintenir à jour. En cas de conflit entre ce guide et la spécification, la spécification prévaut étant supposée correcte. Si vous pensez avoir trouvée une erreur dans la spécification, notifiez s'il vous plaît le groupe d'experts de la JSR-299 (JSR-299 EG).

Partie I. Composants

The *JSR-299* [http://jcp.org/en/jsr/detail?id=299] specification (CDI) defines a set of complementary services that help improve the structure of application code. CDI layers an enhanced lifecycle and interaction model over existing Java component types, including managed beans and Enterprise Java Beans. The CDI services provide:

- un cycle de vie amélioré pour les objets avec état, liés à des *contexts* bien définis,
- une approche fortement typée à *l'injection de dépendances*,
- des interactions avec les objets via une *notification d'évènements facilitée*,
- une meilleure approche pour connecter les *intercepteurs* aux objets, avec une nouvelle sorte d'intercepteurs, appelés *décorateur* qui sont plus appropriés pour résoudre les problèmes métier, and
- un *SPI* pour développer des extensions portables au conteneur.

The CDI services are a core aspect of the Java EE platform and include full support for Java EE modularity and the Java EE component architecture. But the specification does not limit the use of CDI to the Java EE environment. In the Java SE environment, the services might be provided by a standalone CDI implementation like Weld (see *Section 18.4.1, « Module CDI SE »*), or even by a container that also implements the subset of EJB defined for embedded usage by the EJB 3.1 specification. CDI is especially useful in the context of web application development, but the problems it solves are general development concerns and it is therefore applicable to a wide variety of application.

An object bound to a lifecycle context is called a bean. CDI includes built-in support for several different kinds of bean, including the following Java EE component types:

- managed beans, and
- les EJB session.

Both managed beans and EJB session beans may inject other beans. But some other objects, which are not themselves beans in the sense used here, may also have beans injected via CDI. In the Java EE platform, the following kinds of component may have beans injected:

- message-driven beans,
- intercepteurs,
- servlets, servlet filters et servlet event listeners,
- JAX-WS service endpoints and handlers, and
- JSP tag handlers and tag library event listeners.

CDI soulage l'utilisateur d'une API inconnue de se poser les questions suivantes :

- Quel est le cycle de vie de cet objet ?
- Combien de clients simultanés peut-il avoir ?
- Est-il multi-threadé ?
- Comment dois je y avoir accès depuis la partie cliente ?

- Dois-je explicitement le détruire ?
- Où dois-je conserver les références vers cet objet quand je ne suis pas en train de les utiliser ?
- Comment puis-je définir une implémentation alternative, afin que cette implémentation puisse varier au déploiement ?
- Comment dois-je faire pour partager cet objet avec d'autres ?

CDI est plus qu'un framework. C'est un modèle complet de programmation riche. Le *principe fondamental* de CDI est *couplage lâche avec typage fort*. Étudions ce que cette phrase signifie.

Un bean spécifie seulement le type et la sémantique des autres beans dont il dépend. Il n'a pas besoin de connaitre le cycle de vie actuel, l'implémentation concrète, le modèle de threads ou les autres clients des beans qui interagissent avec lui. Mieux encore, l'implémentation concrète, le cycle de vie et le modèle de thread d'un bean peuvent varier suivant le scénario de déploiement, sans affecter les clients. Ce couplage lâche rend votre code plus simple à maintenir.

Les évènements, les intercepteurs et les décorateurs améliorent le couplage lâche propre à ce modèle :

- les *notifications d'évènements* dissocient les producteurs des consommateurs,
- les *intercepteurs* dissocient les problèmes techniques de la logique métier, et
- les *décorateurs* permettent aux problématiques métier d'être compartimentés.

Ce qui est encore plus puissant (et réconfortant) est que CDI fournit toutes ces facilités de façon *fortement typée*. CDI ne s'appuie pas sur des identifiants basés sur des String pour déterminer comment les objets collaboreront ensemble. A la place, CDI utilise l'information de typage qui est déjà disponible dans le modèle objet Java, augmenté en utilisant un nouveau pattern de programmation, appelé *annotations qualifier*, pour brancher ensemble les beans, leurs dépendances, leurs intercepteurs et décorateurs, et leurs consommateurs d'évènements. L'usage de descripteurs XML est réduite aux informations réellement spécifique au déploiement.

Mais CDI n'est pas un modèle restrictif de programmation. Il ne vous dit pas comment vous devez structurer couches applicatives, comment vous devez manipuler la persistance, ou quel framework web vous devez utiliser. Vous aurez à décider ce type de chose vous même.

CDI fournit même un SPI compréhensif, permettant à d'autres types d'objets définis par les futures spécifications Java EE ou par un framework tiers d'être proprement intégré avec CDI, de prendre avantage des services CDI, et d'interagir avec n'importe quel type de bean.

CDI a été influencé par de nombreux frameworks Java existants, dont Seam, Guice et Spring. Cependant, CDI a son propre, vraiment distinct, caractère : plus fortement typé que Seam, plus stateful et moins centré sur XML que Spring, plus compétent avec les applications web et d'entreprise que Guice. Mais cela n'aurait pas été possible sans l'inspiration apportée par ces frameworks ainsi que l'*énorme* collaboration et travail de l'Expert Group JSR-299 (EG).

Enfin, CDI est un standard du *Java Community Process* [http://jcp.org] (JCP). Java EE 6 a besoin que tous les serveurs d'application compatibles prennent en charge la JSR-299 (même avec le profile web).

Introduction

Donc, vous avez envi de commencer à écrire votre premier bean ? Ou peut-être est vous sceptique, vous demandant quelles sections de la spécification CDI vous pouvez sauter ! La bonne nouvelle est que vous avez déjà écrit et utilisé des centaines, voire des milliers de beans. CDI rend juste leur usage courant plus simple pour construire une application !

1.1. Qu'est ce qu'un bean ?

Un bean est exactement ce que vous pensez que c'est. Sauf que maintenant, il possède une véritable identité dans l'environnement du conteneur.

Avant Java EE 6, il n'y avait pas de définition claire du terme "bean" dans la plateforme Java EE. Bien sur, vous appelez depuis des années "beans", des classes Java utilisées dans des applications web et d'entreprise. Il y a quand même quelques concepts différentes pour "beans" dans les spécifications EE, incluant les beans EJB et les managed beans JSF. Pendant ce temps, d'autres frameworks tiers, comme Spring ou Seam, ont introduit leurs propres idées de ce que signifie d'être un "bean". Ce dont nous avons manqué est d'une définition commune.

Java EE 6 définit enfin une définition commune dans la spécification Managed Beans. Les Managed Beans sont défini comme des objets managés par un conteneur avec un minimum de restriction de programmation, aussi connu sous l'acronyme POJO (Plain Old Java Object). Ils supportent un petit ensemble de services basiques, comme l'injection de ressources, les callbacks du cycle de vie et les intercepteurs. Les spécifications complémentaires, comme EJB et CDI, construisent sur ce modèle de base. Mais, *enfin*, il y a un concept uniforme pour un bean et un modèle de composant léger qui est commun pour toute la plateforme Java EE.

Avec vraiment peu d'exceptions, presque toutes les classes concrètes Java qui on un constructeur sans paramètres (ou un constructeur annoté avec `@Inject`) est un bean. Ceci inclut tous les JavaBean et tous les EJB session. Si vous avez déjà quelques JavaBeans ou beans session qui trainent, ils sont déjà des beans—vous n'avez pas besoin d'ajouter de meta-données spéciales. Il y a juste une petite chose que vous devez faire avec de pouvoir les injecter dans votre fourbi : vous devez les mettre dans une archive (un jar, ou un module Java EE comme un war ou un EJB jar) qui contient un fichier de marquage spécial : `META-INF/beans.xml`.

Les JavaBeans et EJBs que vous écrivez tous les jours n'étaient, jusqu'à maintenant, pas capable de prendre avantage des nouveaux services définis par la spécification CDI. Mais vous serez capable d'utiliser chacun d'eux avec CDI—en permettant au conteneur de créer et détruire les instances de vos beans et de les associer avec un contexte défini, de les injecter dans d'autres beans, en les utilisant dans des expressions EL, en les spécialisant avec des annotations qualifiantes, et même de leurs ajouter des intercepteurs et des décorateurs—sans modifier votre code existant. Au plus, vous devrez ajouter quelques annotations.

Maintenant, voyons comment créer votre premier bean qui utilise réellement CDI.

1.2. Mouillons notre chemise

Supposons que nous avons deux classes Java existantes que nous utilisons depuis des années dans des applications variées. La première classe transforme une chaîne de caractères en une liste de phrases :

```
public class SentenceParser {
   public List<String
> parse(String text) { ... }
}
```

La seconde classe existante est un bean session sans état faisant facade pour un système externe qui est capable de traduire des phrases d'un langage à un autre :

```
@Stateless
public class SentenceTranslator implements Translator {
   public String translate(String sentence) { ... }
}
```

`Translator` est une interface EJB locale :

```
@Local
public interface Translator {
   public String translate(String sentence);
}
```

Malheureusement, nous n'avons pas une classe qui traduit des documents texte en intégralité. Alors écrivons un bean pour faire ce travail :

```
public class TextTranslator {
   private SentenceParser sentenceParser;
   private Translator sentenceTranslator;

   @Inject
   TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
      this.sentenceParser = sentenceParser;
      this.sentenceTranslator = sentenceTranslator;
   }

   public String translate(String text) {
      StringBuilder sb = new StringBuilder();
      for (String sentence: sentenceParser.parse(text)) {
          sb.append(sentenceTranslator.translate(sentence));
      }
      return sb.toString();
   }
}
```

Mais attendez ! `TextTranslator` n'a pas de constructeur sans paramètres ! Est-il encore un bean ? Si vous vous rappelez, une classe qui n'a pas de constructeur sans paramètres peut quand même être un bean si elle a un constructeur annoté `@Inject`.

Comme vous l'avez deviné, l'annotation `@Inject` a quelque chose à voir avec l'injection de dépendances ! `@Inject` peut être appliqué à un constructeur ou une méthode d'un bean, et dit au conteneur d'appeler ce constructeur ou cette méthode lors de l'instanciation du bean. Le conteneur injectera les autres beans comme paramètres du constructeur ou de la méthode.

Nous pouvons obtenir une instance de `TextTranslator` en l'injectant dans un constructeur, une méthode ou un attribut du bean, ou un attribut ou méthode d'une classe d'un composant Java EE comme une servlet. Le conteneur choisit l'objet à injecter d'après le type du point d'injection, et non pas d'après le nom de l'attribut, de la méthode ou du paramètre.

Nous allons créer un bean controller d'UI qui utilise l'injection par attribut pour obtenir une instance de `TextTranslator`, et traduire le texte entré par un utilisateur :

```
@Named @RequestScoped
public class TranslateController {
   @Inject TextTranslator textTranslator;

   private String inputText;
   private String translation;

   // JSF action method, perhaps
   public void translate() {
      translation = textTranslator.translate(inputText);
   }

   public String getInputText() {
      return inputText;
   }

   public void setInputText(String text) {
      this.inputText = text;
   }

   public String getTranslation() {
      return translation;
   }
}
```

(1) L'injection par attributs d'une instance `TextTranslator`

Astuce

Notez que le bean controller est de portée requête et est nommé. Comme cette combinaison est si commune dans les applications web, il y a une annotation intégrée pour ça dans CDI que vous pouvez utiliser comme raccourci. Quand l'annotation (stéréotype) `@Model` est déclaré sur une classe, cela crée un bean nommé et de portée requête.

Sinon, nous pouvons obtenir une instance de `TextTranslator` par programmation depuis une instance injectée de `Instance`, paramétré avec le type du bean :

```
@Inject Instance<TextTranslator
> textTranslatorInstance;
...
public void translate() {
   textTranslatorInstance.get().translate(inputText);
}
```

Notez qu'il n'est pas nécessaire de créer une méthode getter ou setter pour injecter un bean dans un autre. CDI peut accéder directement à un attribut injecté (même s'il est privé !), ce qui permet parfois d'aider à éliminer le code inutile. Le nom de l'attribut est arbitraire. C'est le type de l'attribut qui détermine ce qui est injecté.

A l'initialisation du système, le conteneur doit vérifier que qu'il existe exactement un bean qui corresponde à chaque point d'injection. Dans notre exemple, si aucune implémentation de `Translator` est disponible—si l'EJB `SentenceTranslator` n'a pas été déployé—le conteneur nous informera de la *dépendance insatisfaite*. Si plus d'une implémentation de `Translator` est disponible, le conteneur nous informera de la *dépendance ambiguë*.

Avant d'aller trop profondément dans les détails, arrêtons nous et examinons l'anatomie d'un bean. Quels aspects du bean sont significatifs, et qu'est ce qui le rend identifiable ? Au lieu de juste donner des exemples de beans, nous allons définir ce qui *fait* de quelque chose un bean.

Un peu plus sur les beans

Un bean est généralement une classe de l'application qui contient de la logique métier. Il peut être appelé directement depuis du code Java, ou invoqué à travers le langage de script unifié (Unified EL). Un bean peut accéder à des ressources transactionnelles. Les dépendances entre les beans sont gérées automatiquement par le conteneur. La plupart des beans maintiennent un *état* et sont *contextuels*. Le cycle de vie d'un bean est toujours géré par le conteneur.

Mais reprenons un instant. Que signifie réellement *contextuel* ? Puisque les beans peuvent maintenir un état, il faut identifier *quelle* est l'instance de bean utitilisée. Contrairement à un composant sans état (par exemple, les beans session sans état) ou à un singleton (comme les servlets, ou les beans singleton), les différents clients d'un bean le voient dans un état différent. L'état du bean auquel accède le client dépend de l'instance de bean référencée.

Cependant, tout comme un composant sans état ou un singleton, mais *contrairement* à un bean de session avec état, le client ne contrôle pas le cycle de vie de l'instance et ne peut créer ou supprimer explicitement une instance. C'est en effet la *portée* (scope) du bean qui détermine :

- le cycle de vie de chaque instance et
- les client qui partagent une référence sur une instance spécifique du bean.

Pour un thread d'exécution donné dans une application CDI, il peut y avoir un *contexte actif* associé à la portée du bean. Ce contexte peut être propre au thread (par exemple, si le bean a une portée requête), ou être partagé avec d'autres threads (par exemple, si le bean a une portée de session) voire même partagé avec tous les threads (si le bean a une portée application).

Les clients (par exemple, d'autres beans) qui s'exécutent dans le même contexte pointeront sur la même instance de bean. Mais les appelants placés dans autre contexte pourront pointer sur une instance différente (en fonction de la relation entre les contextes).

Un avantage important du modèle contextuel est de pouvoir traiter les beans avec état comme des services ! L'appelant n'a pas à se préoccuper de la gestion du cycle de vie du bean qu'il utilise, *ni même de connaître ce cycle de vie.* Les beans interagissent en envoyant des messages, et les implémentations définissent le cycle de vie de leur propre état. Les beans sont faiblement couplés car :

- ils interagissent via des APIs publiques claires
- leurs cycles de vie sont complètement découplés

We can replace one bean with another different bean that implements the same interface and has a different lifecycle (a different scope) without affecting the other bean implementation. In fact, CDI defines a simple facility for overriding bean implementations at deployment time, as we will see in *Section 4.7, « Alternatives »*.

Il faut noter que les appelants d'un bean ne sont pas forcément eux-mêmes des beans. D'autres objets comme les servlets et les message-driven beans qui sont par nature des objets contextuels non injectables, peuvent aussi obtenir des références sur des beans par injection.

2.1. Anatomie d'un bean

Assez controversé. De manière formelle, l'anatomie d'un bean, d'après les spécifications :

> Un bean dispose des attributs suivants :
>
> - Un ensemble (non vide) de types

- Un ensemble (non vide) de qualifiants
- Une portée
- Facultativement, un nom EL
- Un ensemble de branchements d'interception
- Une implémentation

De plus, un bean peut être ou peut ne pas être une alternative.

Voyons ce que ces nouveaux termes signifient.

2.1.1. Types, qualifiants et injection de dépendances

Les beans obtiennent généralement des références sur d'autres beans par injection de dépendances. Tout attribut injecté spécifie un "contrat" qui doit être satisfait par le bean injecté. Ce contrat est :

- un type, associé à
- un ensemble de qualifiants.

Un type est une de vos classes ou une interface ; un type qui est connu de l'appelant. Si le bean est un bean EJB session, le type est l'interface `@Local` ou une classe locale. Un bean peut avoir plusieurs types. Par exemple, le bean suivant a quatre types :

```
public class BookShop
      extends Business
      implements Shop<Book
> {
   ...
}
```

Les types sont `BookShop`, `Business` et `Shop<Book>`, ainsi que le type implicite`java.lang.Object`. (Notez qu'un type paramétré est un type valide).

Cependant, ce bean session n'a que les interfaces `BookShop`, `Auditable` et `java.lang.Object` comme types, puisque la classe `BookShopBean` n'est pas visible de l'appelant.

```
@Stateful
public class BookShopBean
      extends Business
      implements BookShop, Auditable {
   ...
}
```

Note

The bean types of a session bean include local interfaces and the bean class local view (if any). EJB remote interfaces are not considered bean types of a session bean. You can't inject an EJB

using its remote interface unless you define a *resource*, which we'll meet in *Chapitre 14, Java EE component environment resources*.

Les types peuvent être restreints à un ensemble explicite en annotant le bean avec l'annotation `@Typed` et en listant les classes qui doivent être des types. Par exemple, les types de ce bean ont été réduits à `Shop<Book>`, ainsi que `java.lang.Object`:

```
@Typed(Shop.class)
public class BookShop
      extends Business
      implements Shop<Book
> {
   ...
}
```

Parfois, un seul type ne suffit pas au conteneur pour déterminer quel bean injecter. Par exemple, supposons que nous avons deux implémentations de l'interface `PaymentProcessor` : `CreditCardPaymentProcessor` et `DebitPaymentProcessor`. Injecter le type `PaymentProcessor` introduit une condition ambiguë. Dans ces cas là, le client doit spécifier une information supplémentaire pour préciser par quelle implémentation il est intéressé. Nous modélisons ce type d'information en utilisant un qualifiant.

Un qualifiant est une annotation personnalisée qui est définit elle-même avec l'annotation `@Qualifer`. Une annotation qualifiante est une extension du système de type. Elle permet d'enlever l'ambigüité sur un type sans avoir à gérer des noms en chaîne de caractères. Ceci est un exemple d'annotation qualifiante :

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

Vous n'êtes peut être pas à l'aise avec les définitions d'annotation. En fait, c'est peut être la première fois que vous en rencontrez une. Avec CDI, les définitions d'annotation vont devenir familiers au fur et à mesure que vous en créerez.

Note

Remarquez les noms des annotations fournies par CDI et EJB. Vous verrez que ce sont souvent des adjectifs. Nous vous encourageons à suivre cette convention lorsque vous créez vos propre annotations, puisqu'elles servent à décrire les comportements et les rôles de la classe.

Maintenant que nous avons définit une annotation qualifiante, nous pouvons l'utiliser pour lever l'ambigüité sur une injection. L'injection suivante a le type `PaymentProcessor` et le qualifiant `@CreditCard` :

```
@Inject @CreditCard PaymentProcessor paymentProcessor
```

Pour chaque injection, le conteneur recherche un bean qui satisfait le contrat, c'est à dire un bean qui a le bon type et tous les qualifiants. S'il trouve exactement un bean répondant à ces conditions, il injecte une instance de ce bean. Sinon, il remonte une erreur à l'utilisateur.

Comment spécifions-nous les qualifiants d'un bean ? En annotant la classe, bien sûr ! Le bean suivant a le qualifiant `@CreditCard` et implémente le type `PaymentProcessor`. Par conséquent, il satisfait notre injection qualifiée.

```
@CreditCard
public class CreditCardPaymentProcessor
    implements PaymentProcessor { ... }
```

Note

Si un bean ou un point d'injection ne spécifie pas explicitement un qualifiant, il a le qualifiant par défaut, `@Default`.

That's not quite the end of the story. CDI also defines a simple *resolution rule* that helps the container decide what to do if there is more than one bean that satisfies a particular contract. We'll get into the details in *Chapitre 4, Injection de dépendances et recherche programmatique*.

2.1.2. Portée (Scope)

The *scope* of a bean defines the lifecycle and visibility of its instances. The CDI context model is extensible, accommodating arbitrary scopes. However, certain important scopes are built into the specification, and provided by the container. Each scope is represented by an annotation type.

Par exemple, toutes les applications web peuvent avoir un bean de *portée session* :

```
public @SessionScoped
class ShoppingCart implements Serializable { ... }
```

Une instance portée session est liée à la session de l'utilisateur et est partagée par toutes les requêtes qu'il exécute dans le contexte de cette session.

Note

Gardez à l'esprit qu'une fois qu'un bean est lié à un contexte, il reste dans ce contexte jusqu'à ce que le contexte soit détruit. Il n'y a aucun moyen de supprimer un bean d'un contexte manuellement. Si vous ne voulez pas que le bean demeure dans la session indéfiniment, vous devez utiliser une autre portée avec une durée de vie plus courte, comme les portées "requête" ou "conversation".

Si une portée n'est pas explicitement spécifiée, alors le bean a une portée spéciale appelée *dependent pseudo-scope*. Les beans de cette portée vivent pour servir l'objet dans lequel ils ont été injectés, ce qui veut dire que leur cycle de vie est lié à celui de cet objet.

We'll talk more about scopes in *Chapitre 5, Portées et contextes*.

2.1.3. Nom EL

Si vous voulez référencer un bean dans du code non java qui supporte les expressions Unified EL, par exemple dans une page JSP ou JSF, vous devez donner à ce bean un *nom EL*.

Le nom EL est spécifié en utilisant l'annotation `@Named` comme suit :

```
public @SessionScoped @Named("cart")
class ShoppingCart implements Serializable { ... }
```

Maintenant nous pouvons l'utiliser facilement dans n'importe quelle page JSP ou JSF :

```
<h:dataTable value="#{cart.lineItems}" var="item">
   ...
</h:dataTable
>
```

Note

L'annotation `@Named` n'est pas ce qui fait de la classe un bean. La plupart des classes dans une archive bean sont déjà reconnues comme des beans. L'annotation `@Named` rend juste possible de référencer le bean depuis EL, généralement dans une vue JSF.

Nous pouvons laisser CDI choisir pour nous le nom en laissant vide la valeur de l'annotation `@Named` :

```
public @SessionScoped @Named
class ShoppingCart implements Serializable { ... }
```

Le nom par défaut est le nom de la classe non qualifié, sans majuscule à la première lettre ; dans ce cas, `shoppingCart`.

2.1.4. Alternatives

Nous avons déjà vu comment les qualifiants nous permettent de choisir entre plusieurs implémentations d'une interface pendant le développement. Mais parfois nous avons une interface (ou un autre type) dont l'implémentation varie en fonction de l'environnement du déploiement. Par exemple, nous pouvons vouloir nous servir d'un mock dans un environnement de test. Une *alternative* peut être déclarée en annotant la classe avec l'annotation `@Alternative`.

```
public @Alternative
class MockPaymentProcessor extends PaymentProcessorImpl { ... }
```

Normalement, on annote un bean avec `@Alternative` seulement quand il y a une autre implémentation d'une interface qu'il implémente (ou n'importe lequel de ses types). On peut choisir entre les alternatives au déploiement en *sélectionnant* une alternative dans le descripteur de déploiement CDI `META-INF/beans.xml` du jar ou du module Java EE qui l'utilise. Chaque module peut spécifier une alternative différente à utiliser.

We cover alternatives in more detail in *Section 4.7, « Alternatives »*.

2.1.5. Types de connexion d'intercepteur (Interceptor binding)

Vous être peut être familier avec l'utilisation d'intercepteurs dans EJB 3.0. Dans Java EE 6, cette fonctionnalité a été généralisée pour fonctionner avec d'autres beans gérés. Tout à fait, vous n'avez plus à faire de votre bean un EJB juste pour intercepter ces méthodes. Youhouuu. Donc qu'est-ce que CDI apporte de plus que ça ? Eh bien, pas mal de choses en fait. Voyons quelques notions.

La façon dont les intercepteurs étaient définis en Java EE 5 était contre intuitive. Vous deviez spécifier l'*implémentation* de l'intercepteur directement dans l'*implementation* de l'EJB, soit par l'annotation `@Interceptors` soit dans le descripteur XML. Autant mettre le code de l'intercepteur *dans* l'implémentation ! Deuxièmement, l'ordre dans lequel les intercepteurs sont appelés dépend de l'ordre dans lequel ils sont déclarés dans l'annotation `@Interceptors` ou dans le descripteur XML. Peut être que ce n'est pas si mauvais si vous appliquez les intercepteurs à un seul bean. Mais, si vous les appliquez plusieurs fois, alors il y a de bonnes chances que vous définissiez par inadvertance un ordre différent dans différents beans.

CDI fournit une nouvelle approche pour connecter les intercepteurs aux beans qui introduit un niveau d'indirection (et donc un contrôle). Nous devons définir un *type de connexion d'intercepteur* (Interceptor binding type) qui décrit le comportement implémenté par l'intercepteur.

Un type de connexion d'intercepteur est une annotation personnalisée qui est elle-même annotée `@InterceptorBinding`. Cela nous permet d'éviter une dépendance directe entre les classes des intercepteurs et les classes des beans.

```
@InterceptorBinding
@Inherited
@Target( { TYPE, METHOD })
@Retention(RUNTIME)
public @interface Transactional {}
```

L'intercepteur qui implémente la gestion de la transaction déclare cette annotation :

```
public @Transactional @Interceptor
class TransactionInterceptor { ... }
```

Nous pouvons connecter l'intercepteur à un bean en annotant la classe avec le même type de connexion d'intercepteur :

```
public @SessionScoped @Transactional
class ShoppingCart implements Serializable { ... }
```

Notons que `ShoppingCart` et `TransactionInterceptor` n'ont pas connaissance l'un de l'autre.

Les intercepteurs sont spécifiques au déploiement. (Nous n'avons pas besoin d'un `TransactionInterceptor` dans nos tests unitaires !). Par défaut, un intercepteur est désactivé. Nous pouvons activer un intercepteur en utilisant le descripteur de déploiement CDI `META-INF/beans.xml` du jar ou du module Java EE. C'est aussi là que nous spécifions l'ordre des intercepteurs.

We'll discuss interceptors, and their cousins, decorators, in *Chapitre 9, Intercepteurs* and *Chapitre 10, Décorateurs*.

2.2. Quelles sortes de classes sont les beans ?

Nous avons déjà vu deux types de beans : les JavaBeans et les EJBs session. C'en est fini ? En fait, ce n'est que le début. Explorons les différentes sortes de beans que les implémentations CDI doivent gérer nativement.

2.2.1. Les managed beans

Un managed bean est une classe Java. Le cycle de vie basique et la sémantique d'un managed bean sont définis par la spécification des Managed Beans. Vous pouvez déclarer explicitement un managed bean en annotant sa classe `@ManagedBean`, mais avec CDI vous n'avez pas à le faire. Conformément à la spécification, le conteneur CDI considère comme managed bean toutes les classes qui satisfont les conditions suivantes :

- Ce n'est pas une classe interne non statique.
- C'est une classe contrète, ou annotée `@Decorator`.
- Elle n'est pas annotée avec une annotation définissant un EJB ou déclarée comme une classe EJB dans le fichier `ejb-jar.xml`.
- Elle n'implémente pas `javax.enterprise.inject.spi.Extension`.
- Elle a constructeur approprié — soit :
  - un constructeur sans paramètre, soit
  - un constructeur annoté `@Inject`.

Note

Suivant cette définition, les entités JPA sont des techniquement des managed beans. Cependant, les entités ont leur propre cycle de vie, état et modèle d'identité et sont généralement instanciées par JPA ou en utilisant `new`. Par conséquent, nous ne recommandons pas d'injecter directement une entité. Nous décourageons tout particulièrement d'assigner une portée autre que `@Dependent` à une entité, puisque JPA n'est pas capable de persister les proxies CDI injectés.

L'ensemble non restreint des types pour un managed bean contient la classe, toute superclasse et toutes les interfaces qu'il implémente directement ou indirectement.

Si un managed bean a un attribut public, il doit avoir la portée par défaut `@Dependent`.

Les managed beans supportent les annotations de cycle de vie `@PostConstruct` et `@PreDestroy`.

Les beans session sont aussi, techniquement, des managed beans. Cependant, puisqu'ils ont leur propre cycle de vie et tirent avantage de services supplémentaires de Java EE, la spécification CDI les considère comme des beans différents.

2.2.2. Les beans session

Les beans session appartiennent à la spécification des EJBs. Ils ont un cycle de vie spécial, une gestion d'état et un modèle de concurrence qui est différent des autres managed beans et des objets java non managés. Mais les beans session participe à CDI comme n'importe quel autre bean. Vous pouvez injecter un bean session dans un autre bean session, un managed bean dans un bean session, un bean session dans un managed bean, avoir un managed bean observant un évènement lancé par un bean session, etc.

Note

Les message-driven beans et les entités sont par nature des objets non contextuels et ne peuvent être injectés dans d'autres objets. Cependant, les message-driven beans peuvent tirer avantage de certaines fonctionnalités CDI, comme l'injection de dépendances, les intercepteurs et les décorateurs. En fait, CDI réalisera l'injection dans n'importe quel bean session ou message-driven, même ceux qui ne sont pas des instances contextuelles.

L'ensemble non restreint des types pour un bean session contient toutes les interfaces locales du bean et toutes leurs superinterfaces. Si le bean session est visible localement, l'ensemble non restreint des types contient la classe du bean et toutes les superclasses. En plus, `java.lang.Object` est le type de tous les beans sessions. Mais les interfaces distantes *ne sont pas* inclues dans l'ensemble des types.

Il n'y a pas de raison de déclarer explicitement la portée d'un bean sans état ou d'un bean session singleton. Le conteneur EJB contrôle le cycle de vie de ces beans, conformément à la sémantique des déclarations `@Stateless` ou `@Singleton`. D'un autre côté, un bean session avec état peut ne pas avoir de portée.

Les beans session avec état peuvent définir une *méthode de suppression*, annotée `@Remove`, qui est utilisée par l'application pour indiquer qu'une instance devrait être supprimée. Cependant, pour une instance contextuelle du bean —une instance contrôlée par CDI— cette méthode ne pourra être appelée par l'application que si le bean a la portée `@Dependent`. Pour les beans avec d'autres portées, l'application doit laisser le conteneur détruire le bean.

Donc, quand devons-nous utiliser un bean session au lieu d'un simple managed bean ? A chaque fois que nous avons besoin de services Java EE offerts par les EJB, comme :

- la gestion de transaction et de la sécurité au niveau de la méthode,
- la gestion de la concurrence,
- la passivation au niveau de l'instance pour les beans session avec état et le pooling d'instance pour les beans session sans état,
- l'invocation de service web ou distant, ou
- les timers et les méthodes asynchrones.

Quand nous n'avons besoin de rien de tout cela, un managed bean ordinaire fera très bien l'affaire.

Beaucoup de beans (y compris les beans `@SessionScoped` ou `@ApplicationScoped`) sont disponibles pour un accès concurrent. Donc, la gestion de la concurrence fournie par EJB 3.1 est particulièrement utile. La plupart des beans de portée session et application devraient être des EJBs.

Les beans qui ont des références vers des ressources bas niveau, ou disposent de beaucoup d'états internes bénéficient du cycle de vie géré par le conteneur défini par le modèle d'EJB avec état/sans état/singleton, avec son support de la passivation et du pooling d'instance.

Au final, c'est généralement évident quand on a besoin de gestion de transaction au niveau de la méthode, de sécurité au niveau de la méthode, de timers, de méthodes distantes ou asynchrones.

Ce que nous essayons de dire, c'est d'utiliser un bean session quand vous avez besoin des services qu'il fournit, pas juste parce que vous voulez utiliser l'injection de dépendance, la gestion du cycle de vie ou des intercepteurs. Java EE 6 fournit un modèle de programmation progressif. Il est généralement facile de commencer avec un managed bean ordinaire et, plus tard, de le transformer en EJB juste en ajoutant l'une des annotation suivantes : `@Stateless`, `@Stateful` ou `@Singleton`.

D'un autre côté, n'ayez pas peur d'utiliser des beans session juste parce que vous avez entendu vos amis dire qu'ils sont "lourds". Ce n'est que de la superstition de croire que quelque chose est "plus lourd" juste parce qu'il est hébergé nativement par le conteneur Java EE, plutôt que par un conteneur propriétaire de beans ou par un framework d'injection de dépendances qui ajoute une couche d'obfuscation. D'une manière générale, il faut être sceptique des gens qui utilisent une terminologie vague comme "lourd".

2.2.3. Méthodes de production

Tout ce que nous devons injecter ne peut se résumer à un bean instancié par le conteneur en utilisant `new`. Il y a plein de cas où nous avons besoin d'un contrôle supplémentaire. Comment faire si nous devons décider à l'exécution quelle implémentation d'un type instancier et injecter ? Comment faire si nous devons injecter un objet qui est obtenu par une requête à un service ou une ressource transactionnelle, par exemple en lançant une requête JPA ?

Une *méthode de production* est une méthode qui agit comme une source d'instances. La déclaration de la méthode elle-même décrit le bean et le conteneur invoque cette méthode pour obtenir une instance quand aucune instance n'existe pour le contexte spécifié. Une méthode de production laisse l'application prendre le contrôle total du processus d'instanciation du bean.

Une méthode de production est déclatée en annotant une méthode d'une classe avec l'annotation `@Produces`.

```
@ApplicationScoped
public class RandomNumberGenerator {

   private Random random = new Random(System.currentTimeMillis());

   @Produces @Named @Random int getRandomNumber() {
      return random.nextInt(100);
   }

}
```

Nous ne pouvons pas écrire une classe qui est elle-même un nombre aléatoire. Mais nous pouvons certainement écrire une méthode qui retourne un nombre aléatoire. En faisant de la méthode une méthode de production, nous autorisons la valeur de retour de la méthode — dans ce cas un `Integer`— à être injectée. Nous pouvons même spécifier un qualifiant —in this case `@Random`, une portée—qui dans ce cas est par défaut à `@Dependent`, et un nom EL—qui est dans ce cas par défaut à `randomNumber` en suivant la convention de nommage des propriétés JavaBeans. Maintenant nous pouvons obtenir un nombre aléatoire n'importe où :

```
@Inject @Random int randomNumber;
```

Même dans une expression Unified EL :

```
<p
>Your raffle number is #{randomNumber}.</p
>
```

Une méthode de production ne doit pas être une méthode abstraite d'un managed bean ou d'un bean session. Une méthode de production peut-être statique ou non. Si le bean est un bean session, la méthode de production doit être soit un méthode métier de l'EJB, soit un méthode statique de la classe.

Le type de la méthode de production est dépendant du type de retour :

- If the return type is an interface, the unrestricted set of bean types contains the return type, all interfaces it extends directly or indirectly and `java.lang.Object`.
- If a return type is primitive or is a Java array type, the unrestricted set of bean types contains exactly two types: the method return type and `java.lang.Object`.
- If the return type is a class, the unrestricted set of bean types contains the return type, every superclass and all interfaces it implements directly or indirectly.

Note

Les méthodes de production et les attributs peuvent être de type primitif. Afin de pouvoir resoudre les dépendances, les types primitifs sont considérés comment étant identiques à leur correspondance objet se trouvant dans `java.lang`.

If the producer method has method parameters, the container will look for a bean that satisfies the type and qualifiers of each parameter and pass it to the method automatically—another form of dependency injection.

```
@Produces Set<Roles
> getRoles(User user) {
   return user.getRoles();
}
```

We'll talk much more about producer methods in *Chapitre 8, Méthodes de production*.

2.2.4. Les attributs de production

Un *attribut de production* est une alternative plus simple aux méthodes de production. Un attribut de production est déclaré en annotant un attribut d'une classe avec l'annotation `@Produces`—la même annotation que pour les méthodes de production.

```
public class Shop {
   @Produces PaymentProcessor paymentProcessor = ....;
   @Produces @Catalog List<Product
> products = ....;
}
```

Les règles pour déterminer le type d'un attribut de production suivent les mêmes règles que pour les méthodes de production.

A producer field is really just a shortcut that lets us avoid writing a useless getter method. However, in addition to convenience, producer fields serve a specific purpose as an adaptor for Java EE component environment injection, but to learn more about that, you'll have to wait until *Chapitre 14, Java EE component environment resources*. Because we can't wait to get to work on some examples.

Exemple d'une web application JSF

Let's illustrate these ideas with a full example. We're going to implement user login/logout for an application that uses JSF. First, we'll define a request-scoped bean to hold the username and password entered during login, with constraints defined using annotations from the Bean Validation specification:

```
@Named @RequestScoped
public class Credentials {
    private String username;
    private String password;

    @NotNull @Length(min=3, max=25)
    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    @NotNull @Length(min=6, max=20)
    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }
}
```

Ce bean est lié à l'invite de connexion dans le formulaire JSF suivant :

```
<h:form>
   <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
      <f:validateBean>
         <h:outputLabel for="username"
>Username:</h:outputLabel>
         <h:inputText id="username" value="#{credentials.username}"/>
         <h:outputLabel for="password"
>Password:</h:outputLabel>
         <h:inputSecret id="password" value="#{credentials.password}"/>
      </f:validateBean>
   </h:panelGrid>
   <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}"/>
   <h:commandButton value="Logout" action="#{login.logout}" rendered="#{login.loggedIn}"/>
</h:form
>
```

Les utilisateurs sont représentés par une entité JPA :

```
@Entity
public class User {
   private @NotNull @Length(min=3, max=25) @Id String username;
   private @NotNull @Length(min=6, max=20) String password;

   public String getUsername() { return username; }
   public void setUsername(String username) { this.username = username; }
   public String setPassword(String password) { this.password = password; }
}
```

(Notez que nous aurons aussi besoin d'un fichier `persistence.xml` pour configurer l'unité de persistance JPA contenant `User`.)

The actual work is done by a session-scoped bean that maintains information about the currently logged-in user and exposes the `User` entity to other beans:

```
@SessionScoped @Named
public class Login implements Serializable {

   @Inject Credentials credentials;
   @Inject @UserDatabase EntityManager userDatabase;

   private User user;

   public void login() {
      List<User
> results = userDatabase.createQuery(
         "select u from User u where u.username = :username and u.password = :password")
         .setParameter("username", credentials.getUsername())
         .setParameter("password", credentials.getPassword())
         .getResultList();

      if (!results.isEmpty()) {
         user = results.get(0);
      }
      else {
         // perhaps add code here to report a failed login
      }
   }

   public void logout() {
      user = null;
   }

   public boolean isLoggedIn() {
      return user != null;
   }

   @Produces @LoggedIn User getCurrentUser() {
      return user;
   }

}
```

`@LoggedIn` and `@UserDatabase` are custom qualifier annotations:

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

```
@Qualifier
@Retention(RUNTIME)
```

```
@Target({METHOD, PARAMETER, FIELD})
public @interface UserDatabase {}
```

We need an adaptor bean to expose our typesafe `EntityManager`:

```
class UserDatabaseProducer {
   @Produces @UserDatabase @PersistenceContext
   static EntityManager userDatabase;
}
```

Maintenant `DocumentEditor`, ou n'importe quel autre bean, peut facilement injecter l'utilisateur courant :

```
public class DocumentEditor {
   @Inject Document document;
   @Inject @LoggedIn User currentUser;
   @Inject @DocumentDatabase EntityManager docDatabase;

   public void save() {
      document.setCreatedBy(currentUser);
      docDatabase.persist(document);
   }
}
```

Ou nous pouvons référencer l'utilisateur courant dans une vue JSF :

```
<h:panelGroup rendered="#{login.loggedIn}">
   signed in as #{currentUser.username}
</h:panelGroup
>
```

Espérons que cet exemple vous a donné un avant-goût du modèle de programmation CDI. Dans le prochain chapitre, nous explorerons plus en détails l'injection de dépendances.

Injection de dépendances et recherche programmatique

L'une des fonctionnalités les plus significatives de CDI —certainement la plus reconnue— est l'injection de dépendances ; pardon, l'injection de dépendances *fortement typée*.

4.1. Points d'injection

L'annotation `@Inject` nous permet de définir un point d'injection qui est injecté pendant la création d'un bean. L'injection peut se produire par trois mécanismes différents.

Injection de paramètres dans un *constructeur de bean* :

```
public class Checkout {

   private final ShoppingCart cart;

   @Inject
   public Checkout(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

Un bean ne peut avoir qu'un seul constructeur injectable.

Injection de paramètres dans une *méthode d'initialisation* :

```
public class Checkout {

   private ShoppingCart cart;

   @Inject
   void setShoppingCart(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

Note

Un bean peut avoir plusieurs méthodes d'initialisation. Si le bean est un bean de session, la méthode d'initialisation n'a pas besoin d'être une méthode métier du bean de session.

Et l'injection directe d'attributs :

```
public class Checkout {

   private @Inject ShoppingCart cart;

}
```

Note

Les méthodes getter et setter ne sont pas réquises pour le fonctionnement de l'injection d'attributs (contrairement aux beans managés JSF).

L'injection de dépendances se produit toujours lorsque l'instance du bean est d'abord instanciée par le conteneur. Simplifions un peu, les choses se passent dans cet ordre :

- D'abord, le conteneur appelle le constructeur du bean (le constructeur par défaut or celui annoté avec `@Inject`), pour obtenir une instance d'un bean.
- Ensuite, le conteneur initialise les valeurs de tous les attributs injectés du bean.
- Ensuite, le conteneur appelle toutes les méthodes d'initialisation du bean (l'ordre d'appel n'est pas portable, ne comptez pas dessus).
- Enfin, la méthode `@PostConstruct`, s'il en a une, est appelée.

(La seule complication est que le conteneur peut appeler des méthodes d'initialisation d'une super classe avant d'initialiser les attributs injectés par la sous classe.)

Note

L'un des avantages majeur de l'injection par constructeur est que cela permet que le bean soit immutable.

CDI supporte aussi l'injection de paramètres pour quelques autres méthodes qui sont appelées par le conteneur. Par exemple, l'injection de paramètres est supportée pour les méthodes de production :

```
@Produces Checkout createCheckout(ShoppingCart cart) {
    return new Checkout(cart);
}
```

This is a case where the `@Inject` annotation *is not* required at the injection point. The same is true for observer methods (which we'll meet in*Chapitre 11, Evènements*) and disposer methods.

4.2. Ce qui est injecté

The CDI specification defines a procedure, called*typesafe resolution*, that the container follows when identifying the bean to inject to an injection point. This algorithm looks complex at first, but once you understand it, it's really quite intuitive. Typesafe resolution is performed at system initialization time, which means that the container will inform the developer immediately if a bean's dependencies cannot be satisfied.

Le but de cet algorithme est de permettre à plusieurs beans d'implémenter la même interface et soit :

- de permettre au client à sélectionner quelle implémentation il souhaite utiliser en utilisant un *qualifiant* ou
- allow the application deployer to select which implementation is appropriate for a particular deployment, without changes to the client, by enabling or disabling an*alternative*, or
- en permettant aux beans d'être isolés dans des modules séparés.

Evidemment, si vous avez exactement un bean d'un type donné, et un point d'injection avec le même type, alors le bean A ira dans l'emplacement A. C'est le scénario le plus simple possible. Quand vous démarrez votre application pour la première fois, vous devriez en avoir plusieurs.

Mais ensuite, les choses peuvent commencer à être compliqué. Voyons maintenant comment le conteneur détermine quel bean injecter dans des cas plus compliqués. Nous commencerons par regarder plus attentivement les qualifiants.

4.3. Annotations de qualification

Si nous avons plus d'un bean qui implémente un type de bean particulier, le point d'injection peut spécifier exactement quel bean devrait être injecté en utilisant une annotation de qualification. Par exemple, il peut y avoir deux implémentations de `PaymentProcessor` :

```
@Synchronous
public class SynchronousPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

```
@Asynchronous
public class AsynchronousPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

Où `@Synchronous` et `@Asynchronous` sont des annotations qualifiantes :

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

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

Un développeur ayant besoin d'un bean peut utiliser l'annotation qualifiante pour spécifier exactement quel bean doit être injecté.

En utilisant l'injection d'attribut :

```
@Inject @Synchronous PaymentProcessor syncPaymentProcessor;
@Inject @Asynchronous PaymentProcessor asyncPaymentProcessor;
```

En utilisant l'injection de méthode d'initialisation :

```
@Inject
public void setPaymentProcessors(@Synchronous PaymentProcessor syncPaymentProcessor,
                                 @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   this.syncPaymentProcessor = syncPaymentProcessor;
   this.asyncPaymentProcessor = asyncPaymentProcessor;
}
```

En utilisant l'injection de constructeur :

```
@Inject
public Checkout(@Synchronous PaymentProcessor syncPaymentProcessor,
                @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   this.syncPaymentProcessor = syncPaymentProcessor;
   this.asyncPaymentProcessor = asyncPaymentProcessor;
}
```

Les annotations de qualification peuvent aussi qualifier les arguments des méthodes de production, de libération et d'observation. Combiner les arguments qualifiés avec les méthodes de production est un bon moyen d'avoir une implémentation d'un bean choisi à l'exécution en fonction de l'état du système.

```
@Produces
PaymentProcessor getPaymentProcessor(@Synchronous PaymentProcessor syncPaymentProcessor,
                                     @Asynchronous PaymentProcessor asyncPaymentProcessor) {
   return isSynchronous() ? syncPaymentProcessor : asyncPaymentProcessor;
}
```

If an injected field or a parameter of a bean constructor or initializer method is not explicitly annotated with a qualifier, the default qualifier,`@Default`, is assumed.

Now, you may be thinking, *"What's the different between using a qualifier and just specifying the exact implementation class you want?"* It's important to understand that a qualifier is like an extension of the interface. It does not create a direct dependency to any particular implementation. There may be multiple alternative implementations of`@Asynchronous PaymentProcessor`!

4.4. Les qualifiants intégrés `@Default` et `@Any`

Whenever a bean or injection point does not explicitly declare a qualifier, the container assumes the qualifier`@Default`. From time to time, you'll need to declare an injection point without specifying a qualifier. There's a qualifier for that too. All beans have the qualifier `@Any`. Therefore, by explicitly specifying `@Any` at an injection point, you suppress the default qualifier, without otherwise restricting the beans that are eligible for injection.

Ceci est particulièrement utile si vous voulez effectuer une itération sur tous les beans d'un certain type. Par exemple :

```
@Inject
void initServices(@Any Instance<Service
> services) {
   for (Service service: services) {
      service.init();
   }
}
```

4.5. Qualifiants avec des membres

Les annotations Java peuvent avoir des membres. Nous pouvons utiliser les membres d'annotation pour distinguer d'avantage un qualifiant. Cela limite d'explosion du nombre de nouvelles annotations. Par exemple, au lieu de créer plusieurs qualifiants représentant différentes méthodes de paiement, vous pouvez les regrouper dans une seule annotation avec un membre :

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PayBy {
   PaymentMethod value();
}
```

Ensuite, nous sélectionnons l'une des valeurs possibles pour le membre lorsque nous appliquons le qualifiant :

```
private @Inject @PayBy(CHECK) PaymentProcessor checkPayment;
```

Vous pouvez dire au conteneur d'ignorer un membre d'une annotation de qualification en annotant ce membre `@NonBinding`.

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface PayBy {
   PaymentMethod value();
   @Nonbinding String comment() default "";
}
```

4.6. Qualifiants multiples

Un point d'injection peut spécifier plusieurs annotations de qualification :

```
@Inject @Synchronous @Reliable PaymentProcessor syncPaymentProcessor;
```

Alors, seul un bean qui porte *les deux* annotations de qualification est candidat pour l'injection.

```
@Synchronous @Reliable
public class SynchronousReliablePaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

4.7. Alternatives

Alternatives are beans whose implementation is specific to a particular client module or deployment scenario. This alternative defines a mock implementation of both `@Synchronous PaymentProcessor` and`@Asynchronous PaymentProcessor`, all in one:

```
@Alternative @Synchronous @Asynchronous
public class MockPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

Par défaut, les beans `@Alternative` sont désactivés. Nous devons *activer* une alternative dans le fichier de description `beans.xml` pour le rendre disponible à l'instanciation et l'injection. Cette activation s'applique uniquement aux beans de cette archive.

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <alternatives>
         <class
>org.mycompany.mock.MockPaymentProcessor</class>
   </alternatives>
</beans
>
```

Quand une dépendance ambiguë existe sur le point d'injection, le conteneur tente de résoudre l'ambiguïté en cherchant une alternative activée parmi les beans qui peuvent être injectés. S'il existe qu'une et une seule alternative activée, il s'agit du bean qui sera injecté.

4.8. Résoudre les dépendances insatisfaites et ambiguës :

The typesafe resolution algorithm fails when, after considering the qualifier annotations on all beans that implement the bean type of an injection point and filtering out disabled beans (`@Alternative` beans which are not explicitly enabled), the container is unable to identify exactly one bean to inject. The container will abort deployment, informing us of the unsatisfied or ambiguous dependency.

Au cours de votre développement, vous allez rencontrer cette situation. Voyons comment la résoudre.

To fix an*unsatisfied dependency*, either:

- créez un bean qui implémente le type et qui a tous les qualifiants du point d'injection,
- soyez sur que le bean que vous avez déjà est dans le classpath est dans le module du point d'injection, ou
- explicitly enable an `@Alternative` bean that implements the bean type and has the appropriate qualifier types, using`beans.xml`.

To fix an*ambiguous dependency*, either:

- ajoutez un qualifiant pour distinguer les deux implémentations l'une de l'autre,
- disable one of the beans by annotating it`@Alternative`,
- déplacez l'une des implémentations dans un module qui n'est pas dans le classpath du module ayant le point d'injection, ou
- disable one of two `@Alternative` beans that are trying to occupy the same space, using`beans.xml`.

See *this FAQ* [http://sfwk.org/Documentation/HowDoAResolveAnAmbiguousResolutionExceptionBetweenAProducerMethodAndARawType] for step-by-step instructions for how to resolve an ambiguous resolution exception between a raw bean type and a producer method that returns the same bean type.

Souvenez-vous juste : "Il ne peut en rester qu'un."

On the other hand, if you really do have an optional or multivalued injection point, you should change the type of your injection point to`Instance`, as we'll see in*Section 4.10, « Obtenir une instance contextuelle par recherche programmatique »*.

Maintenant, il y a un problème supplémentaire auquel vous devrez être attentif lorsque vous utiliserez le service d'injection de dépendances.

4.9. Proxies client

Clients of an injected bean do not usually hold a direct reference to a bean instance, unless the bean is a dependent object (scope`@Dependent`).

Imaginez qu'un bean de portée application tienne un référence directe d'un bean de portée requête. Le bean de portée application est partagé avec les différentes requêtes. Cependant, chaque requête devrait voir une instance différente du bean de portée requête—celle courante !

Maintenant, imaginez qu'un bean de portée session porte une référence directe d'un bean de portée application. De temps en temps, le contexte de session est sérialisé sur le disque afin d'utiliser la mémoire efficacement. Cependant, l'instance du bean de portée application ne devrait pas être sérialisé avec le bean de portée session ! On peut avoir besoin de cette référence à n'importe quel moment. Pas besoin de le mettre de coté !

Therefore, unless a bean has the default scope`@Dependent`, the container must indirect all injected references to the bean through a proxy object. This *client proxy* is responsible for ensuring that the bean instance that receives a method invocation is the instance that is associated with the current context. The client proxy also allows beans bound to contexts such as the session context to be serialized to disk without recursively serializing other injected beans.

Malheureusement, à cause de limitations du langage Java, certains types Java ne peuvent être proxifié par le conteneur. Si un point d'injection déclaré avec l'un de ces types est résolu par un bean avec tout autre portée que `@Dependent`, le conteneur annulera le déploiement, en nous informant du problème.

Les types Java suivants ne peuvent pas être proxifiés par le conteneur :

- les classes qui n'ont pas un constructeur non privée et sans paramètre, et
- les classes qui sont déclarées `final` ou qui ont une méthode `final`,
- les tableaux et les types primitifs.

Il est généralement très simple de régler un problème de dépendance non proxifiable. Si le point d'injection de type X donne résultat à une dépendance non proxifiable, il suffit simplement de :

- add a constructor with no parameters toX,
- change the type of the injection point to`Instance<X>`,
- introduce an interfaceY, implemented by the injected bean, and change the type of the injection point toY, or
- if all else fails, change the scope of the injected bean to`@Dependent`.

Note

Une prochaine version de Weld prendra en charge ces limitations, en utilisant des APIs non portables de JVM :

- Sun, IcedTea, Mac : `Unsafe.allocateInstance()` (Le plus efficace)
- IBM, JRockit : `ReflectionFactory.newConstructorForSerialization()`

Mais nous n'avons pas encore pu contourner le problème en implémentant ça.

4.10. Obtenir une instance contextuelle par recherche programmatique

Dans certaines situations, l'injection n'est pas le moyen le plus pratique pour obtenir une référence contextuelle. Par exemple, elle ne peut pas être utilisée quand :

- le type du bean ou des qualifiants changent dynamiquement à l'exécution, ou
- selon le déploiement, il peut y avoir aucun bean qui satisfasse le type et les qualifiants, ou
- nous aimerions itérer sur tous les beans d'un type donné.

In these situations, the application may obtain an instance of the interface`Instance`, parameterized for the bean type, by injection:

```
@Inject Instance<PaymentProcessor
> paymentProcessorSource;
```

La méthode `get()` d'`Instance` produit une instance contextuelle du bean.

```
PaymentProcessor p = paymentProcessorSource.get();
```

Les qualifiants peuvent être spécifiés de deux moyens :

- en annotant le point d'injection `Instance`, ou
- by passing qualifiers to the `select()` of`Event`.

Spécifier les qualifiants sur le point d'injection est beaucoup, beaucoup plus simple :

```
@Inject @Asynchronous Instance<PaymentProcessor
> paymentProcessorSource;
```

Maintenant, le `PaymentProcessor` retourné par `get()` aura la qualifiant `@Asynchronous`.

Alternatively, we can specify the qualifier dynamically. First, we add the `@Any` qualifier to the injection point, to suppress the default qualifier. (All beans have the qualifier`@Any` .)

```
@Inject @Any Instance<PaymentProcessor
> paymentProcessorSource;
```

Ensuite, nous avons besoin d'obtenir une instance de notre type qualifié. Puisque les annotations sont des interfaces, nous pouvons juste écrire `new Asynchronous()`. Il est assez fastidieux de créer une implémentation concrète d'une annotation depuis zéro. A la place, CDI nous permet d'obtenir une instance qualifiant en surchargeant la classe helper `AnnotationLiteral`.

```
class AsynchronousQualifier
extends AnnotationLiteral<Asynchronous
> implements Asynchronous {}
```

Dans certains cas, nous pouvons utiliser une classe anonyme :

```
PaymentProcessor p = paymentProcessorSource
   .select(new AnnotationLiteral<Asynchronous
>() {});
```

Cependant, nous ne pouvons pas utiliser une classe anonyme pour implémenter un qualifiant avec des attributs.

Now, finally, we can pass the qualifier to the `select()` method of`Instance`.

```
Annotation qualifier = synchronously ?
      new SynchronousQualifier() : new AsynchronousQualifier();
PaymentProcessor p = anyPaymentProcessor.select(qualifier).get().process(payment);
```

4.11. L'objet `InjectionPoint`

There are certain kinds of dependent objects (beans with scope`@Dependent`) that need to know something about the object or injection point into which they are injected in order to be able to do what they do. For example:

- La catégorie de log pour un `Logger` dépend de la classe de l'objet qui le possède.
- L'injection d'un paramètre HTTP ou d'une valeur d'un header dépend du paramètre ou du nom de l'header qui est spécifié sur le point d'injection.
- L'injection du résultat d'une évaluation d'une expression EL dépend de l'expression qui est spécifiée sur le point d'injection.

Un bean avec la portée `@Dependent` peut injecter une instance de `InjectionPoint` et accèder aux méta-datas relatives au point d'injection pour lequel il appartient.

Jetons un coup d'oeil à un exemple. Le code suivant est verbeux, et vulnérable au refactoring :

```
Logger log = Logger.getLogger(MyClass.class.getName());
```

Cette intelligente petite méthode de production vous permet d'injecter un `Logger` du JDK sans spécifier explicitement la catégorie du log :

```
class LogFactory {

   @Produces Logger createLogger(InjectionPoint injectionPoint) {
      return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().getName());
   }

}
```

Nous pouvons maintenant écrire :

```
@Inject Logger log;
```

Toujours pas convaincu ? Alors voici un autre exemple. Pour injecter les paramètres HTTP, nous avons besoin de définir un qualifiant :

```
@BindingType
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface HttpParam {
   @Nonbinding public String value();
}
```

Nous voulons utiliser ce qualifiant sur les points d'injection suivants :

```
@HttpParam("username") String username;
@HttpParam("password") String password;
```

La méthode de production suivante fait le travail :

```
class HttpParams

   @Produces @HttpParam("")
   String getParamValue(InjectionPoint ip) {
      ServletRequest request = (ServletRequest) FacesContext.getCurrentInstance().getExternalContext().getRequest
      return request.getParameter(ip.getAnnotated().getAnnotation(HttpParam.class).value());
   }

}
```

Notez que l'obtention de la requête dans cet exemple est dédié à JSF. Pour une solution plus générique, vous pouvez écrire votre propre producteur de requête et l'injecter comme paramètre de la méthode.

Notez aussi que l'attribut `value()` de l'annotation `HttpParam` est ignoré par le conteneur puisqu'il est annoté `@Nonbinding.`

Le conteneur fournit nativement un bean qui implémente l'interface `InjectionPoint` :

```
public interface InjectionPoint {
   public Type getType();
   public Set<Annotation
> getQualifiers();
   public Bean<?> getBean();
   public Member getMember();
   public Annotated getAnnotated();
   public boolean isDelegate();
   public boolean isTransient();
}
```

Portées et contextes

Jusqu'à présent, nous avons vu quelques exemples d'*annotations de type de portée*. La portée d'un bean détermine le cycle de vie des instances du bean. La portée détermine aussi quels clients référencent quelles instances du bean. D'après les spécifications de CDI, une portée détermine :

- Quand une nouvelle instance de n'importe quel bean avec cette portée est créée
- Quand une instance existante de n'importe quel bean avec cette portée est détruite
- Quelles références injectées référencent n'importe quelle instance d'un bean avec cette portée

Par exemple, si vous avez un bean de portée session, `CurrentUser`, tous les beans qui sont appelés dans le contexte de la même `HttpSession` verra la même instance de `CurrentUser`. Cette instance sera automatiquement créée la première fois qu'un `CurrentUser` est nécessaire dans cette session, et automatiquement détruit quand la session se termine.

Note

Les entités JPA ne sont pas en bonne adéquation pour ce modèle. Les entités ont leur propre cycle de vie et modèle d'identité qui ne correspond pas naturellement au modèle utilisé par CDI. Cependant, nous recommandons de traiter les entités comme des beans CDI. Vous allez certainement rencontrer des problèmes si vous essayez de donner une portée à un bean autre que celle par défaut `@Dependent`. Le proxy client bloquera si vous essayez de passer une instance injectée à l'`EntityManager` JPA.

5.1. Type de portée

CDI comprend un *modèle de contexte extensible*. Il est possible de définir de nouvelles portées en créant une nouvelle annotation de type portée :

```
@ScopeType
@Retention(RUNTIME)
@Target({TYPE, METHOD})
public @interface ClusterScoped {}
```

Bien sur, il s'agit de la partie simple du travail. Pour que ce type de portée soit utile, nous avons aussi besoin de définir un objet `Context` qui implémente la portée ! Implémenter un `Context` est une tâche vraiment technique, prévu uniquement pour le développement de framework. Vous pouvez, par exemple, vous attendre à une implémentation de la portée business dans une prochaine version de Seam.

Nous pouvons appliquer une annotation de type de portée à une classe implémentant un bean pour spécifier la portée du bean :

```
@ClusterScoped
public class SecondLevelCache { ... }
```

Habituellement, nous utiliserons une des portées intégrées à CDI.

5.2. Portées intégrées

CDI définit quatre portées intégrées :

- `@RequestScoped`
- `@SessionScoped`
- `@ApplicationScoped`
- `@ConversationScoped`

Pour une application web qui utilise CDI :

- n'importe quelle requête servlet a accès aux portées actives requête, session et application, et, en plus
- n'importe quelle requête JSF a accès à la portée active conversation.

Une extension CDI peut implémenter la prise en charge de la portée conversation dans d'autres frameworks web.

Les portées requête et application sont aussi actifs :

- pendant l'appel de méthodes d'un EJB distant,
- pendant l'appel de méthodes asynchrones d'un EJB,
- pendant les timeouts EJB,
- pendant la distribution d'un message à un bean orienté message,
- pendant la distrubution d'un message à un `MessageListener`, et
- pendant les appels de service web.

Si une application essaie d'appeler un bean avec une portée qui n'a pas de contexte actif, une `ContextNotActiveException` est alors levée par le conteneur à l'exécution.

Managed beans with scope `@SessionScoped` or `@ConversationScoped` must be serializable, since the container passivates the HTTP session from time to time.

Trois des quatre portées intégrées devraient être extrêmement familières pour tous les développeurs Java EE, nous ne perdrons donc pas de temps ici pour en parler. Une des portées est, tout de même, nouvelle.

5.3. La portée conversation

La portée conversation ressemble un peu à la portée traditionnelle session dans le sens où elle porte l'état associé à un utilisateur du système, et s'étend sur plusieurs requêtes envoyées au serveur. Cependant, contrairement à la portée session, la portée conversation :

- est explicitement délimité par l'application, et
- porte l'état associé à un onglet particulier d'un navigateur web dans une application JSF (les navigateurs tendent à partager les cookies d'un domaine, et donc le cookie de session, entres les onglets, ce n'est donc pas le cas de la portée session).

Une conversation représente une tâche—une unité de travail du point de vue utilisateur. Le contexte de conversation porte l'état associé avec ce que l'utilisateur est actuellement en train de faire. Si un utilisateur fait plusieurs choses en même temps, il y a plusieurs conversations.

Le contexte de conversation est actif pendant n'importe quelle requête JSF. La plupart des conversations sont détruites à la fin de la requête. Si une conversation ne doit pas porter l'état à travers plusieurs requête, elle doit explicitement être promue à une *long-running conversation*.

5.3.1. Conversation de démarcation

CDI intègre un bean pour contrôler le cycle de vie des conversations d'une application JSF. Ce bean peut être obtenu pour injection :

```
@Inject Conversation conversation;
```

Pour promouvoir la conversation associée avec la requête courante en une conversation long-running, appelez la méthode `begin()` depuis le code de l'application. Pour planifier la destruction du contexte de conversation long-running courante à la fin de la requête courante, appelez `end()`.

Dans l'exemple suivant, un bean de portée conversation contrôle la conversation avec laquelle il est associé :

```
@ConversationScoped @Stateful
public class OrderBuilder {
   private Order order;
   private @Inject Conversation conversation;
   private @PersistenceContext(type = EXTENDED) EntityManager em;

   @Produces public Order getOrder() {
      return order;
   }

   public Order createOrder() {
      order = new Order();
      conversation.begin();
      return order;
   }

   public void addLineItem(Product product, int quantity) {
      order.add(new LineItem(product, quantity));
   }

   public void saveOrder(Order order) {
      em.persist(order);
      conversation.end();
   }

   @Remove
   public void destroy() {}
}
```

Ce bean est capable de contrôler son propre cycle de vie au travers de l'utilisation de l'API `Conversation`. Mais certains autres beans ont un cycle de vie qui dépend entièrement d'un autre objet.

5.3.2. Propagation de conversation

Le contexte de conversation se propage automatiquement à toutes requêtes de faces JSF (soumission de formulaire JSF) ou redirection. Il ne se propage pas automatiquement avec les requêtes non-faces, par exemple, la navigation via un lien.

We can force the conversation to propagate with a non-faces request by including the unique identifier of the conversation as a request parameter. The CDI specification reserves the request parameter named `cid` for this

use. The unique identifier of the conversation may be obtained from the `Conversation` object, which has the EL bean name `javax.enterprise.context.conversation`.

Par conséquent, le lien suivant propage la conversation :

```
<a href="/addProduct.jsp?cid=#{javax.enterprise.context.conversation.id}"
>Add Product</a
>
```

Il est probablement mieux d'utiliser l'un des composants link de JSF 2 :

```
<h:link outcome="/addProduct.xhtml" value="Add Product">
   <f:param name="cid" value="#{javax.enterprise.context.conversation.id}"/>
</h:link
>
```

Astuce

Le contexte de conversation est propagé au travers des redirections, rendant très simple l'implémentation du pattern courant POST-puis-redirect, sans recourir à de fragiles concepts comme un objet "flash". Le conteneur ajoute automatiquement l'id de conversation dans l'URL de redirection comme paramètre de requête.

5.3.3. Timeout de conversation

Le conteneur a la permission de détruire une conversation et tous les états portés dans son contexte à n'importe quel moment dans le but d'économiser des ressources. Un implémentation CDI fera normalement cela sur la base de certains types de timeout—comme cela n'est pas requis par la spécification. Le timeout est la période d'inactivité avant que la conversation soit détruite (opposé au temps où la conversation est active).

L'objet `Conversation` fournit une méthode pour configurer le timeout. Il s'agit d'indication au conteneur, qu'il est libre d'ignorer.

```
conversation.setTimeout(timeoutInMillis);
```

5.4. La pseudo-portée singleton

En plus des quatre portées intégrées, CDI supporte aussi deux *pseudo-portées*. La première est la *pseudo-portée singleton*, que vous pouvez spécifier en utilisant l'annotation `@Singleton`.

Note

Contrairement aux autres portées, qui appartiennent au package `javax.enterprise.context`, l'annotation `@Singleton` est définie dans le package `javax.inject`.

Vous pouvez deviner ce que "singleton" signifie ici. Il signifie qu'un bean n'est instancié qu'une fois. Malheureusement, il y a un petit problème avec cette pseudo-portée. Les beans avec la portée `@Singleton` n'ont pas d'objet proxy. Les clients tiennent une référence directe à l'instance singleton. Nous devons donc considérer le cas d'un client qui peut être sérialisé, par exemple, n'importe quel bean avec la portée `@SessionScoped` ou `@ConversationScoped`, n'importe quel objet dépendant d'un bean de portée `@SessionScoped` ou `@ConversationScoped`, ou n'importe quel bean de session sans état.

Maintenant, si l'instance du singleton est un objet simple, immutable et sérialisable comme un string, un nombre ou une date, nous n'avons probablement pas trop à nous préoccuper des duplications via sérialisation. Cependant, cela ne l'empêche pas d'être un vrai singleton, et l'on aurait tout aussi bien pu tout simplement le déclarer avec la portée par défaut.

Il y a plusieurs façon de s'assurer que le bean singleton reste un singleton quand son client se sérialise :

- avoir le bean singleton qui implémente `writeResolve()` et `readReplace()` (comme défini par la spécification Java serialization),
- être sur que le client ne conserve qu'une référence transient au bean singleton, ou
- donner au client une référence de type `Instance<X>` où `X` est le type du bean singleton.

Une quatrième, meilleure solution est d'utiliser à la place `@ApplicationScoped`, permettant au conteneur de proxifier le bean, et faire automatiquement attention aux problèmes de sérialisation.

5.5. La pseudo-portée dependent

Enfin, CDI founit la prétendu *pseudo-portée dependent*. Il s'agit de la portée par défaut pour un bean qui ne déclare pas explicitement un type de portée.

Par exemple, ce bean a le type de portée `@Dependent` :

```
public class Calculator { ... }
```

Une instance d'un bean dépendent n'est jamais partagée entre les différents client ou les différents points d'injection. Il est strictement un *objet dépendent* d'autres objets. Il est instancié quand l'objet dont il dépend est créé, et détruit quand l'objet dont il dépend est détruit.

Si une expression Unified EL référence un bean dépendant par un nom EL, une instance du bean est instancié chaque fois que l'expression est évaluée. L'instance n'est pas réutilisée l'évaluation de n'importe quelles autres expressions.

Note

Si vous avez besoin d'accéder directement à un bean par un nom EL dans une page JSF, vous aurez probablement besoin de lui donner une portée différente de `@Dependent`. Sinon, n'importe quelle valeur placée dans le bean par une entrée JSF sera immédiatement perdue. C'est pourquoi, CDI fournit le stéréotype `@Model` ; il vous permet de donner un nom à un bean, et positionne sa portée à `@RequestScoped` d'un seul coup. Si vous avez besoin d'accéder à un bean qui *a* vraiment besoin d'avoir la portée `@Dependent` depuis une page JSF, injectez le dans un bean différent, et exposez le à EL via une méthode getter.

Les beans de portée `@Dependent` n'ont pas besoin d'un objet proxy. Le client porte une référence direction à son instance.

CDI rend simple l'obtention d'une instance dépendante d'un bean, même si le bean est déjà déclaré dans un bean avec n'importe quel autre type de portée.

5.6. Le qualifiant `@New`

La qualifiant intégré `@New` nous permet d'obtenir un objet dépendant d'une classe spécifique.

```
@Inject @New Calculator calculator;
```

La classe doit être un bean managé ou bean session valide, mais n'a pas besoin d'être un bean activé.

Cela fonctionne même si `Calculator` est *déjà* déclaré avec un type de portée différente, par exemple :

```
@ConversationScoped
public class Calculator { ... }
```

Les attributs suivants alors injectés auront chacun une instance différent de `Calculator` :

```
public class PaymentCalc {
   @Inject Calculator calculator;
   @Inject @New Calculator newCalculator;
}
```

L'attribut `calculator` a une instance injectée de `Calculator` de portée conversation. L'attribut `newCalculator` a une nouvelle instance injectée de `Calculator`, avec un cycle de vie qui est lié au `PaymentCalc` possédé.

This feature is particularly useful with producer methods, as we'll see in *Chapitre 8, Méthodes de production*.

Partie II. Démarrer avec Weld, l'implémentation de référence de CDI

Weld, l'Implémentation de Référence (RI) de la JSR-299, est développé dans le cadre du *projet Seam* [http://seamframework.org/Weld]. Vous pouvez télécharger la dernière version communautaire de Weld sur la *page de téléchargement* [http://seamframework.org/Download]. Des informations sur le gestionnaire de sources de Weld et des instructions sur comment obtenir et compiler les sources peuvent être trouvées sur cette même page.

Weld fournis un SPI complet permettant aux conteneurs Java EE tel que JBoss AS et GlassFish d'utiliser Weld comme implémentation intégrée de CDI. Weld fonctionne aussi dans les moteurs de servlets comme Tomcat et Jetty, ou même dans l'environnement standard Java SE.

Weld est livré avec une vaste bibliothèque d'exemples, qui sont un bon point d'entré pour ceux qui veulent apprendre à utiliser CDI.

Démarrer avec Weld

Weld vient avec de nombreux exemples. Nous vous recommandons de démarrer avec `examples/jsf/numberguess` et `examples/jsf/translator`. Numberguess est un exemple web (war) contenant seulement des managed beans non transactionnels. Cet exemple peut être exécuté sur une large gamme de serveurs, incluant JBoss AS, GlassFish, Apache Tomcat, Jetty, Google App Engine, et n'importe quel conteneur compatible Java EE 6. Translator est un exemple entreprise (ear) qui contient des beans session. Cet exemple doit être exécuté sur JBoss AS 6.0, GlassFish 3.0 ou n'importe quel conteneur Java EE 6.

Les deux exemples utilisent JSF 2.0 comme framework web et, à ce titre, peut être trouvé dans le répertoire `examples/jsf` de la distribution Weld.

6.1. Pré-requis

Pour exécuter les exemples avec les scripts de build fournit, vous aurez besoin de :

- la dernière version de Weld, qui contient les exemples
- Ant 1.7.0, pour construire et déployer les exemples
- un environnement d'exécution supporté (versions minimums vues)
  - JBoss AS 6.0.0,
  - GlassFish 3.0,
  - Apache Tomcat 6.0.x (seulement pour les exemples war), ou
  - Jetty 6.1.x (seulement les exemples war)
- (facultatif) Maven 2.x, pour exécuter les exemples dans un conteneur de servlet intégré

Note

Vous aurez besoin d'une installation complète de Ant 1.7.0. Certaines distributions linux fournissent une installation partiel d'Ant qui provoque l'échec de la production. Si vous rencontrez des problèmes, vérifiez que ant-nodeps.jar est dans le classpath.

Dans les prochaines sections, vous aurez à utiliser la commande Ant (`ant`) pour invoquer le script de production Ant dans chaque exemple qui compile, assemble et déploie l'exemple sur JBoss AS et, pour les exemples war, Apache Tomcat. Vous pouvez aussi déployer l'artéfact généré (war ou ear) sur tout autre conteneur qui supporte Java EE 6, comme GlassFish 3.

Si vous avez Maven d'installé, vous pouvez utiliser la commande Maven (`mvn`) pour compiler et assembler l'artéfact autonome (war ou ear) et, pour les exemples war, les exécuter dans un conteneur intégré.

Les sections suivantes couvrent les étapes pour le déploiement avec Ant et Maven en détail. Démarrons avec JBoss AS.

6.2. Déploiement sur JBoss AS

Pour déployer les exemples sur JBoss AS, vous aurez besoin de *JBoss AS 6.0.0* [http://jboss.org/jbossas/] ou suivant. Si une version de la ligne JBoss AS 6.0 n'est pas disponible, vous pouvez télécharger une *version snapshot*

journalière [http://hudson.jboss.org/hudson/view/JBoss%20AS/job/JBoss-AS-6.0.x/]. La raison pour que JBoss AS 6.0.0 ou suivant soit requis est qu'il s'agit de la première version qui intègre le support de CDI et Bean Validation, en en faisant quelque chose de suffisamment proche de Java EE 6 pour exécuter les exemples. La bonne nouvelle est qu'il n'y aura pas de modifications supplémentaires que vous devrez faire sur le serveur. Il est prêt pour y aller !

Après que vous ayez téléchargé JBoss AS, extrayez le. (Nous recommandons de renommer le dossier en ajoutant `as` ; il sera alors clair qu'il s'agit d'un serveur d'application). Vous pouvez le dossier extrait n'importe où vous voulez. Par la suite, c'est ce que nous appellerons le répertoire d'installation JBoss AS, ou `JBOSS_HOME`.

```
$
> unzip jboss-6.0.*.zip
$
> mv jboss-6.0.*/ jboss-as-6.0
```

Pour permettre aux scripts de production de savoir où déployer l'exemple, vous avez à leurs dire où trouver l'installation JBoss AS (i.e., `JBOSS_HOME`). Créez un nouveau fichier nommé `local.build.properties` dans le répertoire des exemples de la distribution Weld et indiquez le path de votre installation JBoss AS dans la propriété `jboss.home`, comme suit :

```
jboss.home=/path/to/jboss-as-6.0
```

Vous êtes maintenant prêt à déployer votre premier exemple !

Placez vous dans le répertoire `examples/jsf/numberguess` et exécutez la cible Ant `deploy` :

```
$
> cd examples/jsf/numberguess
$
> ant deploy
```

Si vous ne l'avez pas encore fait, démarrez JBoss AS. Vous pouvez démarrer JBoss AS depuis soit un shell Linux :

```
$
> cd /path/to/jboss-as-6.0
$
> ./bin/run.sh
```

soit une invite de commande Windows :

```
$
> cd c:\path\to\jboss-as-6.0\bin
$
> run
```

ou vous pouvez démarrer le serveur en utilisant un IDE, comme Eclipse.

Note

Si vous utilisez Eclipse, vous devriez sérieusement envisager d'installer les add-ons *JBoss Tools* [http://www.jboss.org/tools], qui incluent une large variété d'outils pour la JSR-299 et le développement Java EE, ainsi qu'une vue Jboss AS server améliorée.

Attendez quelques secondes pour que l'application se déploie (ou que le serveur d'applications démarre) et voyez si vous pouvez déterminer l'approche la plus efficace pour trouver le nombre aléatoire à l'URL locale *http://localhost:8080/weld-numberguess*.

Note

Le script de production Ant inclut des cibles additionnelles pour JBoss AS pour déployer et supprimer l'archive sous forme éclatée ou packagée et pour ranger les choses.

- `ant restart` - déploie l'exemple sous forme éclatée dans JBoss AS
- `ant explode` - met à jour un exemple éclaté, sans redémarrer le déploiement
- `ant deploy` - déploie l'exemple sous forme compressée dans un jar dans JBoss AS
- `ant undeploy` - supprime l'exemple de JBoss AS
- `ant clean` - nettoie l'exemple

Le second exemple de démarrage, `weld-translator`, traduira votre texte en Latin. (Bon, pas vraiment, mais le stub est là pour que vous au moins l'implémenter. Bonne chance !) Pour l'essayer, placez vous dans le répertoire de l'exemple de traduction et exécutez la cible de déploiement :

```
$
> cd examples/jsf/translator
$
> ant deploy
```

Note

Le traducteur utilise des beans session, qui sont packagés dans un module EJB à l'intérieur d'un ear. Java EE 6 permettra aux beans session d'être déployés dans des modules war, mais c'est un sujet pour un chapitre ultérieur.

Encore, attendez quelques secondes que l'application se déploie (si vous vous ennuyez vraiment, lisez les messages de log), et visitez *http://localhost:8080/weld-translator* pour commencer la pseudo-traduction.

6.3. Déploiement sur GlassFish

Déployer sur GlassFish devrait être facile et familier, non ? Après tout, c'est l'implémentation de référence de Java EE 6 et Weld est l'implémentation de référence de la JSR-299, signifiant que Weld est packagé dans GlassFish. Alors oui, c'est assez facile et familier.

Pour déployer les exemples dans GlassFish, vous aurez besoin de la dernière version de *GlassFish V3* [https://glassfish.dev.java.net/downloads/v3-final.html]. Sélectionnez la version qui termine avec soit `-unix.sh` soit `-windows.exe` suivant votre plateforme. Après que le téléchargement soit complet, exécutez l'installeur. Sous Linux/Unix, vous devrez d'abord rendre le script exécutable.

```
$
> chmod 755 glassfish-v3-unix.sh
$
> ./glassfish-v3-unix.sh
```

Sous Windows, vous avez juste à cliquer sur l'exécutable. Suivez les instructions de l'installeur. Cela créera un domaine unique nommé `domain1`. Vous utiliserez ce domaine pour déployer l'exemple. Nous recommandons de choisir `7070` comme port HTTP principal afin d'éviter les conflits avec une instance de JBoss AS (ou Apache Tomcat).

Si vous avez déployez les exemples de démarrage, `weld-numberguess` ou `weld-translator`, dans JBoss AS, alors vous avez déjà l'artéfact déployable nécessaire. Si non, naviguez dans l'un des deux répertoires et produisez le.

```
$
> cd examples/jsf/numberguess (or examples/jsf/translator)
$
> ant package
```

L'archive déployable pour `weld-numberguess`, nommé `weld-numberguess.war`, se trouve dans le répertoire `target` de l'exemple. L'archive pour l'exemple `weld-translator`, nommé `weld-translator.ear`, se trouve dans le répertoire `ear/target` de l'exemple. Tout ce que vous avez à faire maintenant est de les déployer dans GlassFish.

Un moyen de déployer des applications dans GlassFish est en utilisant la *GlassFish Admin Console* [http://localhost:4848]. Pour exécuter les Console d'Administration, vous avez besoin un domaine de GlassFish, dans votre cas `domain1`. Naviguer dans le dossier `bin` du répertoire où vous avez installé GlassFish et exécuter la commande suivante :

```
$
> asadmin start-domain domain1
```

Après quelques secondes, vous pouvez visiter la Console d'Administration dans le navigateur à l'URL *http://localhost:4848*. Dans l'arborescence à gauche de la page, cliquez sur "Applications", ensuite cliquez sur le bouton "Deploy..." sous le titre "Applications" et sélectionnez l'artéfact déployable pour l'un des deux exemples. Le déployeur devrait reconnaître que vous avez sélectionné un artéfact Java EE et vous permet de le démarrer. Vous pouvez voir les exemples tourner sur *http://localhost:7070/weld-numberguess* ou *http://localhost:7070/weld-translator*, suivant l'exemple que vous avez déployé.

Sinon, vous pouvez déployer l'application dans GlassFish en utilisant la commande asadmin :

```
$
> asadmin deploy target/weld-numberguess.war
```

La raison pour laquelle un même artéfact peut être déployé à la fois sur JBoss AS et GlassFish, sans aucunes modifications, est que toutes les fonctionnalités utilisées font parties de la plateforme standard. Quelle plateforme compétente c'est devenu !

6.4. Déploiement sur Apache Tomcat

Les conteneurs de servlets ne sont pas nécessaires pour prendre en charge les services Java EE comme CDI. Cependant, vous pouvez utiliser CDI dans un conteneur de servlets comme Tomcat en embarquant une implémentation CDI autonome comme Weld.

Weld vient avec un écouteur de servlet qui démarre l'environnement CDI, enregistre le `BeanManager` dans JNDI et fournit l'injection dans les servlets. Au fond, il simule une partie du travail faut par le conteneur Java EE. (Mais vous n'aurez pas les fonctionnalités d'entreprise comme les beans session et les transactions managées par le conteneur.)

Let's give the Weld servlet extension a spin on Apache Tomcat. First, you'll need to download Tomcat 6.0.18 or later from *tomcat.apache.org* [http://tomcat.apache.org/download-60.cgi] and extract it.

```
$
> unzip apache-tomcat-6.0.18.zip
```

Vous avez deux possibilités pour la façon dont vous pouvez déployer l'application dans Tomcat. Vous pouvez la déployer en poussant l'artéfact dans le répertoire de déploiement à chaud en utilisant Ant ou vous pouvez déployer sur le serveur au travers d'HTTP en utilisant un plugin Maven. L'approche Ant ne requiert pas que vous ayez Maven d'installé, nous commencerons donc comme ça. Si vous voulez utiliser Maven, vous pouvez juste sauter plus loin.

6.4.1. Déployer avec Ant

Pour qu'Ant puisse publier l'artéfact dans le répertoire de déploiement à chaud de Tomcat, il a besoin de savoir où est localisé le répertoire d'installation de Tomcat. Encore une fois, nous devons positionner une propriété dans le fichier `local.build.properties` du répertoire des exemples de Weld. Si vous n'avez pas encore créé ce fichier, faite le maintenant. Ensuite, assignez le path de votre installation Tomcat à la propriété `tomcat.home`.

```
tomcat.home=/path/to/apache-tomcat-6
```

Vous êtes maintenant prêts pour déployer l'exemple numberguess sur Tomcat !

Changez encore une fois le répertoire `examples/jsf/numberguess` et exécutez la target Ant `deploy` pour Tomcat :

```
$
> cd examples/jsf/numberguess
$
> ant tomcat.deploy
```

Note

Le script de production Ant inclut des cibles additionnelles pour Tomcat afin de déployer et supprimer l'archive dans un format à plat ou packagé. Les ont le même nom que ceux utilisés pour JBoss AS, mais préfixés avec "tomcat.".

- `ant tomcat.restart` - déploie l'exemple dans un format à plat sur Tomcat
- `ant tomcat.explode` - met à jour un exemple à plat, sans redémarrer le déploiement
- `ant tomcat.deploy` - déploie l'exemple dans un format compressé jar sur Tomcat
- `ant tomcat.undeploy` - supprime l'exemple de Tomcat

Si vous ne l'avez pas encore fait, démarrez Tomcat. Vous pouvez démarrer Tomcat soit depuis un shell Linux :

```
$
> cd /path/to/apache-tomcat-6
$
> ./bin/start.sh
```

soit une invite de commande Windows :

```
$
> cd c:\path\to\apache-tomcat-6\bin
$
> start
```

ou vous pouvez démarrer le serveur en utilisant un IDE, comme Eclipse.

Attendez quelques secondes pour que l'application se déploie (ou que le serveur d'application démarre) et voyez si vous pouvez trouver la meilleure stratégie pour trouver un nombre aléatoire à l'URL locale *http://localhost:8080/weld-numberguess* !

6.4.2. Déployer avec Maven

Vous pouvez aussi déployer l'application sur Tomcat en utilisant Maven. Cette section est un peu plus avancée, alors sautez la si vous n'avez pas envie d'utiliser Maven. Bien sur, vous aurez d'abord à vous assurer que vous avez Maven d'installé dans votre path, de la même façon que l'installation de Ant.

Le plugin Maven communique avec Tomcat via HTTP, il ne s'inquiète pas de savoir où vous avez installé Tomcat. Cependant, la configuration du plugin suppose que vous avez exécuté Tomcat dans sa configuration par défaut, avec pour hostname localhost et pour port `8080`. Le fichier `readme.txt` dans le répertoire d'exemple contient des informations sur la façon de modifier les paramètres de Maven pour s'adapter à une installation différente.

Pour permettre à Maven de communiquer avec Tomcat via HTTP, éditez le fichier `conf/tomcat-users.xml` situé dans votre installation Tomcat et ajoutez la ligne suivante :

```
<user username="admin" password="" roles="manager"/>
```

Redémarrez Tomcat. Vous pouvez maintenant déployer l'application sur Tomcat avec Maven en utilisant la commande :

```
$
> mvn compile war:exploded tomcat:exploded -Ptomcat
```

Une fois que l'application est déployée, vous pouvez la redéployer en utilisant cette commande :

```
$
> mvn tomcat:redeploy -Ptomcat
```

L'argument `-Ptomcat` active la profile `tomcat` définit dans le POM Maven (`pom.xml`). Entre autres choses, le profile active le plugin Tomcat.

Plutôt qu'envoyer le conteneur dans une installation autonome de Tomcat, vous pouvez aussi exécuter l'application dans un conteneur embarqué Tomcat 6.

```
$
> mvn war:inplace tomcat:run -Ptomcat
```

L'avantage d'utiliser un serveur embarqué est que tous changements dans `src/main/webapp` prendront effet immédiatement. Si un changement au fichier de configuration webapp est fait, l'application peut être automatiquement déployée (suivant la configuration du plugin). Si vous effectuez un changement à une ressource du classpath, vous devrez exécuter une production :

```
$
> mvn compile war:inplace -Ptomcat
```

Il y a plusieurs autres goals Maven que vous pouvez utiliser si vous jouez avec l'exemple, qui sont documentés dans le fichier `readme.txt` de l'exemple.

6.5. Déployer sur Jetty

La prise en charge des exemples par Jetty est un ajout récent. Puisque Jetty utilise traditionnellement Maven, il n'y a pas de cible Ant. Vous devez juste lancer une production Maven pour déployer les exemples sur Jetty. De plus, seul l'exemple `weld-numberguess` est configuré pour prendre en charge Jetty au moment où nous écrivons ces lignes.

Si vous avez lu en entier toute la section Tomcat, alors vous êtes prêt pour la suite. La production Maven parallélise le déploiement sur Tomcat embarqué. Si non, pas d'inquiétude. Nous allons encore voir ce que vous avez besoin de savoir dans cette section.

Le POM Maven (`pom.xml`) inclut un profile nommé `jetty` qui active le plugin Jetty pour Maven, que vous pouvez utiliser pour démarrer Jetty dans un mode embarqué et y déployer l'application. Vous n'avez besoin de rien installer de plus autre qu'avoir la commande Maven (`mvn`) dans votre path. Le reste sera téléchargé depuis internet quand la production sera exécutée.

Pour exécuter l'exemple `weld-numberguess` sur Jetty, naviguez dans le répertoire de l'exemple et exécutez le goal `inplace` du plugin war pour Maven suivi du goal `run` du plugin Jetty pour Maven avec le profile `jetty` activé, comme suivant :

```
$
> cd examples/jsf/numberguess
$
> mvn war:inplace jetty:run -Pjetty
```

Le journal de sortie de Jetty sera visible dans la console. Une fois que Jetty indique que l'application est déployée, vous pouvez y accéder à l'URL locale suivante : *http://localhost:9090/weld-numberguess*. Le port est défini dans la configuration du plugin Jetty pour Maven situé dans le profile `jetty`.

N'importe quels changements dans `src/main/webapp` prendra effet immédiatement. Si un changement dans le fichier de configuration webapp est fait, l'application peut être automatiquement redéployée. Le comportement de redéploiement peut être peaufiné dans la configuration du plugin. Si vous effectuez un changement à une ressource du classpath, vous devez exécuter une production et le goal `inplace` du plugin war pour Maven, une fois encore avec le profile `jetty` activé.

```
$
> mvn compile war:inplace -Pjetty
```

Le goal `war:inplace` copie les classes compilées et les jars à l'intérieur de `src/main/webapp`, vers `WEB-INF/classes` et `WEB-INF/lib`, respectivement, mélangeant fichiers sources et compilés. Cependant, la production travaille avec ces fichiers temporaires en les excluant du packaging war et en les nettoyant pendant la phase de clean de Maven.

Vous avez deux options si vous voulez exécuter l'exemple dans Jetty depuis l'IDE. Vous pouvez installer le plugin m2eclipse[link] et exécuter les goals comme décrit précédemment. Votre autre option est de démarrer les conteneur Jetty depuis une application Java.

D'abord, initialisez le projet Eclipse :

```
$
> mvn clean eclipse:clean eclipse:eclipse -Pjetty-ide
```

Ensuite, rassemblez toutes les ressources nécessaires sous `src/main/webapp` :

```
$
> mvn war:inplace -Pjetty-ide
```

Now, you are ready to run the server in Eclipse. Import the project into your Eclipse workspace using "Import Existing Project into Workspace. Then, find the start class in `src/jetty/java` and run its main method as a Java Application. Jetty will launch. You can view the application at the following local URL: *http://localhost:8080*. Pay particular attention to the port in the URL and the lack of a trailing context path.

Maintenant, vous avez les applications de démarrage déployées dans le serveur de votre choix ; vous voulez probablement connaitre un peu plus à propos de la façon dont elles fonctionnent.

Immersion dans les exemples Weld

It's time to pull the covers back and dive into the internals of Weld example applications. Let's start with the simpler of the two examples, `weld-numberguess`.

7.1. L'exemple numberguess (devine le nombre) en profondeur

Dans l'application numberguess vous avez 10 tentatives pour trouver un nombre entre 1 et 100. Après chaque tentative, on vous dit si votre proposition était trop haute ou trop basse.

L'exemple numberguess comprend un ensemble de beans, fichiers de configuration et vues Facelets (JSF), le tout assemblé comme un module war. Commençons en examinant les fichiers de configuration.

All the configuration files for this example are located in `WEB-INF/`, which can be found in the `src/main/webapp` directory of the example. First, we have the JSF 2.0 version of `faces-config.xml`. A standardized version of Facelets is the default view handler in JSF 2.0, so there's really nothing that we have to configure. Thus, the configuration consists of only the root element.

```
<faces-config version="2.0"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
</faces-config
>
```

Il y a également un fichier `beans.xml` vide, qui indique au conteneur d'inspecter les beans de cette application et d'activer les services CDI.

Enfin, il y a le familier `web.xml` :

```
<web-app version="2.5"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

   <display-name
>weld-jsf-numberguess-war</display-name>
   <description                                                    1
>Weld JSF numberguess example (war)</description>

   <servlet>
      <servlet-name
>Faces Servlet</servlet-name>
      <servlet-class                                              2
>javax.faces.webapp.FacesServlet</servlet-class>
```

```
      <load-on-startup
>1</load-on-startup>
   </servlet>

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

   <context-param>
      <param-name
>javax.faces.DEFAULT_SUFFIX</param-name>
      <param-value
>.xhtml</param-value>
   </context-param>

   <session-config>
      <session-timeout
>10</session-timeout>
   </session-config>

</web-app
>
```

1. Enable and initialize the JSF servlet
2. Configure requests for URLs ending in `.jsf` to be handled by JSF
3. Tell JSF that we will be giving our JSF views (Facelets templates) an extension of `.xhtml`
4. Configure a session timeout of 10 minutes

Note

This demo uses JSF 2 as the view framework, but you can use Weld with any servlet-based web framework, such as JSF 1.2 or Wicket.

Let's take a look at the main JSF view, `src/main/webapp/home.xhtml`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core">

   <ui:composition template="/template.xhtml">
      <ui:define name="content">
         <h1
>Guess a number...</h1>
         <h:form id="numberGuess">
            <div style="color: red">
```

```
            <h:messages id="messages" globalOnly="false"/>
            <h:outputText id="Higher" value="Higher!"
               rendered="#{game.number gt game.guess and game.guess ne 0}"/>
            <h:outputText id="Lower" value="Lower!"
               rendered="#{game.number lt game.guess and game.guess ne 0}"/>
         </div>

         <div>
            I'm thinking of a number between #{game.smallest} and #{game.biggest}.
            You have #{game.remainingGuesses} guesses remaining.
         </div>

         <div>
            Your guess:
            <h:inputText id="inputGuess" value="#{game.guess}"
               size="3" required="true" disabled="#{game.number eq game.guess}"
               validator="#{game.validateNumberRange}"/>
            <h:commandButton id="guessButton" value="Guess"
               action="#{game.check}" disabled="#{game.number eq game.guess}"/>
         </div>
         <div>

  <h:commandButton id="restartButton" value="Reset" action="#{game.reset}" immediate="true"/>
         </div>
      </h:form>
   </ui:define>
</ui:composition>
</html
>
```

1. Facelets is the built-in templating language for JSF. Here we are wrapping our page in a template which defines the layout.
2. There are a number of messages which can be sent to the user, "Higher!", "Lower!" and "Correct!"
3. As the user guesses, the range of numbers they can guess gets smaller - this sentence changes to make sure they know the number range of a valid guess.
4. This input field is bound to a bean property using a value expression.
5. A validator binding is used to make sure the user doesn't accidentally input a number outside of the range in which they can guess - if the validator wasn't here, the user might use up a guess on an out of bounds number.
6. And, of course, there must be a way for the user to send their guess to the server. Here we bind to an action method on the bean.

Cet exemple se fait en 4 classes, les deux premières étant des qualifiants. D'abord, il y a le qualifiant `@Random`, utilisé pour l'injection d'un nombre aléatoire :

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

Il y a aussi le qualifiant `@MaxNumber`, utilisé pour l'injection du nombre maximum qui peut être injecté :

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

La classe de portée application `Generator` est responsable de créer le nombre aléatoire, via une méthode de production. Il expose aussi le nombre maximum possible via une méthode de production :

```
@ApplicationScoped
public class Generator implements Serializable {

   private java.util.Random random = new java.util.Random(System.currentTimeMillis());

   private int maxNumber = 100;

   java.util.Random getRandom() {
      return random;
   }

   @Produces @Random int next() {
      return getRandom().nextInt(maxNumber);
   }

   @Produces @MaxNumber int getMaxNumber() {
      return maxNumber;
   }

}
```

Le `Generator` est de portée application, vous n'aurez donc pas un nombre aléatoire différence à chaque fois.

Note

La déclaration de package ainsi que les imports ont été exclu dans ce code. Le code complet est disponible dans le source code des exemples.

Le dernier bean de l'application est la classe `Game` de portée session. C'est le point d'entré principal de l'application. Il est responsable pour configurer ou réinitialiser le jeu, capturer et valider les suppositions de l'utilisateur et fournir du feedback à l'utilisateur avec un `FacesMessage`. Nous avons utilisé la méthode du cycle de vie e jeu en recherchant un nombre aléatoire depuis le bean `@Random Instance<Integer>`.

Vous noterez que nous avons déjà ajouté l'annotation `@Named` à cette classe. Cette annotation est uniquement requise quand vous voulez rendre un bean accessible depuis un vue JSF via EL (i.e., #{game}).

```
@Named
@SessionScoped
public class Game implements Serializable {

   private int number;
   private int guess;
```

```
   private int smallest;
   private int biggest;
   private int remainingGuesses;

   @Inject @MaxNumber private int maxNumber;
   @Inject @Random Instance<Integer
> randomNumber;

   public Game() {}

   public void check() {
      if (guess
> number) {
         biggest = guess - 1;
      }
      else if (guess < number) {
         smallest = guess + 1;
      }
      else if (guess == number) {
         FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Correct!"));
      }
      remainingGuesses--;
   }

   @PostConstruct
   public void reset() {
      this.smallest = 0;
      this.guess = 0;
      this.remainingGuesses = 10;
      this.biggest = maxNumber;
      this.number = randomNumber.get();
   }

  public void validateNumberRange(FacesContext context,  UIComponent toValidate, Object value) {
      if (remainingGuesses <= 0) {
         FacesMessage message = new FacesMessage("No guesses left!");
         context.addMessage(toValidate.getClientId(context), message);
         ((UIInput) toValidate).setValid(false);
         return;
      }
      int input = (Integer) value;

      if (input < smallest || input
> biggest) {
         ((UIInput) toValidate).setValid(false);

         FacesMessage message = new FacesMessage("Invalid guess");
         context.addMessage(toValidate.getClientId(context), message);
      }
   }

   public int getNumber() {
      return number;
   }

   public int getGuess() {
      return guess;
   }
```

```
   public void setGuess(int guess) {
      this.guess = guess;
   }

   public int getSmallest() {
      return smallest;
   }

   public int getBiggest() {
      return biggest;
   }

   public int getRemainingGuesses() {
      return remainingGuesses;
   }

}
```

7.1.1. L'exemple numberguess dans Apache Tomcat ou Jetty

Une paire de modifications doivent être faite à l'artéfact numberguess dans le but de le déployer dans Tomcat ou Jetty. D'abord, Weld doit être déployé comme une librairie d'Application Web dans `WEB-INF/lib` puisque le conteneur de servlets nous ne fournit pas de services CDI. Pour votre confort, nous fournissons un jar unique approprié pour exécuter Weld dans n'importe quel conteneur de servlet (Jetty inclus), `weld-servlet.jar`.

Note

Nous devons aussi inclure les jars de JSF, EL, et les annotations communes (`jsr250-api.jar`), toutes celles qui sont fournies par la plateforme Java EE (un serveur d'applications Java EE). Est-ce que vous commencer à comprendre pourquoi il est important d'utiliser une plateforme Java EE ?

Second, we need to explicitly specify the servlet listener in `web.xml`, again because the container isn't doing this stuff for you. The servlet listener boots Weld and controls it's interaction with requests.

```
<listener>
   <listener-class
>org.jboss.weld.environment.servlet.Listener</listener-class>
</listener
>
```

When Weld boots, it places the `javax.enterprise.inject.spi.BeanManager`, the portable SPI for obtaining bean instances, in the ServletContext under a variable name equal to the fully-qualified interface name. You generally don't need to access this interface, but Weld makes use of it.

7.2. L'exemple numberguess pour Java SE avec Swing

Cet exemple montre comment utiliser l'extension Weld SE dans une application Java SE basée sur Swing sans dépendances EJB ou servlet. Cet exemple peut être trouvé dans le dossier `examples/se/numberguess` de la distribution Weld.

7.2.1. Création du projet Eclipse

Pour utiliser l'exemple Weld SE numberguess dans Eclipse, vous pouvez ouvrir nativement l'exemple en utilisant le *plugin m2eclipse* [http://m2eclipse.sonatype.org/].

If you have m2eclipse installed, you can open any Maven project directly. From within Eclipse, select *File -> Import... -> Maven Projects*. Then, browse to the location of the Weld SE numberguess example. You should see that Eclipse recognizes the existence of a Maven project.

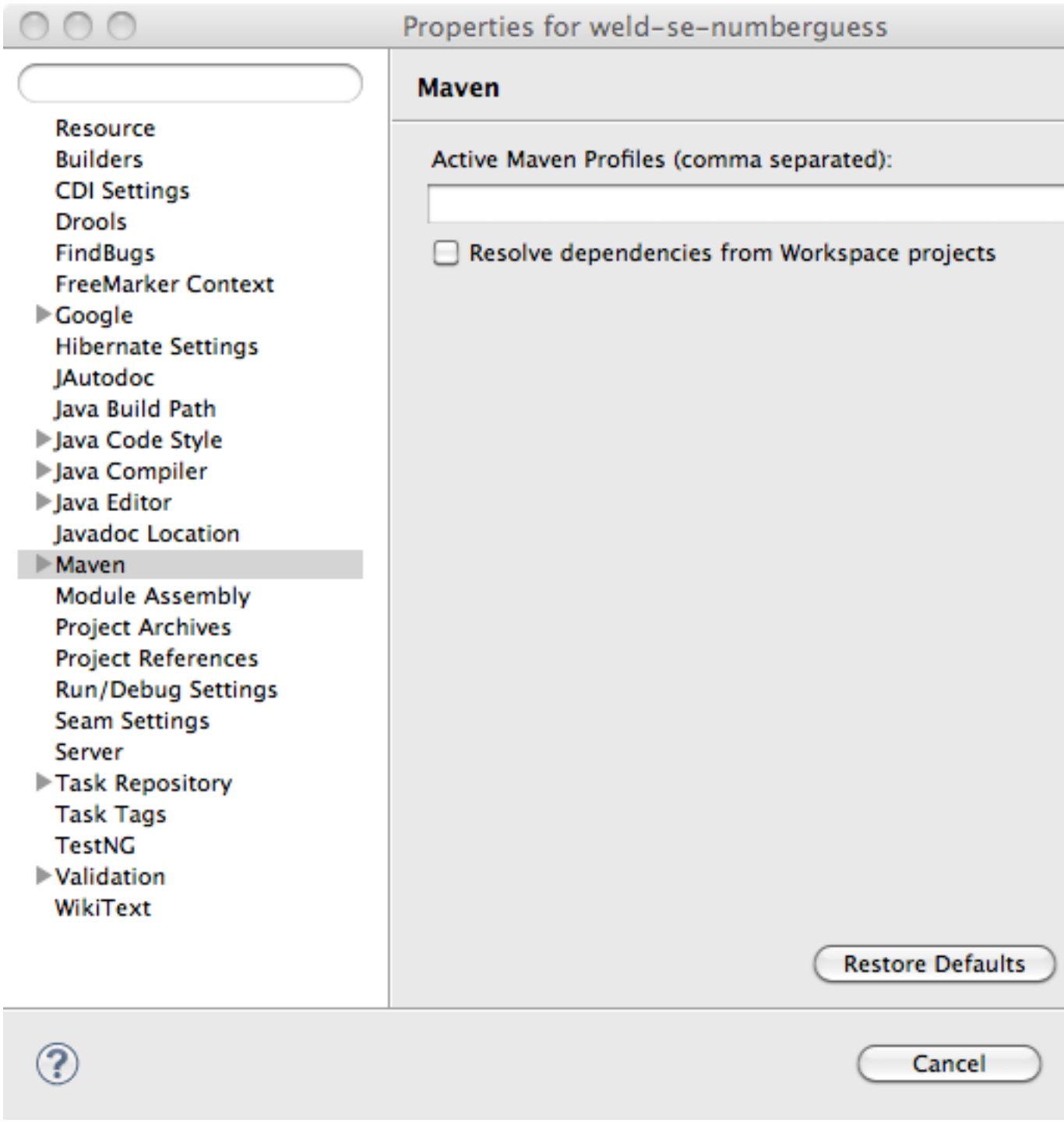
Ceci créera un projet dans votre workspace nommé `weld-se-numberguess`.

Si vous n'utilisez pas le plugin m2eclipse, vous avez à suivre différentes étapes pour importer le projet. D'abord, passez à l'exemple Weld SE numberguess, ensuite exécutez le plugin Maven pour Eclipse en activant le profile jetty, comme ceci :

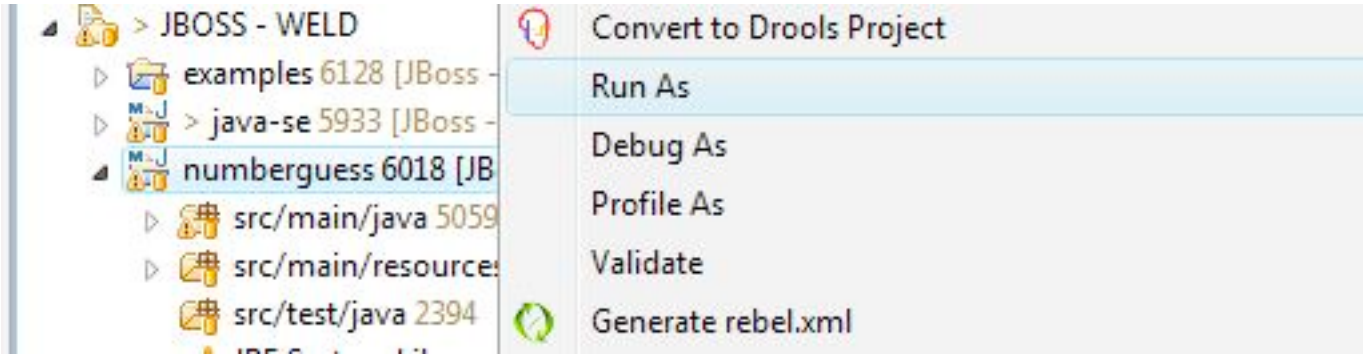
Il est temps de voir l'exemple s'exécuter !

7.2.2. Exécuter l'exemple depuis Eclipse

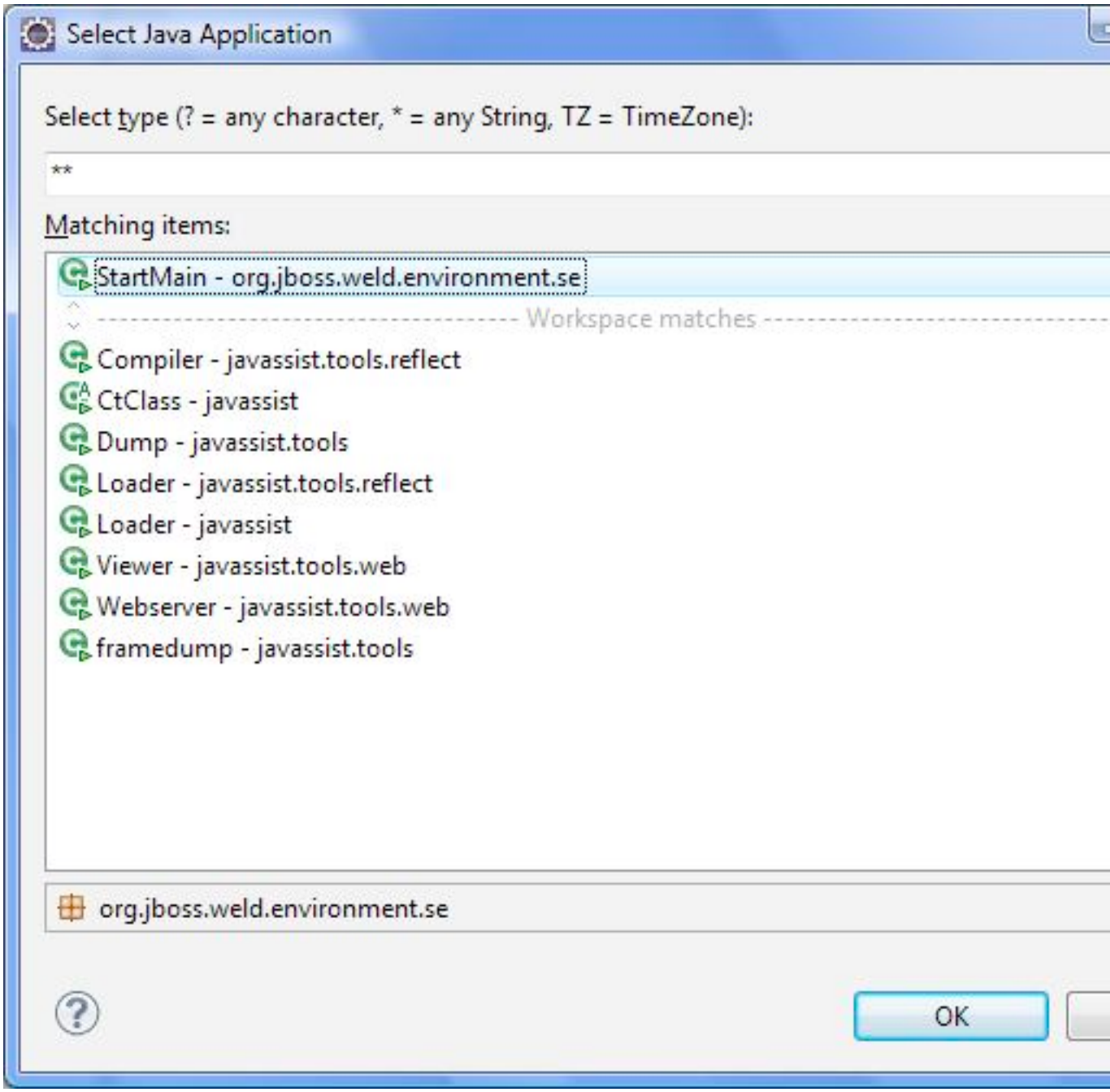
Disable m2eclipse's *Workspace Resolution*, to make sure that Eclipse can find `StartMain`. Right click on the project, and choose *Properties -> Maven*, and uncheck *Resolve dependencies from Workspace projects*:



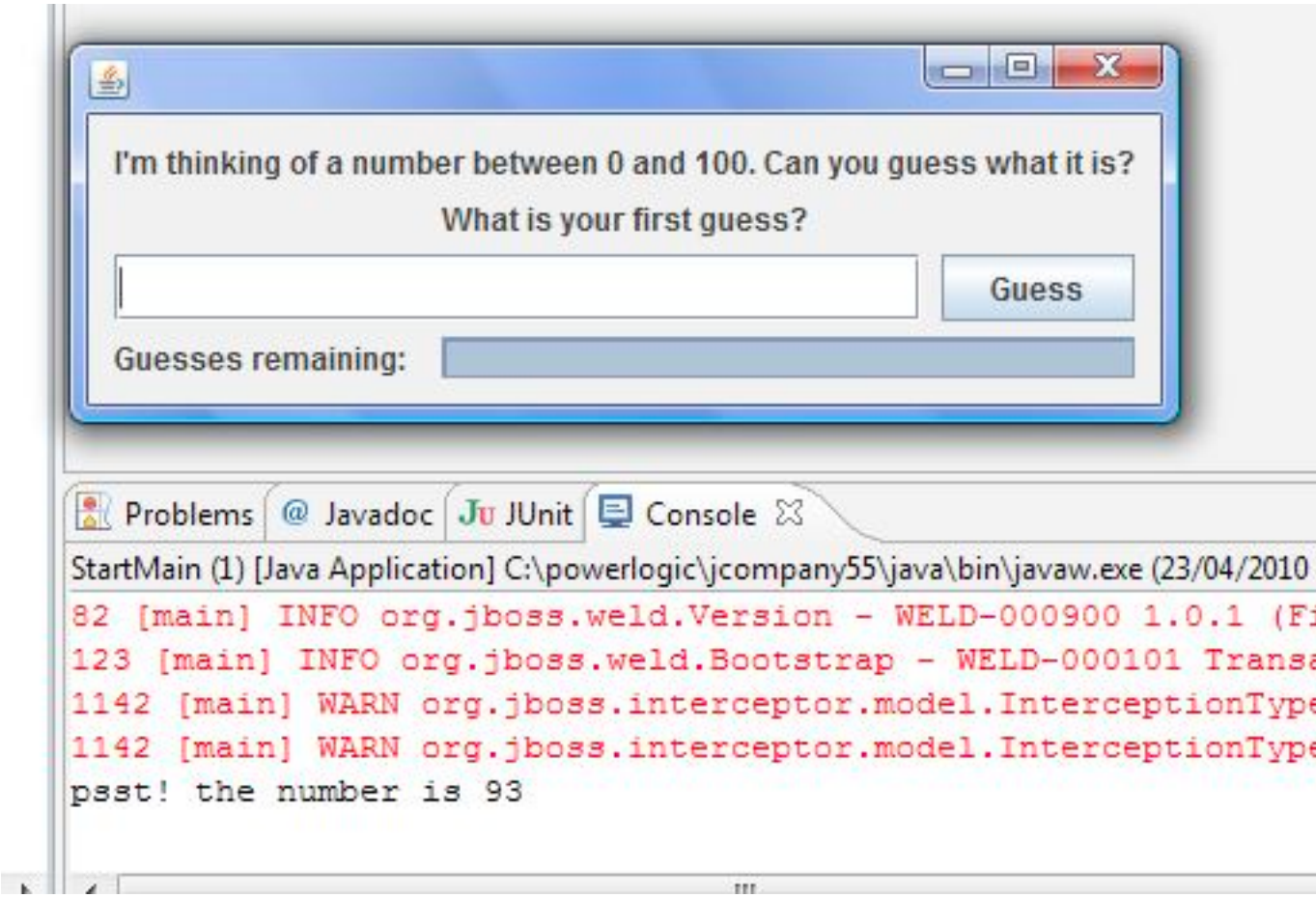
Right click on the project, and choose *Run As -> Run As Application*:



Repérez la classe `StartMain` :



L'application devrait maintenant être exécutée !



7.2.3. Exécuter l'exemple depuis la ligne de commande

- Assurez vous que Maven 3 est installé et dans votre PATH
- Assurez vous que la variable d'environnement JAVA_HOME pointe vers votre installation du JDK
- Ouvrez une ligne de commande ou un terminal dans le répertoire `examples/se/numberguess`
- Exécutez la commande suivante

```
mvn -Drun
```

7.2.4. Comprendre le code

Jetons un oeil au code significatif et aux fichiers de configuration qui font cet exemple.

Comme d'habitude, il y a un fichier vide `beans.xml` dans le package racine (`src/main/resources/beans.xml`), qui indique que cette application est une application CDI.

La logique principale du jeu est localisée dans `Game.java`. Voici le code de cette classe, mettant en valeur les choses qui diffère de la version web de l'application :

```
@ApplicationScoped                                                    (1)
public class Game                                                     (2)
{

   public static final int MAX_NUM_GUESSES = 10;

   private Integer number;
   private int guess = 0;
   private int smallest = 0;

   @Inject
   @MaxNumber
   private int maxNumber;

   private int biggest;
   private int remainingGuesses = MAX_NUM_GUESSES;
   private boolean validNumberRange = true;

   @Inject
   Generator rndGenerator;

   public Game()
   {
   }

   ...                                                                (3)

   public boolean isValidNumberRange()
   {
      return validNumberRange;
   }

   public boolean isGameWon()
   {
      return guess == number;
   }

   public boolean isGameLost()
   {
      return guess != number && remainingGuesses <= 0;
   }                                                                  (4)

   public boolean check()
   {
      boolean result = false;

      if (checkNewNumberRangeIsValid())
      {
         if (guess
> number)
         {
            biggest = guess - 1;
         }

         if (guess < number)
         {
```

```
            smallest = guess + 1;
         }

         if (guess == number)
         {
            result = true;
         }

         remainingGuesses--;
      }

      return result;
   }

   private boolean checkNewNumberRangeIsValid()
   {
      return validNumberRange = ((guess
>= smallest) && (guess <= biggest));
   }

   @PostConstruct
   public void reset()
   {
      this.smallest = 0;
      this.guess = 0;
      this.remainingGuesses = 10;
      this.biggest = maxNumber;
      this.number = rndGenerator.next();
   }
}
```

(5)

1. The bean is application scoped rather than session scoped, since an instance of a Swing application typically represents a single 'session'.
2. Notice that the bean is not named, since it doesn't need to be accessed via EL.
3. In Java SE there is no JSF `FacesContext` to which messages can be added. Instead the `Game` class provides additional information about the state of the current game including:

   - If the game has been won or lost
   - If the most recent guess was invalid

   This allows the Swing UI to query the state of the game, which it does indirectly via a class called `MessageGenerator`, in order to determine the appropriate messages to display to the user during the game.
4. Since there is no dedicated validation phase, validation of user input is performed during the `check()` method.
5. The `reset()` method makes a call to the injected `rndGenerator` in order to get the random number at the start of each game. Note that it can't use `Instance.get()` like the JSF example does because there will not be any active contexts like there are during a JSF request.

The `MessageGenerator` class depends on the current instance of `Game` and queries its state in order to determine the appropriate messages to provide as the prompt for the user's next guess and the response to the previous guess. The code for `MessageGenerator` is as follows:

```
public class MessageGenerator
{
```

```
   @Inject                                                                    (1)
   private Game game;

   public String getChallengeMessage()                                        (2)
   {
      StringBuilder challengeMsg = new StringBuilder("I'm thinking of a number between ");
      challengeMsg.append(game.getSmallest());
      challengeMsg.append(" and ");
      challengeMsg.append(game.getBiggest());
      challengeMsg.append(". Can you guess what it is?");

      return challengeMsg.toString();
   }

   public String getResultMessage()                                           (3)
   {
      if (game.isGameWon())
      {
         return "You guessed it! The number was " + game.getNumber();
      }
      else if (game.isGameLost())
      {
         return "You are fail! The number was " + game.getNumber();
      }
      else if (!game.isValidNumberRange())
      {
         return "Invalid number range!";
      }
      else if (game.getRemainingGuesses() == Game.MAX_NUM_GUESSES)
      {
         return "What is your first guess?";
      }
      else
      {
         String direction = null;

         if (game.getGuess() < game.getNumber())
         {
            direction = "Higher";
         }
         else
         {
            direction = "Lower";
         }

         return direction + "! You have " + game.getRemainingGuesses() + " guesses left.";
      }
   }
}
```

(1) The instance of `Game` for the application is injected here.

(2) The `Game`'s state is interrogated to determine the appropriate challenge message ...

(3) ... and again to determine whether to congratulate, console or encourage the user to continue.

Finally we come to the `NumberGuessFrame` class which provides the Swing front end to our guessing game.

```
public class NumberGuessFrame extends javax.swing.JFrame
{
   @Inject                                                          (1)
   private Game game;

   @Inject                                                          (2)
   private MessageGenerator msgGenerator;

   public void start(@Observes ContainerInitialized event)          (3)
   {
      java.awt.EventQueue.invokeLater(new Runnable()
      {
         public void run()
         {
            initComponents();
            setVisible(true);
         }
      });
   }

   private void initComponents()                                    (4)
   {

      buttonPanel = new javax.swing.JPanel();
      mainMsgPanel = new javax.swing.JPanel();
      mainLabel = new javax.swing.JLabel();
      messageLabel = new javax.swing.JLabel();
      guessText = new javax.swing.JTextField();
      ...
      mainLabel.setText(msgGenerator.getChallengeMessage());
      mainMsgPanel.add(mainLabel);

      messageLabel.setText(msgGenerator.getResultMessage());
      mainMsgPanel.add(messageLabel);
      ...
   }

   private void guessButtonActionPerformed( java.awt.event.ActionEvent evt )  (5)
   {
      int guess =  Integer.parseInt(guessText.getText());
      game.setGuess( guess );
      game.check();
      refreshUI();
   }

   private void replayBtnActionPerformed(java.awt.event.ActionEvent evt)
   {
      game.reset();                                                 (6)
      refreshUI();
   }

   private void refreshUI() {
      mainLabel.setText( msgGenerator.getChallengeMessage() );
      messageLabel.setText( msgGenerator.getResultMessage() );
      guessText.setText( "" );
```

```
      guessesLeftBar.setValue( game.getRemainingGuesses() );
      guessText.requestFocus();
   }

   // swing components
   private javax.swing.JPanel borderPanel;
   ...
   private javax.swing.JButton replayBtn;

}
```

1. The injected instance of the game (logic and state).
2. The injected message generator for UI messages.
3. This application is started in the prescribed Weld SE way, by observing the `ContainerInitialized` event.
4. This method initializes all of the Swing components. Note the use of the `msgGenerator` here.
5. `guessButtonActionPerformed` is called when the 'Guess' button is clicked, and it does the following:
   - Gets the guess entered by the user and sets it as the current guess in the `Game`
   - Calls `game.check()` to validate and perform one 'turn' of the game
   - Calls `refreshUI`. If there were validation errors with the input, this will have been captured during `game.check()` and as such will be reflected in the messages returned by `MessageGenerator` and subsequently presented to the user. If there are no validation errors then the user will be told to guess again (higher or lower) or that the game has ended either in a win (correct guess) or a loss (ran out of guesses).
6. `replayBtnActionPerformed` simply calls `game.reset()` to start a new game and refreshes the messages in the UI.

7.3. L'exemple translator (traducteur) en profondeur

L'exemple translator prendra n'importe quelles phrases que vous entrerez, et les traduiront en Latin. (En fait, pas vraiment, mais le stub est là pour que vous l'implémentiez, au pire. Bonne chance !)

L'exemple translator est produit comme un ear et contient des EJBs. A la suite, sa structure est plus complexe que l'exemple numberguess.

Note

Java EE 6, qui intègre EJB 3.1, vous permet de packager des EJBs dans un war, qui rendra cette structure beaucoup plus simple ! Utiliser un ear offre d'autres avantages.

D'abord, jetons un oeil à l'aggregator ear, qui est localisé dans le répertoire `ear` de l'exemple. Maven génère automatiquement le `application.xml` pour nous depuis cette configuration de plugin :

```
<plugin>
   <groupId
>org.apache.maven.plugins</groupId>
   <artifactId
>maven-ear-plugin</artifactId>
   <configuration>
      <modules>
```

```
         <webModule>
            <groupId
>org.jboss.weld.examples.jsf.translator</groupId>
            <artifactId
>weld-jsf-translator-war</artifactId>
            <contextRoot
>/weld-translator</contextRoot>
         </webModule>
      </modules>
   </configuration>
</plugin
>
```

This configuration overrides the web context path, resulting in this application URL: *http://localhost:8080/weld-translator*.

Note

Si vous n'avez pas utilisé Maven pour générer ces fichiers, vous aurez besoin de `META-INF/application.xml` :

```
<application version="5"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/application_5.xsd">

  <display-name
>weld-jsf-translator-ear</display-name>
  <description
>The Weld JSF translator example (ear)</description>

  <module>
    <web>
      <web-uri
>weld-translator.war</web-uri>
      <context-root
>/weld-translator</context-root>
    </web>
  </module>
  <module>
    <ejb
>weld-translator.jar</ejb>
  </module>
</application
>
```

Ensuite, jetons un oeil au war, qui est localisé dans le répertoire `war` de l'exemple. Tout comme dans l'exemple numberguess, nous avons un `faces-config.xml` for JSF 2.0 et un `web.xml` (pour activer JSF) sous WEB-INF, localisé à `src/main/webapp/WEB-INF`.

Le plus intéressant est la vue JSF utilisée pour traduire le texte. Tout comme dans l'exemple numberguess, nous avons un template, qui entoure la forme (omis ici par souci de concision) :

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

   <table>
      <tr align="center" style="font-weight: bold">
         <td>
            Your text
         </td>
         <td>
            Translation
         </td>
      </tr>
      <tr>
         <td>

  <h:inputTextarea id="text" value="#{translator.text}" required="true" rows="5" cols="80"/>
         </td>
         <td>
            <h:outputText value="#{translator.translatedText}"/>
         </td>
      </tr>
   </table>
   <div>
      <h:commandButton id="button" value="Translate" action="#{translator.translate}"/>
   </div>

</h:form
>
```

L'utilisateur peut entrer n'importe quel texte dans la zone de texte à gauche, et appuyer le bouton de traduction pour voir le résultat à droite.

Enfin, jetez un oeil au module EJB, qui est situé dans le répertoire `ejb` de l'exemple. Dans `src/main/resources/META-INF`, il a juste un `beans.xml` vide, utilisé pour marquer l'archive comme contenant des beans.

Nous avons gardé le plus intéressant pour la fin, le code ! Le projet a deux beans simples, `SentenceParser` et `TextTranslator` et deux beans session, `TranslatorControllerBean` et `SentenceTranslator`. Vous devriez être maintenant assez familier , nous allons donc mettre en évidence les choses les plus intéressantes ici.

Tout deux, `SentenceParser` et `TextTranslator` sont des beans dépendants, et `TextTranslator` utilise l'injection par dépendance :

```
public class TextTranslator implements Serializable {

   private SentenceParser sentenceParser;

   @EJB private Translator translator;

   @Inject public TextTranslator(SentenceParser sentenceParser) {
      this.sentenceParser = sentenceParser;
   }
```

```
   public String translate(String text) {
      StringBuilder sb = new StringBuilder();
      for (String sentence: sentenceParser.parse(text)) {
         sb.append(translator.translate(sentence)).append(". ");
      }
      return sb.toString().trim();
   }

}
```

`TextTranslator` utilise le bean simple (vraiment juste une pure classe Java !) `SentenceParser` pour analyser les phrases et ensuite appelle le beans sans état avec l'interface métier locale `Translator` pour effectuer la traduction. C'est là que la magie arrive. Bien sur, nous ne pouvions pas développer un traducteur complet, mais il est suffisamment convaincant pour ceux qui ne comprennent pas le Latin !

```
@Stateless
public class SentenceTranslator implements Translator {

   public String translate(String sentence) {
      return "Lorem ipsum dolor sit amet";
   }

}
```

Enfin, il y a le contrôleur orienté UI. Il s'agit d'un bean session sans état, de portée requête et nommé, qui injecte le traducteur. Il récupère le texte de l'utilisateur et l'envoie au traducteur. Le bean a aussi des getters et des setters pour tous les attributs sur la page.

```
@Stateful
@RequestScoped
@Named("translator")
public class TranslatorControllerBean implements TranslatorController {

   @Inject private TextTranslator translator;

   private String inputText;

   private String translatedText;

   public void translate() {
      translatedText = translator.translate(inputText);
   }

   public String getText() {
      return inputText;
   }

   public void setText(String text) {
      this.inputText = text;
   }

   public String getTranslatedText() {
      return translatedText;
```

```
   }

   @Remove public void remove() {}

}
```

Ceci conclu notre courte visite des exemples Weld de démarrage. Pour en savoir plus sur Weld, visitez *http://www.seamframework.org/Weld*.

Partie III. Découplage et typage fort

Le premier thème majeur de CDI est *couplage lâche*. Nous avons déjà vu trois moyens de réaliser un couplage lâche :

- les *alternatives* qui activent le polymorphisme au déploiement,
- les *méthodes de production* qui activent le polymorphisme à l'exécution, et
- *contextual lifecycle management* decouples bean lifecycles.

Ces techniques servent à permettre un couplage lâche de l'appelant et de l'appelé. L'appelant n'est ni étroitement lié à une implémentation d'une interface, ni requis pour gérer le cycle de vie de l'implémentation. Cette approche permet *aux objets sans état d'interagir comme s'ils étaient des services*.

Le couplage lâche rend un système plus *dynamique*. Le système peut réagir aux changements d'une manière bien définie. Dans le passé, les frameworks qui ont tenté de fournir les aides énumérées ci-dessus le faisaient invariablement en sacrifiant la sécurité de type (notamment en utilisant des descripteurs XML). CDI est la première technologie, et certainement la première spécification de la plateforme Java EE, qui atteint ce niveau de couplage lâche de façon typée.

CDI fournit trois importantes aides supplémentaires dans l'objectif d'un couplage lâche :

- les *intercepteurs* qui découplent les problèmes techniques de la logique métier,
- les *décorateurs* qui peuvent être utiliser pour découpler certains problèmes métier, et
- les *notifications d'évènements* qui découplent les producteurs d'évènements des consommateurs d'évènements.

The second major theme of CDI is *strong typing*. The information about the dependencies, interceptors and decorators of a bean, and the information about event consumers for an event producer, is contained in typesafe Java constructs that may be validated by the compiler.

You don't see string-based identifiers in CDI code, not because the framework is hiding them from you using clever defaulting rules—so-called "configuration by convention"—but because there are simply no strings there to begin with!

Le bénéfice évident de cette approche est que *tout* IDE peut fournir de l'autocomplétion, de la validation et du refactoring sans avoir besoin d'un outillage spécial. Mais il y a un second bénéfice, moins évident. Quand vous pensez à identifier des objets, des évènements ou des intercepteurs via annotations à la place de noms, vous avez l'opportunité d'élever le niveau sémantique de votre code.

CDI vous encourage à développer des annotations qui modélisent des concepts, par exemple,

- `@Asynchronous`,
- `@Mock`,
- `@Secure` ou
- `@Updated`,

au lieu d'utiliser des noms composés comme

- `asyncPaymentProcessor`,
- `mockPaymentProcessor`,

- `SecurityInterceptor` ou
- `DocumentUpdatedEvent`.

Les annotations sont réutilisables. Elles aident à décrire des qualités communes de parties disparates du système. Elles nous aident à catégoriser et à comprendre notre code. Elles nous aident à traiter des problèmes communs d'une manière commune. Elles rendent notre code plus litéral et plus compréhensible.

CDI *stereotypes* take this idea a step further. A stereotype models a common *role* in your application architecture. It encapsulates various properties of the role, including scope, interceptor bindings, qualifiers, etc, into a single reusable package. (Of course, there is also the benefit of tucking some of those annotations away).

Nous sommes maintenant prêts à répondre à quelques fonctionnalités plus avancées de CDI. Gardez à l'esprit que ces fonctionnalités existent pour rendre notre code à la fois plus facile à valider et plus compréhensible. La plupart du temps vous n'aurez jamais vraiment *besoin* d'utiliser ces fonctions, mais si vous les utilisez à bon escient, vous pourrez en venir à apprécier leur puissance.

Méthodes de production

Les méthodes de production nous permettent de surmonter certaines limitations qui surviennent quand un conteneur, au lieu de l'application est responsable de l'instanciation des objets. Ils sont aussi la façon la plus simple pour intégrer des objets qui ne sont pas des beans dans l'environnement CDI.

D'après les spécifications :

> Une méthode de production agit comme une source pour les objets à injecter, où :
>
> - les objets à injecter ne sont pas requis pour être des instances de bean,
> - le type concret des objets à injecter peut varier à l'exécution ou
> - les objets nécessitent certaines initialisations personnalisées qui ne sont pas effectuées par le constructeur de bean

Par exemple, les méthodes de production nous permettent :

- d'exposer un entité JPA en tant que bean,
- d'exposer n'importe quelle classe du JDK en tant que bean,
- de définir plusieurs beans, avec des portées ou initialisations différentes, pour une même implémentation, ou
- de changer l'implémentation d'un bean à l'exécution.

En particulier, les méthodes de production nous permettent d'utiliser le polymorphisme à l'exécution avec CDI. Comme nous l'avons vu, les beans alternatifs sont une solution au problème de polymorphisme au déploiement. Mais une fois que le système est déployé, l'implémentation de CDI est fixée. Une méthode de production n'a pas une telle limitation :

```
@SessionScoped
public class Preferences implements Serializable {
   private PaymentStrategyType paymentStrategy;
   ...
   @Produces @Preferred
   public PaymentStrategy getPaymentStrategy() {
       switch (paymentStrategy) {
           case CREDIT_CARD: return new CreditCardPaymentStrategy();
           case CHECK: return new CheckPaymentStrategy();
           case PAYPAL: return new PayPalPaymentStrategy();
           default: return null;
       }
   }
}
```

Considérons un point d'injection :

```
@Inject @Preferred PaymentStrategy paymentStrategy;
```

Ce point d'injection a le même type ainsi que les annotations qualifiantes que la méthode de production, il est ainsi résolu par la méthode de production en utilisant les règles d'injection usuelles de CDI. La méthode de production sera appelée par le conteneur pour obtenir une instance pour fournir ce point d'injection.

8.1. Portée d'une méthode de production

La portée d'une méthode de production est par défaut à `@Dependent`, et ainsi elle sera appelée *chaque fois* que le conteneur injectera ce champ ou tout autre champ qui sera résolu par la même méthode de production. Ainsi, il pourrait y avoir plusieurs instance pour l'objet `PaymentStrategy` pour chaque session utilisateur.

Pour changer ce comportement, nous pouvons ajouter l'annotation `@SessionScoped` à la méthode.

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
   ...
}
```

Maintenant, quand la méthode de production est appelée, le `PaymentStrategy` retourné sera lié au contexte de la saison. La méthode de production ne sera pas rappelée dans la même session.

Note

La méthode de production ne doit *pas* hériter de la portée du bean qui déclare la méthode. Il y a deux beans différents ici : la méthode de production, et le bean qui la déclare. La portée de la méthode de production détermine la fréquence d'appel de la méthode, et le cycle de vie des objets retournés par la méthode. La portée du bean qui déclare la méthode de production détermine le cycle de vie sur lequel la méthode de production est appelée.

8.2. Injection dans les méthodes de production

Il existe un problème potentiel avec le code ci-dessus. Les implémentations de `CreditCardPaymentStrategy` sont instanciées en utilisant l'opérateur Java `new`. Les objets qui sont directement instanciés par l'application ne peuvent pas profiter des avantages de l'injection de dépendances et n'ont pas d'intercepteurs.

Si c'est n'est pas ce que nous voulons, nous pouvons utiliser l'injection de dépendances dans une méthode de production pour obtenir les instances.

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                          CheckPaymentStrategy cps,
                                          PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      case PAYPAL: return ppps;
      default: return null;
   }
}
```

Attendez, que se passe-t-il si `CreditCardPaymentStrategy` est de portée requête ? La méthode de production aura pour effet de "promouvoir" l'instance courante de portée requête en portée session. Il existe peur de chance que ce ne soit pas un bug ! L'objet de portée requête sera détruit par le conteneur avant que la session termine, mais la référence de l'objet sera laissée "en suspens" dans la session. Cette erreur ne sera *pas* détectée par le

conteneur, alors s'il vous plait faites encore plus attention quand vous retournez des instances de bean depuis des méthodes de production !

Il y existe au moins trois façon pour résoudre ce problème. Nous pouvons changer la portée de l'implémentation `CreditCardPaymentStrategy`, mais cela aura pour des effets pour les autres utilisateurs de ce bean. Une meilleure option serait de changer la portée de la méthode de production en `@Dependent` ou `@RequestScoped`.

Mais une solution encore plus courante est d'utiliser l'annotation qualifiante spécifique `@New`.

8.3. Utilisation de `@New` avec les méthodes de production

Examinez la méthode de production suivante :

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(@New CreditCardPaymentStrategy ccps,
                                          @New CheckPaymentStrategy cps,
                                          @New PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      case PAYPAL: return ppps;
      default: return null;
   }
}
```

Then a new *dependent* instance of `CreditCardPaymentStrategy` will be created, passed to the producer method, returned by the producer method and finally bound to the session context. The dependent object won't be destroyed until the `Preferences` object is destroyed, at the end of the session.

8.4. Méthodes de libération

Certaines méthodes de production retournent des objets qui nécessitent une destruction explicite. Par exemple, si on a besoin de fermer une connexion JDBC :

```
@Produces @RequestScoped Connection connect(User user) {
   return createConnection(user.getId(), user.getPassword());
}
```

La destruction peut être effectuée par une *méthode de libération* correspondante, définie par la même classe que la méthode de production :

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

La méthode de libération doit avoir au minimum un paramètre, annoté `@Disposes`, de même type et avec les mêmes qualifiants que la méthode de production. La méthode de libération est automatiquement appelée quand le contexte se termine (dans notre cas, la fin de la requête), et ce paramètre reçoit l'objet produit par la méthode de production. Si la méthode de libération a des paramètres supplémentaires, le conteneur cherchera un bean qui corresponde au type et aux qualifiants de chaque paramètre et le passera automatiquement à la méthode.

Intercepteurs

La fonctionnalité d'interception est définie dans la spécification Java Interceptors. CDI améliore cette fonctionnalité avec plus de sophistication, de sémantique, et avec une approche basée sur les annotations pour lier les intercepteurs aux beans.

La spécification des intercepteurs définie deux types de points d'interception :

- l'interception de méthodes métier, et
- l'interception de callback du cycle de vie.

En outre, la spécification EJB définit l'interception de méthodes timeout.

Un *intercepteur de méthodes métier* s'applique à des appels des méthodes d'un bean par les clients de ce bean :

```
public class TransactionInterceptor {
   @AroundInvoke
   public Object manageTransaction(InvocationContext ctx) throws Exception { ... }
}
```

Un *intercepteur de callback du cycle de vie* s'applique à des appels des callback du cycle de vie par le conteneur :

```
public class DependencyInjectionInterceptor {
   @PostConstruct
   public void injectDependencies(InvocationContext ctx) { ... }
}
```

Une classe intercepteur peut intercepter à la fois les callbacks du cycle de vie et les méthodes métier.

Un *intercepteur de méthode timeout* s'applique à des appels de méthodes timeout d'EJB par le conteneur :

```
public class TimeoutInterceptor {
   @AroundTimeout
   public Object manageTransaction(InvocationContext ctx) throws Exception { ... }
}
```

9.1. Liaisons d'intercepteur

Supposons que nous voulons déclarer que certains de nos beans soient transactionnels. La première chose dont nous avons besoin est un *interceptor binding type* pour préciser exactement quels beans nous intéresse :

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {}
```

Maintenant, nous pouvons facilement spécifier que notre `ShoppingCart` est un objet transactionnel :

```
@Transactional
public class ShoppingCart { ... }
```

Ou, si vous préférez, vous pouvez préciser que juste une seule méthode soit transactionnelle :

```
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

9.2. Implémenter les intercepteurs

C'est génial, mais nous allons bien devoir implémenter quelque part l'intercepteur qui fournit cet aspect de gestion des transactions. Tout ce que nous devons faire est de créer un intercepteur standard, et l'annoter avec `@Interceptor` et `@Transactional`.

```
@Transactional @Interceptor
public class TransactionInterceptor {
   @AroundInvoke
   public Object manageTransaction(InvocationContext ctx) throws Exception { ... }
}
```

Les intercepteurs peuvent profiter de l'injection de dépendances :

```
@Transactional @Interceptor
public class TransactionInterceptor {

    @Resource UserTransaction transaction;

    @AroundInvoke
    public Object manageTransaction(InvocationContext ctx) throws Exception { ... }

}
```

Plusieurs intercepteurs peuvent utiliser le même type d'interceptor binding.

9.3. Activer les intercepteurs

Par défaut, tous les intercepteurs sont désactivés. Nous devons *activer* notre intercepteur dans le descripteur `beans.xml` d'une bean archive. Cette activation s'applique seulement aux beans de cette archive.

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
```

```
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class
>org.mycompany.myapp.TransactionInterceptor</class>
   </interceptors>
</beans
>
```

Whoah ! Pourquoi cette daube d'XML ?

Eh bien, avoir la déclaration XML est en réalité une *bonne chose*. Cela résout deux problèmes :

- cela nous permet de spécifier totalement l'ordre pour tous les intercepteurs de notre système, en s'assurant d'un comportement déterministe, et
- cela nous permet d'activer ou désactiver des classes d'intercepteur au déploiement.

Par exemple, nous pouvons spécifier que notre intercepteur de sécurité s'exécute avant notre intercepteur de transaction.

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <interceptors>
      <class
>org.mycompany.myapp.SecurityInterceptor</class>
      <class
>org.mycompany.myapp.TransactionInterceptor</class>
   </interceptors>
</beans
>
```

Ou nous pouvons les désactiver tous les deux sur notre environnement de test en ne le mentionnant simplement pas dans `beans.xml` ! Ah, tellement simple.

9.4. Liaison d'intercepteur avec attributs

Supposons que nous voulons ajouter des informations supplémentaires à notre annotation `@Transactional` :

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {
   boolean requiresNew() default false;
}
```

CDI utilisera la valeur de `requiresNew` pour choisir entre deux intercepteurs différents, `TransactionInterceptor` et `RequiresNewTransactionInterceptor`.

```
@Transactional(requiresNew = true) @Interceptor
public class RequiresNewTransactionInterceptor {
   @AroundInvoke
   public Object manageTransaction(InvocationContext ctx) throws Exception { ... }
}
```

Maintenant, nous pouvons utiliser `RequiresNewTransactionInterceptor` de cette façon :

```
@Transactional(requiresNew = true)
public class ShoppingCart { ... }
```

Mais que se passe-t-il si nous n'avons qu'un intercepteur et que nous voulons que le conteneur ignore la valeur de `requiresNew` lors de la liaison des intercepteur ? Peut-être que cette information n'est utilise que pour l'implémentation de l'intercepteur. Nous pouvons utiliser l'annotation `@Nonbinding` :

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Secure {
   @Nonbinding String[] rolesAllowed() default {};
}
```

9.5. Plusieurs annotations de liaison d'intercepteur

Habituellement, nous utilisons une combinaison de liaisons d'intercepteurs pour lier plusieurs intercepteur à un bean. Par exemple, la déclaration suivante serait utilisée pour lier `TransactionInterceptor` et `SecurityInterceptor` au même bean :

```
@Secure(rolesAllowed="admin") @Transactional
public class ShoppingCart { ... }
```

Cependant, dans des cas vraiment complexes, un intercepteur peut lui-même spécifier une combinaison de liaisons d'intercepteur :

```
@Transactional @Secure @Interceptor
public class TransactionalSecureInterceptor { ... }
```

Cet intercepteur peut alors être lié à la méthode `checkout()` en utilisant n'importe quelle des combinaisons suivantes :

```
public class ShoppingCart {
   @Transactional @Secure public void checkout() { ... }
}
```

```
@Secure
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

```
@Transactional
public class ShoppingCart {
   @Secure public void checkout() { ... }
}
```

```
@Transactional @Secure
public class ShoppingCart {
   public void checkout() { ... }
}
```

9.6. Héritage de type de liaison d'intercepteur

Une limitation de la prise en charge des annotations par le langage Java est le manque d'héritage pour ces dernières. Vraiment, les annotations devraient pouvoir être réutilisées nativement, pour permettre le fonctionnement de ce genre de choses :

```
public @interface Action extends Transactional, Secure { ... }
```

Mais heureusement, CDI fonctionne malgré ce manque dans Java. Nous pouvons annoter une liaison d'intercepteur avec d'autres liaisons d'intercepteur (ceci est nommé une *meta-annotation*). Les liaisons d'intercepteur sont transitives — n'importe quel bean avec la première liaison d'intercepteur hérite des liaisons d'intercepteur déclarées comme meta-annotations.

```
@Transactional @Secure
@InterceptorBinding
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action { ... }
```

Maintenant, n'importe quel bean annoté `@Action` sera lié avec `TransactionInterceptor` et `SecurityInterceptor`. (Et même `TransactionalSecureInterceptor`, s'il existe.)

9.7. Utilisation de `@Interceptors`

L'annotation `@Interceptors` définie par la spécification des intercepteurs (et utilisée par les spécifications "managed bean" et EJB) est tout de même supporté dans CDI.

```
@Interceptors({TransactionInterceptor.class, SecurityInterceptor.class})
public class ShoppingCart {
```

```
    public void checkout() { ... }
}
```

Cependant, cette approche souffre des inconvénients suivants :

- l'implémentation de l'intercepteur est codé en dur dans le code métier,
- les intercepteurs ne peuvent être facilement désactivés au déploiement, et
- l'ordre des intercepteur n'est pas global — il est déterminé par l'ordre de lequel les intercepteurs sont listés au niveau de la classe.

Par conséquent, nous recommandons l'usage des liaisons d'intercepteur à la sauce CDI.

Décorateurs

Les intercepteurs sont un puissant moyen de capturer et de séparer les préoccupations qui sont *orthogonales* à l'application (et système de type). Tout intercepteur est capable d'intercepter les invocations de tout type Java. Cela les rend parfaits pour résoudre les problèmes techniques tels que la gestion des transactions, la sécurité et la journalisation des appels. Cependant, par nature, les intercepteurs ne sont pas conscients de la sémantique réelle des événements qu'ils interceptent. Ainsi, les intercepteurs ne sont pas un outil approprié pour séparer des préoccupations liées au métier.

The reverse is true of *decorators*. A decorator intercepts invocations only for a certain Java interface, and is therefore aware of all the semantics attached to that interface. Since decorators directly implement operations with business semantics, it makes them the perfect tool for modeling some kinds of business concerns. It also means that a decorator doesn't have the generality of an interceptor. Decorators aren't able to solve technical concerns that cut across many disparate types. Interceptors and decorators, though similar in many ways, are complementary. Let's look at some cases where decorators fit the bill.

Supposons que nous ayons une interface représentant des comptes :

```
public interface Account {
   public BigDecimal getBalance();
   public User getOwner();
   public void withdraw(BigDecimal amount);
   public void deposit(BigDecimal amount);
}
```

Plusieurs beans différents implémentent l'interface `Account` dans notre système. Nous avons cependant une obligation légale qui nous oblige à enregistrer les grosses transactions faites sur le système dans un journal spécial, et ce, pour tout type de compte. Ceci est un cas d'utilisation idéal d'un décorateur.

Un décorateur est un bean (peut-être même une classe abstraite) qui implémente le type qu'il décore, et qui est annoté `@Decorator`.

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {
   ...
}
```

Le décorateur implémente les méthodes du type décoré qu'il veut intercepter.

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {
   @Inject @Delegate @Any Account account;

   @PersistenceContext EntityManager em;

   public void withdraw(BigDecimal amount) {
      ...
   }
```

```
   public void deposit(BigDecimal amount);
      ...
   }
}
```

Contrairement aux autres beans, un décorateur peut être une classe abstraite. Donc, s'il n'y a rien de spécial que le décorateur doit faire pour une méthode particulière de l'interface décorée, vous n'avez pas besoin d'implémenter cette méthode.

Les intercepteurs pour une méthode sont appelés avant les décorateurs qui s'appliquent à cette méthode.

10.1. Delegate object

Decorators have a special injection point, called the *delegate injection point*, with the same type as the beans they decorate, and the annotation `@Delegate`. There must be exactly one delegate injection point, which can be a constructor parameter, initializer method parameter or injected field.

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {
   @Inject @Delegate @Any Account account;
   ...
}
```

Un décorateur est lié à tout bean qui :

- has the type of the delegate injection point as a bean type, and
- has all qualifiers that are declared at the delegate injection point.

This delegate injection point specifies that the decorator is bound to all beans that implement `Account`:

```
@Inject @Delegate @Any Account account;
```

A delegate injection point may specify any number of qualifier annotations. The decorator will only be bound to beans with the same qualifiers.

```
@Inject @Delegate @Foreign Account account;
```

The decorator may invoke the delegate object, which has much the same effect as calling `InvocationContext.proceed()` from an interceptor. The main difference is that the decorator can invoke *any* business method on the delegate object.

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {
   @Inject @Delegate @Any Account account;
```

```
   @PersistenceContext EntityManager em;

   public void withdraw(BigDecimal amount) {
      account.withdraw(amount);
      if ( amount.compareTo(LARGE_AMOUNT)
>0 ) {
         em.persist( new LoggedWithdrawl(amount) );
      }
   }

   public void deposit(BigDecimal amount);
      account.deposit(amount);
      if ( amount.compareTo(LARGE_AMOUNT)
>0 ) {
         em.persist( new LoggedDeposit(amount) );
      }
   }
}
```

10.2. Activation des décorateurs

By default, all decorators are disabled. We need to *enable* our decorator in the `beans.xml` descriptor of a bean archive. This activation only applies to the beans in that archive.

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <decorators>
         <class
>org.mycompany.myapp.LargeTransactionDecorator</class>
   </decorators>
</beans
>
```

Cette déclaration a le même objectif pour les décorateur que la déclaration `<interceptors>` qui sert aux intercepteurs :

- cela nous permet de spécifier un ordre précis pour tous les décorateurs dans notre système, assurant un comportement déterministe, et
- cela nous permet d'activer ou désactiver les classes de décorateurs au moment du déploiement.

Evènements

L'injection de dépendances permet le couplage lâche en permettant la variation de l'implémentation du type injecté, soit au déploiement soit à l'exécution. Les évènements vont un cran plus loin, permettant aux beans de n'avoir aucune dépendance du tout à la compilation. Les *producteurs* lèvent les évènements qui sont délivrés aux *observateurs* d'évènements par le conteneur.

Ce schéma de base peut ressembler à l'habituel pattern observeur/observable, mais il y a quelques modifications :

- non seulement les producteurs d'évènements sont découplés des observauters; mais les les observateurs sont complèment découplés des producteurs,
- les observateurs peuvent spécifier en ensemble de "sélecteurs" pour réduire le nombre d'évènements qu'ils vont recevoir, et
- les observateurs peuvent être notifiés immédiatement, ou peuvent spécifier que la délivrance d'un évènement devrait être retardée jusqu'à la fin de la transaction courante.

La notification d'évènements CDI utilise plus ou moins la même approche typesafe que nous avons déjà vu avec le service d'injection de dépendances.

11.1. Producteurs d'évènements

L'objet event porte l'état du producteur au consommateur. L'objet event n'est rien d'autre qu'un instance d'une classe Java concrète. (La seule restriction est que le type d'un évènement ne peut pas contenir de variables). Un évènement peut avoir des qualifiants, ce qui permet aux observateurs de le distinguer des autres évènements du même type. La fonction des qualifiants est, comme les sélecteurs de topic, de permettre à l'observateur de réduire le nombre d'évènements qu'il observe.

Un qualifiant d'évènement est juste un qualifiant normal, défini en utilisant `@Qualifier`. Voyons un exemple :

```
@Qualifier
@Target({FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Updated {}
```

11.2. Observateurs d'évènements

Une *méthode d'observation* est une méthode d'un bean avec un paramètre annoté `@Observes`.

```
public void onAnyDocumentEvent(@Observes Document document) { ... }
```

Le paramètre annoté est appelé le *paramètre évènement*. Le type du paramètre évènement est le *type de l'évènement* observé, dans notre exemple `Document`. Le paramètre évènement peut aussi spécifier de qualifiants.

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

Une méthode d'observation n'a pas besoin de spécifier un qualifiant—dans ce cas elle sera notifiée par *tous* les évènements d'un type donné. Si la méthode spécifie des qualifiants, elle ne sera intéressé que par les évènements ayant ces qualifiants.

The observer method may have additional parameters, which are injection points:

```
public void afterDocumentUpdate(@Observes @Updated Document document, User user) { ... }
```

11.3. Producteurs d'évènements

Les producteurs d'évènements lèvent les évènements en utilisant une instance de l'interface paramétrée `Event`. Une instance de cette interface est obtenue par injection :

```
@Inject @Any Event<Document
> documentEvent;
```

Un producteur lève des évènements en appelant la méthode `fire()` de l'interface `Event`, en passant un *objet évènement* :

```
documentEvent.fire(document);
```

Cet évènement particulier sera délivré à toutes les méthodes observatrices qui :

- ont un paramètre d'évènement pour lequel l'objet évènement (le `Document`) est assignable, et
- ne spécifient aucun qualifiant.

Le conteneur appelle simplement toutes les méthodes d'observation, en passant l'objet évènement comme valeur du paramètre évènement. Si une méthode d'observation lance une exception, le conteneur arrête d'appeler les méthodes d'observation, et l'exception est relancée par la méthode `fire()`.

Les qualifiants peuvent être appliqués a un évènement en suivant l'une de ces deux façons :

- en annotant le point d'injection de `Event`, ou
- en passant les qualifiants à la méthode `select()` d'`Event`.

Spécifier les qualifiants au point d'injection est beaucoup plus simple :

```
@Inject @Updated Event<Document
> documentUpdatedEvent;
```

Ensuite, tous les évènements déclenchés par cette instance d'`Event` ont le qualifiant `@Updated`. L'évènement sera délivré à toutes méthodes d'observation qui :

- a un paramètre d'évènement pour lequel l'objet évènement est assignable, et

- ne spécifie aucune qualifiant d'évènement *excepté* pour les qualifiants d'évènements qui correspondent à ceux spécifiés sur le point d'injection d'`Event`.

L'inconvénient d'annoter le point d'injection est que nous ne pouvons pas spécifier le qualifiant dynamiquement. CDI nous laisse obtenir une instance de qualifiant en surchargeant la classe `AnnotationLiteral`. De cette façon, nous pouvons passer le qualifiant à la méthode `select()` d'`Event`.

```
documentEvent.select(new AnnotationLiteral<Updated
>(){}).fire(document);
```

Les évènements peuvent avoir plusieurs qualifiants, assemblés en utilisant une combinaison d'annotations sur le point d'injection d'`Event` et les instances des qualifiants passées à la méthode `select()`.

11.4. Méthodes d'observation transactionnelles

Par défaut, il n'y a pas d'observateur dans le contexte courant, le conteneur instanciera l'observateur dans le but de lui envoyer un évènement. Ce comportement n'est pas toujours désiré. Nous voulons peut-être envoyer uniquement des évènements aux instances d'observateur qui existent déjà dans les contextes courants.

A conditional observer is specified by adding `receive = IF_EXISTS` to the `@Observes` annotation.

```
public void refreshOnDocumentUpdate(@Observes(receive = IF_EXISTS) @Updated Document d) { ... }
```

Un bean avec la visibilité `@Dependent` ne peut pas être un observateur conditionnel car il ne serait jamais appelé!

11.5. Qualifiant d'évènement avec des membres

An event qualifier type may have annotation members:

```
@Qualifier
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Role {
   RoleType value();
}
```

La valeur du membre est utilisée pour préciser d'avantage les messages délivrés à l'observateur :

```
public void adminLoggedIn(@Observes @Role(ADMIN) LoggedIn event) { ... }
```

Les membres des qualifiants d'évènement peuvent être spécifiés statiquement par le producteur d'évènement, par des annotations au point d'injection du notificateur d'évènement :

```
@Inject @Role(ADMIN) Event<LoggedIn
> loggedInEvent;
```

Alternatively, the value of the event qualifier type member may be determined dynamically by the event producer. We start by writing an abstract subclass of `AnnotationLiteral`:

```
abstract class RoleBinding
   extends AnnotationLiteral<Role
>
   implements Role {}
```

Le producteur d'évènement passe une instance de cette classe à `select()` :

```
documentEvent.select(new RoleBinding() {
   public void value() { return user.getRole(); }
}).fire(document);
```

11.6. Plusieurs qualifiants d'évènements

Event qualifier types may be combined, for example:

```
@Inject @Blog Event<Document
> blogEvent;
...
if (document.isBlog()) blogEvent.select(new AnnotationLiteral<Updated
>(){}).fire(document);
```

Les observateurs doivent totalement correspondre au type qualifié de l'évènement. Prenons les observateurs de l'exemple ci-après :

```
public void afterBlogUpdate(@Observes @Updated @Blog Document document) { ... }
```

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

```
public void onAnyBlogEvent(@Observes @Blog Document document) { ... }
```

```
public void onAnyDocumentEvent(@Observes Document document) { ... }}}
```

Le seul observateur notifié sera :

```
public void afterBlogUpdate(@Observes @Updated @Blog Document document) { ... }
```

Cependant, s'il y avait aussi un observateur :

```
public void afterBlogUpdate(@Observes @Any Document document) { ... }
```

Il serait aussi notifié étant donné que @Any agit comme un alias pour n'importe quel qualifiant.

11.7. Observateurs transactionnels

Les Observateurs transactionnels recoivent des notifications d'évènements avant ou après la fin d'une phase de la transaction dans laquelle l'évènement a été levé. Par exemple, la méthode d'observation suivante a besoin de rafraîchir un ensemble de résultat de requête qui est cachée dans le contexte de l'application, mais seulement quand les transactions qui mettent à jour l'arbre `Category` réussissent :

```
public void refreshCategoryTree(@Observes(during = AFTER_SUCCESS) CategoryUpdateEvent event) { ... }
```

Il y a cinq sortes d'observateurs transactionnels :

- Les observateurs `IN_PROGESS` sont immédiatement appelés (par défaut)

  Les observateurs `AFTER_SUCCESS` sont appelés après la fin d'une phase de la transaction, mais seulement si la transaction se termine avec succès

- Les observateurs `AFTER_FAILURE` sont appelés après la fin d'une phase de la transaction, mais seulement si la transaction ne se termine pas correctement
- Les observateurs `AFTER_COMPLETION` sont appelés après la fin de la phase de la transaction
- Les observateurs `BEFORE_COMPLETION` sont appelés avant la fin de la phase de la transaction

Les observateurs transactionels sont très importants dans un modèle d'objets à état, parce que l'état est souvent maintenu plus longtemps qu'une simple transaction atomique.

Imaginez que nous ayons caché un ensemble de résultats de requête JPA dans la portée application :

```
@ApplicationScoped @Singleton
public class Catalog {

   @PersistenceContext EntityManager em;

   List<Product
> products;

   @Produces @Catalog
   List<Product
> getCatalog() {
      if (products==null) {
         products = em.createQuery("select p from Product p where p.deleted = false")
            .getResultList();
      }
      return products;
   }
```

```
}
```

De temps en temps, un `Product` est créé ou détruit. Quand cela survient, nous avons besoin de rafraîchir le catalogue de `Product`. Mais nous devrions attendre jusqu'à *après* que la transaction se soit terminée avec succès avant d'exécuter ce rafraîchissement !

Le bean qui crée et détruit les `Products` pourrait lever des évènements, par exemple :

```
@Stateless
public class ProductManager {
   @PersistenceContext EntityManager em;
   @Inject @Any Event<Product
> productEvent;

   public void delete(Product product) {
      em.delete(product);
      productEvent.select(new AnnotationLiteral<Deleted
>(){}).fire(product);
   }

   public void persist(Product product) {
      em.persist(product);
      productEvent.select(new AnnotationLiteral<Created
>(){}).fire(product);
   }
   ...
}
```

Et maintenant `Catalog` peut observer les évènements après l'exécution avec succès de la transaction :

```
@ApplicationScoped @Singleton
public class Catalog {
   ...
   void addProduct(@Observes(during = AFTER_SUCCESS) @Created Product product) {
      products.add(product);
   }

   void removeProduct(@Observes(during = AFTER_SUCCESS) @Deleted Product product) {
      products.remove(product);
   }
}
```

Stéréotypes

La spécification CDI définit un stéréotype de la façon suivante :

> In many systems, use of architectural patterns produces a set of recurring bean roles. A stereotype allows a framework developer to identify such a role and declare some common metadata for beans with that role in a central place.
>
> A stereotype encapsulates any combination of:
>
> - une portée par défaut, et
> - a set of interceptor bindings.
>
> Un stéréotype peut aussi définir que :
>
> - tous beans avec un stéréotype ont des noms EL par défaut, ou que
> - tous beans avec un stéréotype sont des alternatives.
>
> Un bean peut déclarer zéro, un ou plusieurs stéréotypes. Les annotations stéréotype peuvent être appliquées sur une classe or méthode de production ou un attribut.

A stereotype is an annotation, annotated `@Stereotype`, that packages several other annotations. For instance, the following stereotype identifies action classes in some MVC framework:

```
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
...
public @interface Action {}
```

Nous utilisons le stéréotype en appliquant l'annotation à un bean.

```
@Action
public class LoginAction { ... }
```

Evidemment, nous devons appliquer d'autres annotations à notre stéréotype ou sinon il n'ajoutera pas plus de valeur.

12.1. Portée par défaut d'un stéréotype

Un stéréotype peu définir une portée par défaut pour les beans annotés avec le stéréotype. Par exemple :

```
@RequestScoped
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Action {}
```

Une action particulière peut toujours surcharger cette valeur par défaut, si nécessaire :

```
@Dependent @Action
public class DependentScopedLoginAction { ... }
```

Naturellement, la surcharge d'une seule valeur par défaut n'est pas très utile. Mais souvenez-vous, les stéréotypes peuvent définir plus simplement la portée par défaut.

12.2. Branchement d'interception pour les stéréotypes

A stereotype may specify a set of interceptor bindings to be inherited by all beans with that stereotype.

```
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Action {}
```

This helps us get technical concerns, like transactions and security, even further away from the business code!

12.3. Nom par défaut avec les stéréotypes

Vous pouvez spécifier que tous les beans avec un certain stéréotype ont un nom EL par défaut quand un nom n'est pas explicitement défini pour ce bean. Tout ce que nous devons faire est d'ajouter une annotation `@Named` sans paramètre :

```
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Action {}
```

Maintenant, le bean `LoginAction` aura `loginAction` comme nom par défaut.

12.4. Stéréotypes alternatifs

A stereotype can indicate that all beans to which it is applied are `@Alternatives`. An *alternative stereotype* lets us classify beans by deployment scenario.

```
@Alternative
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
```

```
public @interface Mock {}
```

We can apply an alternative stereotype to a whole set of beans, and activate them all with one line of code in `beans.xml`.

```
@Mock
public class MockLoginAction extends LoginAction { ... }
```

12.5. Empilage de stéréotypes

This may blow your mind a bit, but stereotypes may declare other stereotypes, which we'll call *stereotype stacking*. You may want to do this if you have two distinct stereotypes which are meaningful on their own, but in other situation may be meaningful when combined.

Voici un exemple qui combine les stéréotypes `@Action` et `@Auditable` :

```
@Auditable
@Action
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
public @interface AuditableAction {}
```

12.6. Stéréotypes intégrés

Nous avons déjà rencontré deux stéréotypes standards définis par la spécification CDI : `@Interceptor` and `@Decorator`.

CDI définit un stéréotype par défaut supplémentaire, `@Model`, qui devrait être fréquemment utilisé dans les applications web :

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Model {}
```

Au lieu d'utiliser les managed beans de JSF, annotez juste un bean avec `@Model`, et utilisez le directement dans votre vue JSF !

Spécialisation, héritage et alternatives

Quand vous commencerez à développer pour la première fois avec CDI, vous aurez probablement qu'une seule implémentation pour chaque type de bean. Dans ce cas, il est facile de comprendre comment les beans sont sélectionnés pour être injectés. Comme la complexité de votre application grandit, les occurrences multiples du même type de bean commencent à apparaître, soit parce que vous avez plusieurs implémentations soit deux beans qui partagent un héritage (Java) commun. C'est à ce moment que vous avez à commencer à étudier les règles de la spécialisation, de l'héritage et des alternatives pour travailler avec les dépendances insatisfaites ou ambiguës ou pour éviter que certains beans soient appelés.

La spécification CDI reconnait deux scénarios distincts dans lesquels un bean étend un autre :

- Le second bean *spécialise* le premier bean dans certains scénarios de déploiement. Dans ces déploiements, le second bean replace complètement le premier, accomplissant le même rôle dans le système.
- Le second bean replace simplement l'implémentation Java, et porte par ailleurs aucune relation avec le premier bean. Le premier bean peut même ne pas avoir été conçu pour être utilisé comme un objet contextuel.

Le second cas est celui par défaut dans CDI. Il est possible d'avoir deux beans dans le système de même type (interface ou classe parente). Comme vous l'avez appris, vous choisissez entre les deux implémentations en utilisant les qualifiants.

Le premier cas est l'exception, et requiert aussi plus d'attention. Dans tous déploiements concernés, seulement un bean peut accomplir un rôle donné à la fois. Ceci signifie qu'un bean doit être activé et un autre désactivé. Il y a deux modificateurs impliqués : `@Alternative` et `@Specializes`. Nous commencerons par regarder les alternatives et ensuite montrer les garanties que la spécialisation ajoute.

13.1. Utiliser les stéréotypes alternatifs

CDI vous permet de *surcharger* l'implémentation d'un type au déploiement en utilisant une alternative. Par exemple, le bean suivant fournit une implémentation par défaut de l'interface `PaymentProcessor` :

```
public class DefaultPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Mais dans notre environnement de pré-production, nous ne voulons pas vraiment soumettre des paiements au système externe, c'est pourquoi nous surchargeons l'implémentation de `PaymentProcessor` avec un bean différent :

```
public @Alternative
class StagingPaymentProcessor
      implements PaymentProcessor {
   ...
```

```
}
```

or

```
public @Alternative
class StagingPaymentProcessor
      extends DefaultPaymentProcessor {
   ...
}
```

Nous avons déjà vu comment vous pouvez activer cette alternative en l'inscrivant dans le descripteur `beans.xml`.

Mais supposons que nous avons plusieurs alternatives dans l'environnement de pré-production. Il pourrait être plus pratique de pouvoir tous les activer d'un coup. Nous allons donc faire de `@Staging` un stéréotype `@Alternative` et annoté le bean de pré-production avec ce stéréotype. Vous verrez comment ce niveau d'indirection est payante. D'abord, nous créons le stéréotype :

```
@Alternative
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface Staging {}
```

Ensuite, nous replaçons l'annotation `@Alternative` de notre bean par `@Staging` :

```
@Staging
public class StagingPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Enfin, nous activons le stéréotype `@Staging` dans le descripteur `beans.xml` :

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
   <alternatives>
         <stereotype
>org.mycompany.myapp.Staging</stereotype>
   </alternatives>
</beans
>
```

Maintenant, peu importe le nombre de beans de pré-production, ils seront tous activés en même temps.

13.2. Un petit problème avec les alternatives

Quand nous activons une alternative, est-ce que cela signifie que l'implémentation par défaut est désactivée ? Et bien, pas exactement. Si l'implémentation par défaut a un qualifiant, par exemple `@LargeTransaction`, et que l'alternative ne l'a pas, vous pourrez toujours injecté l'implémentation par défaut.

```
@Inject @LargeTransaction PaymentProcessor paymentProcessor;
```

Nous avons donc complètement remplacé l'implémentation par défaut dans le déploiement du système. La seule façon pour qu'un bean puisse complètement surcharger un autre bean sur tous les points d'injection est d'implémenter tous les types de bean et déclarer tous les qualifiants du second bean. Cependant, si le second bean déclare une méthode de production ou une méthode d'observation, alors ce ne sera pas assez pour garantir que le second bean ne sera jamais appelé ! Nous avons besoin de quelque chose en plus.

CDI fournit une fonctionnalité spéciale, appelée *spécialisation*, qui aide le développeur d'éviter ces pièges. La spécialisation est un moyen d'informer le système de votre intention de complètement replacer et désactiver une implémentation d'un bean.

13.3. Utiliser la spécialisation

Quand le but est de remplacer une implémentation par un autre, pour éviter les erreurs de développement, le premier bean devrait :

- étendre directement de la classe du second bean, ou
- surcharger directement la méthode de production, dans ce cas le second bean est une méthode de production, et enfin

déclarer explicitement qu'il *spécialise* le second bean :

```
@Alternative @Specializes
public class MockCreditCardPaymentProcessor
      extends CreditCardPaymentProcessor {
   ...
}
```

Quand un bean activé en spécialise un autre, ce dernier n'est jamais instancié ou appelé par le conteneur. Même s'il définit une méthode de production ou d'observation, la méthode ne sera jamais appelée.

Alors, pourquoi la spécialisation fonctionne, et qu'est ce que cela a à voir avec l'héritage ?

Une fois que nous aurons informé le conteneur que notre bean alternatif est destiné à se présenter comme remplaçant de l'implémentation par défaut, l'implémentation alternative hérite automatiquement de tous les qualifiants de l'implémentation par défaut. Ainsi, dans notre exemple, `MockCreditCardPaymentProcessor` hérite des qualifiants `@Default` et `@CreditCard`.

Par ailleurs, si l'implémentation par défaut déclare un nom EL en utilisant `@Named`, le nom est hérité par l'alternative spécialisée.

Java EE component environment resources

Java EE 5 already introduced some limited support for dependency injection, in the form of component environment injection. A component environment resource is a Java EE component, for example a JDBC datasource, JMS queue or topic, JPA persistence context, remote EJB or web service.

Naturally, there is now a slight mismatch with the new style of dependency injection in CDI. Most notably, component environment injection relies on string-based names to qualify ambiguous types, and there is no real consistency as to the nature of the names (sometimes a JNDI name, sometimes a persistence unit name, sometimes an EJB link, sometimes a non-portable "mapped name"). Producer fields turned out to be an elegant adaptor to reduce all this complexity to a common model and get component environment resources to participate in the CDI system just like any other kind of bean.

Fields have a duality in that they can both be the target of Java EE component environment injection and be declared as a CDI producer field. Therefore, they can define a mapping from a string-based name in the component environment, to a combination of type and qualifiers used in the world of typesafe injection. We call a producer field that represents a reference to an object in the Java EE component environment a *resource*.

14.1. Définir une ressource

La spécification CDI utilise le terme *resource* pour référencer, de manière générique, n'importe quel genre d'objet qui peut être disponible dans l'environnement Java EE :

- JDBC `Datasources`, JMS `Queues`, `Topics` et `ConnectionFactorys`, JavaMail `Sessions` et les autres ressources transactionnelles en incluant les connecteurs JCA,
- JPA `EntityManagers` et `EntityManagerFactorys`,
- EJBs distants, et
- services web

We declare a resource by annotating a producer field with a component environment injection annotation: `@Resource`, `@EJB`, `@PersistenceContext`, `@PersistenceUnit` or `@WebServiceRef`.

```
@Produces @WebServiceRef(lookup="java:app/service/Catalog")
Catalog catalog;
```

```
@Produces @Resource(lookup="java:global/env/jdbc/CustomerDatasource")
@CustomerDatabase Datasource customerDatabase;
```

```
@Produces @PersistenceContext(unitName="CustomerDatabase")
@CustomerDatabase EntityManager customerDatabasePersistenceContext;
```

```
@Produces @PersistenceUnit(unitName="CustomerDatabase")
```

```
@CustomerDatabase EntityManagerFactory customerDatabasePersistenceUnit;
```

```
@Produces @EJB(ejbLink="../their.jar#PaymentService")
PaymentService paymentService;
```

L'attribut peut être static (mais non final).

Une déclaration de ressource contient réellement deux éléments d'information :

- the JNDI name, EJB link, persistence unit name, or other metadata needed to obtain a reference to the resource from the component environment, and
- le type et les qualifiants que nous utiliserons pour injecter la référence dans nos beans.

Note

Il pourrait sembler étrange de déclarer des ressources en code Java. N'est-ce pas ce genre de choses qui devraient être spécifiques au déploiement ? Certainement, et c'est pourquoi il est logique de déclarer vos ressources dans une classe annotée `@Alternative`.

14.2. Injection typesafe de ressource

Ces ressources peuvent maintenant être injectés de façon courante.

```
@Inject Catalog catalog;
```

```
@Inject @CustomerDatabase Datasource customerDatabase;
```

```
@Inject @CustomerDatabase EntityManager customerDatabaseEntityManager;
```

```
@Inject @CustomerDatabase EntityManagerFactory customerDatabaseEntityManagerFactory;
```

```
@Inject PaymentService paymentService;
```

Le type et les qualifiants de la ressource sont déterminés par la déclaration de l'attribut de production.

It might seem like a pain to have to write these extra producer field declarations, just to gain an additional level of indirection. You could just as well use component environment injection directly, right? But remember that you're going to be using resources like the `EntityManager` in several different beans. Isn't it nicer and more typesafe to write

```
@Inject @CustomerDatabase EntityManager
```

au lieu de

```
@PersistenceContext(unitName="CustomerDatabase") EntityManager
```

partout ?

Partie IV. CDI l'écosystème Java EE

The third theme of CDI is *integration*. We've already seen how CDI helps integrate EJB and JSF, allowing EJBs to be bound directly to JSF pages. That's just the beginning. The CDI services are integrated into the very core of the Java EE platform. Even EJB session beans can take advantage of the dependency injection, event bus, and contextual lifecycle management that CDI provides.

CDI est également conçu pour fonctionner de concert avec des technologies en dehors de la plateforme en fournissant des points d'intégration dans la plateforme Java EE via un SPI. Cet SPI place CDI comme la base d'un nouvel écosystème d'extensions *portables* et d'intégration avec des frameworks et technologies existantes. Les service CDI seront capables d'atteindre un ensemble de technologies diverses, tel que les moteurs de gestion des processus métier (BPM), les frameworks web existants et les modèles de composants standards de facto. Bien sur, la plateforme Java EE ne sera jamais capable de standardiser toutes les technologies intéressantes utilisées dans le monde Java, mais CDI rend plus facile l'utilisation de technologies qui ne font pas encore partie de la plateforme et cela de manière transparente dans l'environnement Java EE.

We're about to see how to take full advantage of the Java EE platform in an application that uses CDI. We'll also briefly meet a set of SPIs that are provided to support portable extensions to CDI. You might not ever need to use these SPIs directly, but don't take them for granted. You will likely be using them indirectly, every time you use a third-party extension, such as Seam.

Intégration dans Java EE

CDI is fully integrated into the Java EE environment. Beans have access to Java EE resources and JPA persistence contexts. They may be used in Unified EL expressions in JSF and JSP pages. They may even be injected into other platform components, such as servlets and message-driven Beans, which are not beans themselves.

15.1. Beans intégrés

Dans l'environnement Java EE, le conteneur fournis les beans intégrés suivants, tous avec le qualifiant `@Default` :

- la `UserTransaction` JTA courante,
- un `Principal` représentant l'identité de l'appelant courant,
- la `ValidationFactory` *Bean Validation* [http://jcp.org/en/jsr/detail?id=303] par défaut, et
- un `Validator` pour le `ValidationFactory` par défaut.

Note

La spécification CDI n'exige pas les objets du contexte de servlet, `HttpServletRequest`, `HttpSession` et `ServletContext` d'être exposé comme des beans injectables. Si vous voulez vraiment être capable d'injecter ces objets, il est facile de créer une extension portable pour les exposer comme des beans. Cependant, nous recommandons que l'accès direct à ces objets soit limité aux servlets, aux filtres de servlet et aux écouteurs d'évènements de servlet, où ils peuvent être obtenu de façon habituelle comme cela est défini par la spécification Java Servlets. Le `FacesContext` n'est par ailleurs par injectable. Vous pouvez l'obtenir en appelant `FacesContext.getCurrentInstance()`.

Note

Oh, vous voulez *vraiment* injecter le `FacesContext` ? D'accord ! Alors essayez cette méthode de production :

```
class FacesContextProducer {
   @Produces @RequestScoped FacesContext getFacesContext() {
      return FacesContext.getCurrentInstance();
   }
}
```

15.2. Injecter des ressources Java EE dans un bean

All managed beans may take advantage of Java EE component environment injection using `@Resource`, `@EJB`, `@PersistenceContext`, `@PeristenceUnit` and `@WebServiceRef`. We've already seen a couple of examples of this, though we didn't pay much attention at the time:

```
@Transactional @Interceptor
```

```
public class TransactionInterceptor {
   @Resource UserTransaction transaction;

  @AroundInvoke public Object manageTransaction(InvocationContext ctx) throws Exception { ... }
}
```

```
@SessionScoped
public class Login implements Serializable {
   @Inject Credentials credentials;
   @PersistenceContext EntityManager userDatabase;
    ...
}
```

The Java EE `@PostConstruct` and `@PreDestroy` callbacks are also supported for all managed beans. The `@PostConstruct` method is called after *all* injection has been performed.

Of course, we advise that component environment injection be used to define CDI resources, and that typesafe injection be used in application code.

15.3. Appeler un bean à partir d'une Servlet

Il est facile d'utiliser un bean depuis une Servlet dans Java EE 6. Injectez simplement le bean en utilisant la méthode d'injection par attribut ou par méthode d'initialisation.

```
public class Login extends HttpServlet {
   @Inject Credentials credentials;
   @Inject Login login;

   @Override
   public void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      credentials.setUsername(request.getParameter("username")):
      credentials.setPassword(request.getParameter("password")):
      login.login();
      if ( login.isLoggedIn() ) {
         response.sendRedirect("/home.jsp");
      }
      else {
         response.sendRedirect("/loginError.jsp");
      }
   }

}
```

Since instances of servlets are shared across all incoming threads, the bean client proxy takes care of routing method invocations from the servlet to the correct instances of `Credentials` and `Login` for the current request and HTTP session.

15.4. Appeler un bean à partir d'un Message-Driven Bean

L'injection CDI s'applique à tous les EJBs, même quand ils ne sont pas des beans managés. Plus précisément, vous pouvez utiliser l'injection CDI dans les Message-Driven Beans, qui ne sont pas des objets contextuels par nature.

Vous pouvez même utiliser des liaisons d'intercepteurs de CDI avec des Message-Driven Beans.

```
@Transactional @MessageDriven
public class ProcessOrder implements MessageListener {
   @Inject Inventory inventory;
   @PersistenceContext EntityManager em;

   public void onMessage(Message message) {
      ...
   }
}
```

Mais faites attention, il n'y a pas de session ni de contexte de conversation disponible lorsqu'un message est délivré à un Message-Driven Bean. Seuls les Web Beans `@RequestScoped` et `@ApplicationScoped` sont disponibles.

Mais que dire des beans qui *envoient* des messages JMS ?

15.5. Terminaisons JMS

Envoyer des messages en utilisant JMS peut être assez complexe, à cause du nombre d'objets différents que vous devez traiter. Pour les queues, nous avons `Queue`; `QueueConnectionFactory`, `QueueConnection`, `QueueSession` et `QueueSender`. Pour les topics nous avons `Topic`, `TopicConnectionFactory`, `TopicConnection`, `TopicSession` et `TopicPublisher`. Chacun de ces objets a son propre cycle de vie et son modèle de threads dont nous devons nous soucier.

Vous pouvez utiliser les attributs de productions ou les méthodes pour préparer toutes ces ressources afin de les injecter dans un bean :

```
public class OrderResources {
   @Resource(name="jms/ConnectionFactory")
   private ConnectionFactory connectionFactory;

   @Resource(name="jms/OrderQueue")
   private Queue orderQueue;

   @Produces @OrderConnection
   public Connection createOrderConnection() throws JMSException {
    return connectionFactory.createConnection();
   }

   public void closeOrderConnection(@Disposes @OrderConnection Connection connection)
         throws JMSException {
      connection.close();
   }

   @Produces @OrderSession
```

```
   public Session createOrderSession(@OrderConnection Connection connection)
         throws JMSException {
      return connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
   }

   public void closeOrderSession(@Disposes @OrderSession Session session)
         throws JMSException {
      session.close();
   }

   @Produces @OrderMessageProducer
   public MessageProducer createOrderMessageProducer(@OrderSession Session session)
         throws JMSException {
      return session.createProducer(orderQueue);
   }

   public void closeOrderMessageProducer(@Disposes @OrderMessageProducer MessageProducer producer)
         throws JMSException {
      producer.close();
   }
}
```

Dans cet exemple, vous avec juste à injecter `MessageProducer`, `Connection` ou `QueueSession` qui ont été préparé :

```
@Inject Order order;
@Inject @OrderMessageProducer MessageProducer producer;
@Inject @OrderSession QueueSession orderSession;

public void sendMessage() {
   MapMessage msg = orderSession.createMapMessage();
   msg.setLong("orderId", order.getId());
   ...
   producer.send(msg);
}
```

Le cycle de vie des objects JMS injectés est complètement contrôlé par le conteneur.

15.6. Paquetage et déploiement

CDI ne définit pas d'archive de déploiement spéciale. Vous pouvez packager les beans dans des jars, EJB jars ou wars—dans n'importe quel emplacement de déploiement dans le classpath de l'application. Toutefois, l'archive doit être une "bean archive". Cela signifie que chaque archive qui contient des beans *doit* inclure un fichier nommé `beans.xml` dans le répertoire `META-INF` du classpath ou le répertoire `WEB-INF` de la racine web (pour les archives de type war). Le fichier peut être vide. Les beans déployés dans des archives qui n'ont pas de fichier `beans.xml` ne seront pas disponibles pour être utilisé dans l'application.

Dans un conteneur EJB intégrable, les beans peuvent être déployés dans n'importe quel emplacement où les EJBs peuvent être déployés. De nouveau, chaque emplacement doit contenir un fichier `beans.xml`.

Extensions portables

CDI est conçu pour être une plateforme pour des frameworks, des extensions et l'intégration avec d'autres technologies. Pour cela, CDI expose une série de SPIs pour l'usage des développeurs d'extensions portables pour CDI. Par exemple, les sortes d'extensions suivantes furent envisagées par les concepteurs de CDI :

- intégration avec des moteurs de gestion de processus métiers (BPM),
- intégration avec des frameworks tiers tels que Spring, Seam, SWT ou Wicket, et
- une nouvelle technologie basée sur le modèle de programmation de CDI.

Plus officiellement, d'après les spécifications :

> Une extension portable peut s'intégrer avec le conteneur en :
>
> - Fournissant ses propres beans, intercepteurs et décorateurs au conteneur
> - Injectant des dépendances dans ses propres objets en utilisant le service d'injection de dépendances
> - Fournissant une implémentation de contexte pour une portée personnalisée
> - Augmentant ou surchargeant les méta-données basées sur les annotations avec des méta-données d'autres sources

16.1. Créer une `Extension`

La première étape de la création d'une extension portable est d'écrire une classe qui implémente `Extension`. Cette interface marqueur ne doit pas définir de méthode, mais elle doit de répondre aux contraintes l'architecture de service provider de Java SE.

```
class MyExtension implements Extension { ... }
```

Ensuite, nous devons enregistrer notre extension comme un service provider en créant le fichier nommé `META-INF/services/javax.enterprise.inject.spi.Extension`, qui contient le nom de notre classe d'extension :

```
org.mydomain.extension.MyExtension
```

Une extension n'est pas un bean, exactement, car il est instancié par le conteneur pendant la procédure d'initialisation, avant que n'importe quel bean ou contexte existe. Cependant, il peut être injecté dans d'autres beans une fois la procédure d'initialisation terminée.

```
@Inject
MyBean(MyExtension myExtension) {
   myExtension.doSomething();
```

```
}
```

Et, comme les beans, les extensions peuvent avoir des méthodes d'observation. Habituellement, les méthodes d'observation observe les *évènement du cycle de vie du conteneur*.

16.2. Evènements du cycle de vie du conteneur

Pendant la procédure d'initialisation, le conteneur lève une série d'évènement, incluant :

- `BeforeBeanDiscovery`
- `ProcessAnnotatedType`
- `ProcessInjectionTarget` et `ProcessProducer`
- `ProcessBean` et `ProcessObserverMethod`
- `AfterBeanDiscovery`
- `AfterDeploymentValidation`

Les extensions peuvent observer ces évènements :

```
class MyExtension implements Extension {

   void beforeBeanDiscovery(@Observes BeforeBeanDiscovery bbd) {
      Logger.global.debug("beginning the scanning process");
   }

   <T
> void processAnnotatedType(@Observes ProcessAnnotatedType<T
> pat) {
      Logger.global.debug("scanning type: " + pat.getAnnotatedType().getJavaClass().getName());
   }

   void afterBeanDiscovery(@Observes AfterBeanDiscovery abd) {
      Logger.global.debug("finished the scanning process");
   }

}
```

En fait, l'extension peut faire un peu plus que juste observer. L'extension peut modifier le méta-modèle du conteneur et plus. Voici un exemple vraiment simple :

```
class MyExtension implements Extension {

   <T
> void processAnnotatedType(@Observes ProcessAnnotatedType<T
> pat) {
      //tell the container to ignore the type if it is annotated @Ignore
      if ( pat.getAnnotatedType().isAnnotionPresent(Ignore.class) ) pat.veto();
   }
```

```
}
```

La méthode d'observation peut injecter un `BeanManager`

```
<T
> void processAnnotatedType(@Observes ProcessAnnotatedType<T
> pat, BeanManager beanManager) { ... }
```

16.3. L'object `BeanManager`

Le centre nerveux pour étendre CDI est l'objet `BeanManager`. L'interface `BeanManager` nous permet d'obtenir des beans, intercepteurs, décorateurs, observateurs et contextes programmatiquement.

```
public interface BeanManager {
   public Object getReference(Bean<?> bean, Type beanType, CreationalContext<?> ctx);
   public Object getInjectableReference(InjectionPoint ij, CreationalContext<?> ctx);
   public <T
> CreationalContext<T
> createCreationalContext(Contextual<T
> contextual);
   public Set<Bean<?>
> getBeans(Type beanType, Annotation... qualifiers);
   public Set<Bean<?>
> getBeans(String name);
   public Bean<?> getPassivationCapableBean(String id);
   public <X
> Bean<? extends X
> resolve(Set<Bean<? extends X
>
> beans);
   public void validate(InjectionPoint injectionPoint);
   public void fireEvent(Object event, Annotation... qualifiers);
   public <T
> Set<ObserverMethod<? super T
>
> resolveObserverMethods(T event, Annotation... qualifiers);
   public List<Decorator<?>
> resolveDecorators(Set<Type
> types, Annotation... qualifiers);
   public List<Interceptor<?>
> resolveInterceptors(InterceptionType type, Annotation... interceptorBindings);
   public boolean isScope(Class<? extends Annotation
> annotationType);
   public boolean isNormalScope(Class<? extends Annotation
> annotationType);
   public boolean isPassivatingScope(Class<? extends Annotation
> annotationType);
   public boolean isQualifier(Class<? extends Annotation
> annotationType);
   public boolean isInterceptorBinding(Class<? extends Annotation
> annotationType);
   public boolean isStereotype(Class<? extends Annotation
> annotationType);
```

```
   public Set<Annotation
> getInterceptorBindingDefinition(Class<? extends Annotation
> bindingType);
   public Set<Annotation
> getStereotypeDefinition(Class<? extends Annotation
> stereotype);
   public Context getContext(Class<? extends Annotation
> scopeType);
   public ELResolver getELResolver();
   public ExpressionFactory wrapExpressionFactory(ExpressionFactory expressionFactory);
   public <T
> AnnotatedType<T
> createAnnotatedType(Class<T
> type);
   public <T
> InjectionTarget<T
> createInjectionTarget(AnnotatedType<T
> type);
}
```

N'importe quel bean ou autre composant Java EE qui prend en charge l'injection peut obtenir une instance de `BeanManager` par injection :

```
@Inject BeanManager beanManager;
```

Les composants Java EE peuvent obtenir une instance de `BeanManager` depuis JNDI en recherchant le nom `java:comp/BeanManager`. N'importe quelle opération de `BeanManager` peut être appelée à n'importe quel moment pendant l'exécution de l'application.

Étudions certaines des interfaces exposées par le `BeanManager`.

16.4. L'interface `InjectionTarget`

La première chose qu'un développeur de framework va regarder dans le SPI d'extension portable est une façon d'injecter des beans CDI dans des objets qui ne sont pas sous le contrôle de CDI. L'interface `InjectionTarget` peut faire ça vraiment simplement.

Note

Nous recommandons à ces frameworks de laisser à CDI le travail d'instancier les objets contrôlés par le framework. De cette façon, les objets contrôles par le framework peuvent prendre avantage de l'injection de constructeur. Cependant, si le framework nécessite l'utilisation d'un constructeur avec une signature spéciale, le framework devra instancier l'objet lui-mêmes, et donc seul l'injection de méthode et d'attribut sera supportée.

```
//get the BeanManager from JNDI
BeanManager beanManager = (BeanManager) new InitialContext().lookup("java:comp/BeanManager");

//CDI uses an AnnotatedType object to read the annotations of a class
AnnotatedType<SomeFrameworkComponent
```

```
> type = beanManager.createAnnotatedType(SomeFrameworkComponent.class);

//The extension uses an InjectionTarget to delegate instantiation, dependency injection
//and lifecycle callbacks to the CDI container
InjectionTarget<SomeFrameworkComponent
> it = beanManager.createInjectionTarget(type);

//each instance needs its own CDI CreationalContext
CreationalContext ctx = beanManager.createCreationalContext(null);

//instantiate the framework component and inject its dependencies
SomeFrameworkComponent instance = it.produce(ctx);  //call the constructor
it.inject(instance, ctx);  //call initializer methods and perform field injection
it.postConstruct(instance);  //call the @PostConstruct method

...

//destroy the framework component instance and clean up dependent objects
it.preDestroy(instance);  //call the @PreDestroy method
it.dispose(instance);  //it is now safe to discard the instance
ctx.release();  //clean up dependent objects
```

16.5. L'interface Bean

Les instances de l'interface `Bean` représentent les beans. Il y a une instance de `Bean` enregistrée auprès de l'objet `BeanManager` pour chaque bean dans l'application. Il y a même des objets `Bean` représentant les intercepteurs, décorateurs et méthodes de production.

The `Bean` interface exposes all the interesting things we discussed in *Section 2.1, « Anatomie d'un bean »*.

```
public interface Bean<T
> extends Contextual<T
> {
   public Set<Type
> getTypes();
   public Set<Annotation
> getQualifiers();
   public Class<? extends Annotation
> getScope();
   public String getName();
   public Set<Class<? extends Annotation
>
> getStereotypes();
   public Class<?> getBeanClass();
   public boolean isAlternative();
   public boolean isNullable();
   public Set<InjectionPoint
> getInjectionPoints();
}
```

Il y a une façon simple pour trouver quels beans existent dans l'application :

```
Set<Bean<?>
```

```
> allBeans = beanManager.getBeans(Obect.class, new AnnotationLiteral<Any
>() {});
```

Il est possible d'étendre l'interface `Bean` pour permettre aux extensions portables de fournir la prise en charge de nouveaux types de beans, derrière ceux définis par la spécification CDI. Par exemple, nous pouvons utiliser l'interface `Bean` pour permettre aux objets managés par un autre framework d'être injectés dans des beans.

16.6. Enregistrer un `Bean`

Le type le plus commun d'extensions CDI portables enregistre un bean (ou des beans) dans le conteneur.

Dans cet exemple, nous faisons une classe framework, `SecurityManager` disponible à l'injection. Pour faire des choses un peu plus intéressantes, nous allons déléguer à `InjectionTarget` du conteneur l'instanciation et l'injection de l'instance `SecurityManager`.

```
public class SecurityManagerExtension implements Extension {

    void afterBeanDiscovery(@Observes AfterBeanDiscovery abd, BeanManager bm) {

        //use this to read annotations of the class
        AnnotatedType<SecurityManager
> at = bm.createAnnotatedType(SecurityManager.class);

        //use this to instantiate the class and inject dependencies
        final InjectionTarget<SecurityManager
> it = bm.createInjectionTarget(at);

        abd.addBean( new Bean<SecurityManager
>() {

            @Override
            public Class<?> getBeanClass() {
                return SecurityManager.class;
            }

            @Override
            public Set<InjectionPoint
> getInjectionPoints() {
                return it.getInjectionPoints();
            }

            @Override
            public String getName() {
                return "securityManager";
            }

            @Override
            public Set<Annotation
> getQualifiers() {
                Set<Annotation
> qualifiers = new HashSet<Annotation
>();
                qualifiers.add( new AnnotationLiteral<Default
>() {} );
                qualifiers.add( new AnnotationLiteral<Any
>() {} );
```

```
                return qualifiers;
            }

            @Override
            public Class<? extends Annotation
> getScope() {
                return SessionScoped.class;
            }

            @Override
            public Set<Class<? extends Annotation
>
> getStereotypes() {
                return Collections.emptySet();
            }

            @Override
            public Set<Type
> getTypes() {
                Set<Type
> types = new HashSet<Type
>();
                types.add(SecurityManager.class);
                types.add(Object.class);
                return types;
            }

            @Override
            public boolean isAlternative() {
                return false;
            }

            @Override
            public boolean isNullable() {
                return false;
            }

            @Override
            public SecurityManager create(CreationalContext<SecurityManager
> ctx) {
                SecurityManager instance = it.produce(ctx);
                it.inject(instance, ctx);
                it.postConstruct(instance);
                return instance;
            }

            @Override
            public void destroy(SecurityManager instance,
                                CreationalContext<SecurityManager
> ctx) {
                it.preDestroy(instance);
                it.dispose(instance);
                ctx.release();
            }

        } );
    }
```

```
}
```

Mais une extension portable peut aussi bricoler les beans qui sont découverts automatiquement par le conteneur.

16.7. Wrapper un `AnnotatedType`

L'une des choses les plus intéressantes qu'une extension peut faire est de traiter les annotations d'une classe bean *avant* que le conteneur construise son méta-modèle.

Commençons avec un exemple d'un extension qui fourni la prise en charge de l'utilisation de `@Named` au niveau package. Le nom du niveau package est utilisé pour qualifier les noms EL pour tous les beans définis par ce package. L'extension portable utilise l'évènement `ProcessAnnotatedType` pour wrapper l'objet `AnnotatedType` et surcharger la `value()` de l'annotation `@Named`.

```
public class QualifiedNameExtension implements Extension {

    <X
> void processAnnotatedType(@Observes ProcessAnnotatedType<X
> pat) {

        //wrap this to override the annotations of the class
        final AnnotatedType<X
> at = pat.getAnnotatedType();

        AnnotatedType<X
> wrapped = new AnnotatedType<X
>() {

            @Override
            public Set<AnnotatedConstructor<X
>
> getConstructors() {
                return at.getConstructors();
            }

            @Override
            public Set<AnnotatedField<? super X
>
> getFields() {
                return at.getFields();
            }

            @Override
            public Class<X
> getJavaClass() {
                return at.getJavaClass();
            }

            @Override
            public Set<AnnotatedMethod<? super X
>
> getMethods() {
                return at.getMethods();
            }

            @Override
```

```
            public <T extends Annotation
> T getAnnotation(final Class<T
> annType) {
                if ( Named.class.equals(annType) ) {
                    class NamedLiteral
                            extends AnnotationLiteral<Named
>
                            implements Named {
                        @Override
                        public String value() {
                            Package pkg = at.getClass().getPackage();
                            String unqualifiedName = at.getAnnotation(Named.class).value();
                            final String qualifiedName;
                            if ( pkg.isAnnotationPresent(Named.class) ) {
                                qualifiedName = pkg.getAnnotation(Named.class).value()
                                      + '.' + unqualifiedName;
                            }
                            else {
                                qualifiedName = unqualifiedName;
                            }
                            return qualifiedName;
                        }
                    }
                    return (T) new NamedLiteral();
                }
                else {
                    return at.getAnnotation(annType);
                }
            }

            @Override
            public Set<Annotation
> getAnnotations() {
                return at.getAnnotations();
            }

            @Override
            public Type getBaseType() {
                return at.getBaseType();
            }

            @Override
            public Set<Type
> getTypeClosure() {
                return at.getTypeClosure();
            }

            @Override
            public boolean isAnnotationPresent(Class<? extends Annotation
> annType) {
                return at.isAnnotationPresent(annType);
            }

        };

        pat.setAnnotatedType(wrapped);
    }
```

```
}
```

Voici un second exemple, qui ajoute l'annotation `@Alternative` à toutes les classes qui implémentent une certaine interface `Service`.

```
class ServiceAlternativeExtension implements Extension {

   <T
> void processAnnotatedType(@Observes ProcessAnnotatedType<T
> pat) {

      final AnnotatedType<T
> type = pat.getAnnotatedType();

      if ( Service.class.isAssignableFrom( type.getJavaClass() ) ) {

         //if the class implements Service, make it an @Alternative
         AnnotatedType<T
> wrapped = new AnnotatedType<T
>() {

            @Override
            public boolean isAnnotationPresent(Class<? extends Annotation
> annotationType) {
               return annotationType.equals(Alternative.class) ?
                  true : type.isAnnotationPresent(annotationType);
            }

            //remaining methods of AnnotatedType
            ...
         }

         pat.setAnnotatedType(wrapped);
      }
   }

}
```

Le `AnnotatedType` n'est pas la seule chose qui peut être wrappée par une extension.

16.8. Wrapper une `InjectionTarget`

L'interface `InjectionTarget` expose les opérations pour produire et détruire une instance d'un composant, injecter ses dépendances et appeler ses callbacks de cycle de vie. Une extension portable peut wrapper le `InjectionTarget` pour n'importe quel composant Java EE qui supporte l'injection, lui permettant d'intercepter n'importe quelle de ces opérations quand elles sont appelées par le conteneur.

Voici une extension CDI portable qui lit les valeurs depuis des fichiers de propriétés et configure les attributs des composants Java EE, incluant les servlets, EJBs, beans managés, intercepteurs et plus. Dans cet exemple, les propriétés pour une classe comme `org.mydomain.blog.Blogger` proviendront d'une ressource nommée `org/mydomain/blog/Blogger.properties`, et le nom d'une propriété doit correspondre au nom de l'attribut qui doit être configuré. `Blogger.properties` peut donc contenir :

```
firstName=Gavin
lastName=King
```

L'extension portable fonctionne en wrappant le `InjectionTarget` du conteneur et configure les valeurs des attributs depuis la méthode `inject()`.

```
public class ConfigExtension implements Extension {

        <X
> void processInjectionTarget(@Observes ProcessInjectionTarget<X
> pit) {

                //wrap this to intercept the component lifecycle
            final InjectionTarget<X
> it = pit.getInjectionTarget();

        final Map<Field, Object
> configuredValues = new HashMap<Field, Object
>();

        //use this to read annotations of the class and its members
        AnnotatedType<X
> at = pit.getAnnotatedType();

        //read the properties file
        String propsFileName = at.getClass().getSimpleName() + ".properties";
        InputStream stream = at.getJavaClass().getResourceAsStream(propsFileName);
        if (stream!=null) {

            try {
                Properties props = new Properties();
                props.load(stream);
                for (Map.Entry<Object, Object
> property : props.entrySet()) {
                    String fieldName = property.getKey().toString();
                    Object value = property.getValue();
                    try {
                        Field field = at.getJavaClass().getField(fieldName);
                        field.setAccessible(true);
                        if ( field.getType().isAssignableFrom( value.getClass() ) ) {
                            configuredValues.put(field, value);
                        }
                        else {
                            //TODO: do type conversion automatically
                            pit.addDefinitionError( new InjectionException(
                                    "field is not of type String: " + field ) );
                        }
                    }
                    catch (NoSuchFieldException nsfe) {
                        pit.addDefinitionError(nsfe);
                    }
                    finally {
                        stream.close();
                    }
                }
```

```
            }
            catch (IOException ioe) {
                pit.addDefinitionError(ioe);
            }
        }

        InjectionTarget<X
> wrapped = new InjectionTarget<X
>() {

            @Override
            public void inject(X instance, CreationalContext<X
> ctx) {
                it.inject(instance, ctx);

                //set the values onto the new instance of the component
                for (Map.Entry<Field, Object
> configuredValue: configuredValues.entrySet()) {
                    try {
                        configuredValue.getKey().set(instance, configuredValue.getValue());
                    }
                    catch (Exception e) {
                        throw new InjectionException(e);
                    }
                }
            }

            @Override
            public void postConstruct(X instance) {
                it.postConstruct(instance);
            }

            @Override
            public void preDestroy(X instance) {
                it.dispose(instance);
            }

            @Override
            public void dispose(X instance) {
                it.dispose(instance);
            }

            @Override
            public Set<InjectionPoint
> getInjectionPoints() {
                return it.getInjectionPoints();
            }

            @Override
            public X produce(CreationalContext<X
> ctx) {
                return it.produce(ctx);
            }

        };

        pit.setInjectionTarget(wrapped);

    }
```

```
}
```

Il y a plein d'autres choses à propos du SPI des extensions portables que ce que nous avons discuté ici. Consultez la spécification CDI ou la Javadoc pour plus d'information. Pour l'instant, nous allons juste mentionner un point d'extension supplémentaire.

16.9. L'interface `Context`

L'interface `Context` prend en charge l'ajout de nouvelles portées à CDI, ou d'extensions des portées intégrées à de nouveaux environnements.

```
public interface Context {
   public Class<? extends Annotation
> getScope();
   public <T
> T get(Contextual<T
> contextual, CreationalContext<T
> creationalContext);
   public <T
> T get(Contextual<T
> contextual);
   boolean isActive();
}
```

Par exemple, nous pourrions implémenter `Context` pour ajouter une portée de type processus métier à CDI, ou pour ajouter le support d'une portée conversation à une application qui utilise Wicket.

Etapes suivantes

Comme CDI est nouveau, il n'y a pas encore beaucoup d'informations disponibles en ligne. Cela va changer avec le temps. Peu importe, la spécification CDI reste la référence sur CDI. La spécification fait moins de 100 pages et est très lisible (ne vous inquiétez pas, ce n'est pas comme le manuel de votre lecteur Blu-ray). Bien sûr, elle couvre de nombreux détails que nous avons sauté ici. La spécification est disponible sur la *page de la JSR-299* [http://jcp.org/en/jsr/detail?id=299] sur le site du JCP.

L'implémentation de référence de CDI, Weld, est développée sur le *projet Seam* [http://seamframework.org/Weld]. L'équipe de développement de Weld et le responsable de la spécification CDI bloguent sur *in.relation.to* [http://in.relation.to]. A l'origine, ce guide a été basé sur une série d'articles de blog publiés là-bas lorsque la spécification a été mise au point. C'est probablement la meilleure source d'information sur l'avenir de la CDI, Weld et Seam.

Nous vous encourageons à suivre la liste de diffusion *weld-dev* [https://lists.jboss.org/mailman/listinfo/weld-dev] et à vous impliquer dans le *développement* [http://seamframework.org/Weld/Development]. Si vous êtes en train de lire ce guide, vous avez probablement quelque chose à offrir.

Partie V. Guide de référence Weld

Weld est l'implémentation de référence de la JSR-299, et est utilisé par JBoss AS et GlassFish pour fournir les services CDI aux applications Java Enterprise Edition (Java EE). Weld va également au-delà des environnements et des API définies par la spécification JSR-299 en fournissant une prise en charge d'un certain nombre d'autres environnements (comme les conteneurs de servlets tel que Tomcat, ou Java SE).

Vous pouvez également consulter le projet des extensions Weld qui fournit des extensions portables à CDI (telles que l'injection de journal et des annotations supplémentaires pour la création de beans), et Seam, qui fournit l'intégration avec d'autres couches de vue (tels que GWT et Wicket), d'autres frameworks (comme Drools) ainsi que des extensions à l'écosystème (tel que la prise en charge de la sécurité).

If you want to get started quickly using Weld (and, in turn, CDI) with JBoss AS, GlassFish or Tomcat and experiment with one of the examples, take a look at *Chapitre 6, Démarrer avec Weld*. Otherwise read on for a exhaustive discussion of using Weld in all the environments and application servers it supports and the Weld extensions.

Les serveurs d'application et les environnements pris en charge par Weld

18.1. Utiliser Weld avec JBoss AS

Si vous utilisez JBoss AS 6.0, aucune configuration supplémentaire n'est nécessaire pour utiliser Weld (ou CDI dans notre cas). Tout ce que vous devez faire est de transformer votre application en archive de bean en ajoutant `META-INF/beans.xml` au classpath ou `WEB-INF/beans.xml` à la racine web !

Note

En outre, la Servlet de Weld prend en charge JBoss EAP 5.1, pour faire l'usage de la variante `jboss5` de la Servlet de Weld.

18.2. GlassFish

Weld est aussi intégré dans GlassFish à partir de V3. Depuis que GlassFish V3 est l'implémentation de référence de Java EE 6, il doit prendre en charge toutes les fonctionnalités de CDI. Quelle meilleure façon pour GlassFish de prendre en charge ces fonctionnalités que d'utiliser Weld, l'implémentation de référence de la JSR-299 ? Packagez juste votre application CDI et déployez.

18.3. Les conteneurs de Servlet (comme Tomcat ou Jetty)

Même si la JSR-299 n'impose pas de prendre en charger les environnements servlet, Weld peut être utilisé dans un conteneur de servlet, comme Tomcat 6.0 ou Jetty 6.1.

Note

There is a major limitation to using a servlet container. Weld doesn't support deploying session beans, injection using `@EJB` or`@PersistenceContext`, or using transactional events in servlet containers. For enterprise features such as these, you should really be looking at a Java EE application server.

Weld peut être utilisé comme une librairie dans une application web qui est déployée dans un conteneur Servlet. Vous devrez placer `weld-servlet.jar` dans le répertoire `WEB-INF/lib` relatif à la racine web. `weld-servlet.jar` est un "uber-jar", ce qui signifie qu'il contient tous les bouts de Weld et CDI requis pour l'exécution dans un conteneur de Servlet, pour votre commodité. Sinon, vous pouvez utiliser ses jars de composant. Une liste des dépendances transitives peut être trouvée dans le fichier `META-INF/DEPENDENCIES.txt` inclus dans l'artifact `weld-servlet.jar`.

Vous aurez aussi besoin de spécifier explicitement le servlet listener (utiliser pour démarrer Weld, et contrôler ses interactions avec les requêtes) dans `WEB-INF/web.xml` de la racine web :

```
<listener>
   <listener-class
>org.jboss.weld.environment.servlet.Listener</listener-class>
</listener
>
```

18.3.1. Tomcat

Tomcat a JNDI en lecture seule, c'est pourquoi Weld ne peut pas lier automatiquement l'extension SPI de BeanManager. Pour lier le BeanManager dans JNDI, vous devrez peupler `META-INF/context.xml` de la racine web avec le contenu suivant :

```
<Context>
   <Resource name="BeanManager"
      auth="Container"
      type="javax.enterprise.inject.spi.BeanManager"
      factory="org.jboss.weld.resources.ManagerObjectFactory"/>
</Context
>
```

and make it available to your deployment by adding this to the bottom of`web.xml`:

```
<resource-env-ref>
   <resource-env-ref-name
>BeanManager</resource-env-ref-name>
   <resource-env-ref-type>
      javax.enterprise.inject.spi.BeanManager
   </resource-env-ref-type>
</resource-env-ref
>
```

Tomcat only allows you to bind entries to`java:comp/env`, so the BeanManager will be available at `java:comp/env/BeanManager`

Weld prend aussi en charge l'injection de Servlet dans Tomcat 6.

18.3.2. Jetty

Comme Tomcat, Jetty a JNDI en lecture seule, c'est pourquoi Weld ne peut pas lier automatiquement le BeanManager. Pour lier le BeanManager à JNDI dans Jetty 6, vous devrez peupler `WEB-INF/jetty-env.xml` avec le contenu suivant :

```
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
   "http://jetty.mortbay.org/configure.dtd">
<Configure id="webAppCtx" class="org.mortbay.jetty.webapp.WebAppContext">
   <New id="BeanManager" class="org.mortbay.jetty.plus.naming.Resource">
      <Arg
><Ref id="webAppCtx"/></Arg
>
```

```
      <Arg
>BeanManager</Arg>
      <Arg>
         <New class="javax.naming.Reference">
            <Arg
>javax.enterprise.inject.spi.BeanManager</Arg
>
            <Arg
>org.jboss.weld.resources.ManagerObjectFactory</Arg>
            <Arg/>
         </New>
      </Arg>
   </New>
</Configure
>
```

Jetty 7 a bougé dans la fondation Eclipse ; si vous utilisez Jetty 7 mettez le contenu suivant dans votre `WEB-INF/jetty-env.xml` :

```
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
   "http://www.eclipse.org/jetty/configure.dtd">

<Configure id="webAppCtx" class="org.eclipse.jetty.webapp.WebAppContext">
    <New id="BeanManager" class="org.eclipse.jetty.plus.jndi.Resource">
        <Arg
> <Ref id="webAppCtx"/> </Arg>
        <Arg
>BeanManager</Arg>
        <Arg>
            <New class="javax.naming.Reference">
                <Arg
>javax.enterprise.inject.spi.BeanManager</Arg>
                <Arg
>org.jboss.weld.resources.ManagerObjectFactory</Arg>
                <Arg/>
            </New>
        </Arg>
    </New>
</Configure
>
```

Tout comme avec Tomcat, vous devez le rendre disponible à votre déploiement en ajoutant ceci à la fin de `web.xml` :

```
<resource-env-ref>
   <resource-env-ref-name
>BeanManager</resource-env-ref-name>
   <resource-env-ref-type>
      javax.enterprise.inject.spi.BeanManager
   </resource-env-ref-type>
</resource-env-ref
>
```

Notice that Jetty doesn't not have built-in support for an `javax.naming.spi.ObjectFactory` like Tomcat, so it's necessary to manually create the `javax.naming.Reference` to wrap around it.

Jetty only allows you to bind entries to`java:comp/env`, so the BeanManager will be available at `java:comp/env/BeanManager`

Weld prend aussi en charge l'injection de Servlet dans Jetty 6. Pour l'activer, ajoutez le fichier `WEB-INF/jetty-web.xml` avec le contenu suivant dans votre war :

```
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
   "http://jetty.mortbay.org/configure.dtd">
<Configure id="webAppCtx" class="org.mortbay.jetty.webapp.WebAppContext">
   <Call class="org.jboss.weld.environment.jetty.WeldServletHandler" name="process">
      <Arg
><Ref id="webAppCtx"/></Arg>
   </Call>
</Configure
>
```

18.4. Java SE

De plus, pour améliorer l'intégration de la couche Java Entreprise, la spécification "Contextes et Injection de Dépendances pour la plateforme Java EE" définit aussi un framework d'injection de dépendance fortement typé et avec état, à l'état de l'art, qui peut s'avérer utile dans un large éventail d'application. Pour aider les développeurs à prendre avantage de ça, Weld fournit un moyen simple pour être exécuté dans l'environnement Java Standard Edition (SE) indépendamment de toutes APIs Java EE.

Quand exécutées dans l'environnement SE, les fonctionnalités suivantes de Weld sont disponibles :

- Managed beans avec les callbacks de cycle de vie `@PostConstruct` et `@PreDestroy`
- L'injection de dépendances avec les qualifiants et les alternatives
- les portées `@Application`, `@Dependent` et `@Singleton`
- Les intercepteurs et les décorateurs
- Les stéréotypes
- Les évènements
- Le support des extensions portables

Les beans EJB ne sont pas supportés.

18.4.1. Module CDI SE

Weld fournit une extension qui démarrera un manager de bean CDI dans Java SE, en enregistrant automatiquement tous les beans simples trouvés dans le classpath. Les paramètres de la ligne de commande peuvent être injectés en utilisant l'une des choses suivantes :

```
@Inject @Parameters List<String
> params;
```

```
@Inject @Parameters String[] paramsArray;
```

La seconde forme est utile pour la compatibilité avec des classes existantes.

Note

Les paramètres de ligne de commande ne peuvent pas être disponible pour l'injection tant que l'évènement `ContainerInitialized` n'est pas levé. Si vous avez besoin d'accéder aux paramètres pendant l'initialisation, vous pouvez le faire via la méthode `public static String[] getParameters()` de `StartMain`.

Voici un exemple d'une simple application CDI SE :

```
@Singleton
public class HelloWorld
{
   public void printHello(@Observes ContainerInitialized event, @Parameters List<String
> parameters) {
        System.out.println("Hello " + parameters.get(0));
   }
}
```

18.4.2. Amorçage de CDI SE

Les applications CDI SE peuvent être ammorçées de façons suivantes.

18.4.2.1. L'évènement ContainerInitialized

Grâce à la puissance du modèle d'évènement fortement typé de CDI, les développeurs d'application n'ont pas besoin un quelconque code d'amorçage. Le module Weld SE vient avec une méthode main intégrée qui amorcera CDI pour vous et lèvera ensuite un évènement `ContainerInitialized`. Le point d'entré pour le code de votre application sera donc un simple bean qui observe l'évènement `ContainerInitialized`, comme dans l'exemple précédent.

Dans ce cas, votre application sera démarrée en appelant la méthode main fournit de cette façon :

```
java org.jboss.weld.environment.se.StartMain <args
>
```

18.4.2.2. API d'amorçage par programmation

Pour une flexibilité accrue, CDI SE vient aussi avec un API d'amorçage qui peut être appelée depuis votre application afin d'initialiser CDI et obtenir des références des beans de votre application et des évènements. L'API consiste en deux classes : `Weld` et `WeldContainer`.

```
public class Weld
{
```

```
   /** Boots Weld and creates and returns a WeldContainer instance, through which
    * beans and events can be accesed. */
   public WeldContainer initialize() {...}

   /** Convenience method for shutting down the container. */
   public void shutdown() {...}

}
```

```
public class WeldContainer
{

   /** Provides access to all beans within the application. */
   public Instance<Object
> instance() {...}

   /** Provides access to all events within the application. */
   public Event<Object
> event() {...}

   /** Provides direct access to the BeanManager. */
   public BeanManager getBeanManager() {...}

}
```

Here's an example application main method which uses this API to initialize a bean of type`MyApplicationBean`.

```
public static void main(String[] args) {
   Weld weld = new Weld();
   WeldContainer container = weld.initialize();
   container.select(MyApplicationBean.class).get();
   weld.shutdown();
}
```

Sinon, l'application peut être démarrée en levant un évènement personnalisé qui pourra alors être observé par un autre bean simple. L'exemple suivant lève `MyEvent` au démarrage.

```
public static void main(String[] args) {
   Weld weld = new Weld();
   WeldContainer container = weld.initialize();
   container.event().select(MyEvent.class).fire( new MyEvent() );
   weld.shutdown();
}
```

18.4.3. Contexte de threads

In contrast to Java EE applications, Java SE applications place no restrictions on developers regarding the creation and usage of threads. Therefore Weld SE provides a custom scope annotation,`@ThreadScoped`, and corresponding context implementation which can be used to bind bean instances to the current thread. It is intended

to be used in scenarios where you might otherwise use`ThreadLocal`, and does in fact use `ThreadLocal` under the hood.

Pour utiliser l'annotation @ThreadScoped vous devez activer le `RunnableDecorator` qui 'écoute' toutes les exécutions de `Runnable.run()` et les décore en configurant le contexte de thread préalablement, lié au thread courant, et détruit le contexte après.

```
<beans>
  <decorators>
     <class
>org.jboss.weld.environment.se.threading.RunnableDecorator</class>
  </decorator>
</beans
>
```

Note

Il n'est pas nécessaire d'utiliser @ThreadScoped dans toutes les applications multi-threadées. Le contexte de thread n'est prévu comme un remplaçant pour définir votre propre contexte spécifique d'application. Il est généralement utile dans les situations où vous voulez utiliser directement ThreadLocal, qui sont typiquement rares.

18.4.4. Réglage du Classpath

Weld SE comes packaged as a 'shaded' jar which includes the CDI API, Weld Core and all dependant classes bundled into a single jar. Therefore the only Weld jar you need on the classpath, in addition to your application's classes and dependant jars, is the Weld SE jar. If you are working with a pure Java SE application you launch using`java`, this may be simpler for you.

Si vous préférez travailler avec les dépendances individuelles, alors vous pouvez utiliser le jar `weld-core` qui contient juste les classes Weld SE. Bien sur, dans ce mode, vous aurez besoin d'assembler le classpath vous-même. Ce mode est utile, par exemple, si vous souhaitez d'utiliser un moteur de log slf4j alternatif.

If you work with a dependency management solution such as Maven you can declare a dependency on`org.jboss.weld.se:weld-se-core`.

Gestion du context

19.1. Gérer les contextes de production

Weld allows you to easily manage the built in contexts by injecting them and calling lifecycle methods. Weld defines two types of context, *managed* and *unmanaged*. Managed contexts can be activated (allowing bean instances to be retrieved from the context), invalidated (scheduling bean instances for destruction) and deactivated (stopping bean instances from being retrieved, and if the context has been invalidated, causing the bean instances to destroyed). Unmanaged contexts are always active; some may offer the ability to destroy instances.

Managed contexts can either be *bound* or *unbound*. An unbound context is scoped to the thread in which it is activated (instances placed in the context in one thread are not visible in other threads), and is destroyed upon invalidation and deactivation. Bound contexts are attached to some external data store (such as the Http Session or a manually propagated map) by *associating* the data store with the context before calling activate, and dissociating the data store after calling activate.

Astuce

Weld contrôle automatiquement le contexte de cycle de vie dans beaucoup de scénarios comme les requêtes HTTP, l'invocation d'EJB distant, et l'invocation MDB. Beaucoup d'extensions CDI offrent le contexte de cycle de vie pour d'autres environnements, il vaut mieux vérifier qu'il n'existe pas une extension convenable avant de décider de gérer vous même le contexte.

Weld provides a number of built in contexts, which are shown in *Tableau 19.1, « Les contextes disponibles dans Weld »*.

Tableau 19.1. Les contextes disponibles dans Weld

Portée (Scope)	Qualifiants (Qualifiers)	Contexte (Context)	Notes (Notes)
`@Dependent`	`@Default`	`DependentContext`	Le contexte dépendant est non lié et non managé.
`@RequestScoped`	`@Unbound`	`RequestContext`	Un contexte de requête non lié, utile pour tester
`@RequestScoped`	`@Bound`	`RequestContext`	Un contexte de requête lié à une map propagée manuellement, utile pour tester ou pour les environnements autre que Servlet
	`@Default`	`BoundRequestContext`	
`@RequestScoped`	`@Http`	`RequestContext`	Un contexte de requête lié à une requête Servlet, utilisé pour n'importe quel contexte de requête basé sur Servlet
	`@Default`	`HttpRequestContext`	
`@RequestScoped`	`@Ejb`	`RequestContext`	Un contexte de requête lié au contexte d'invocation d'un intercepteur, utilisé par l'invocation d'EJB
	`@Default`	`EjbRequestContext`	

Portée (Scope)	Qualifiants (Qualifiers)	Contexte (Context)	Notes (Notes)
			en dehors des requêtes Servlet
`@ConversationScoped`	`@Bound`	`ConversationContext`	Un contexte de conversation lié à deux maps propagées manuellement (une qui représente la requête et une qui représente la session), utilise pour tester ou les environnements autre que Servlet
	`@Default`	`BoundConversationContext`	
`@ConversationScoped`	`@Http`	`ConversationContext`	Un contexte de conversation lié à une requête Servlet, utilisé pour n'importe quel contexte de conversation basé sur Servlet
	`@Default`	`HttpConversationContext`	
`@SessionScoped`	`@Bound`	`SessionContext`	Un contexte de session lié à une map manuellement propagée, utile pour les tests ou les environnements autre que Servlet
	`@Default`	`BoundSessionContext`	
`@SessionScoped`	`@Http`	`SessionContext`	Un contexte de session lié à une requête Servlet, utilisé pour n'importe contexte de session basé sur Servlet
	`@Default`	`HttpSessionContext`	
`@ApplicationScoped`	`@Default`	`ApplicationContext`	Un contexte d'application soutenu par un singleton de portée application, il est non managé et non lié mais il offre la possibilité de détruire toutes les entrées
`@SingletonScoped`	`@Default`	`SingletonContext`	Un contexte singleton soutenu par un singleton de portée application, il est non managé et non lié mais offre la possibilité de détruire toutes les entrées

Les contextes non managés offrent peu d'intérêt dans une discussion sur la gestion des cycles de vie de contexte, donc à partir de maintenant nous allons nous concentrer sur les contextes managés (les contextes non managés jouent bien entendu un rôle vital dans le fonctionnement de votre application et de Weld !). Comme vous avez pu le voir sur la liste au-dessus, les contextes managés offrent un grand nombre d'implémentation différentes pour la même portée ; en général, chaque sorte de contexte pour une portée a la même API. Nous allons naviguer au travers d'un grand nombre de scénarios courants sur la gestion du cycle de vie ; armé de cette connaissance, et de la Javadoc, vous devriez être capable de travailler avec n'importe quelle implémentation de contexte offert par Weld.

Nous commencerons simplement avec `BoundRequestContext`, que vous pouvez utiliser pour fournir la portée de requête en dehors d'une requête Servlet ou de l'invocation d'un EJB.

```
   /* Inject the BoundRequestContext. */
   /* Alternatively, you could look this up from the BeanManager */
   @Inject BoundRequestContext requestContext;

   ...

   /* Start the request, providing a data store which will last the lifetime of the request */
   public void startRequest(Map<String, Object
> requestDataStore) {
      // Associate the store with the context and acticate the context
      requestContext.associate(requestDataStore);
      requestContext.activate();
   }

   /* End the request, providing the same data store as was used to start the request */
   public void endRequest(Map<String, Object
> requestDataStore) {
      try {
         /* Invalidate the request (all bean instances will be scheduled for destruction) */
         requestContext.invalidate();
         /* Deactivate the request, causing all bean instances to be destroyed (as the context
 is invalid) */
         requestContext.deactivate();
      } finally {
         /* Ensure that whatever happens we dissociate to prevent any memory leaks */
         requestContext.dissociate(requestDataStore);
      }
   }
```

Le contexte de session lié fonctionne de la même façon, excepté que l'invalidation et la désactivation du contexte de session provoquent à n'importe quelles conversations dans la session d'être également détruites. Le contexte de session http et le contexte de requête http fonctionnent aussi de façon similaire, et peuvent être utilisé si vous vous retrouvez à créer vous même les threads d'une requête http). Le contexte de session http offre en outre une méthode qui permet de détruire immédiatement le contexte.

Note

Weld's session contexts are "lazy" and don't require a session to actually exist until a bean instance must be written.

Le contexte de conversation offre un peu plus d'options, que nous allons étudier par ici.

```
   @Inject BoundConversationContext conversationContext;

   ...

   /* Start a transient conversation */
   /* Provide a data store which will last the lifetime of the request */
   /* and one that will last the lifetime of the session */
```

```
   public void startTransientConversation(Map<String, Object
> requestDataStore,
                                          Map<String, Object
> sessionDataStore) {
      resumeOrStartConversation(requestDataStore, sessionDataStore, null);
   }

   /* Start a transient conversation (if cid is null) or resume a non-transient */
   /* conversation. Provide a data store which will last the lifetime of the request */
   /* and one that will last the lifetime of the session */
   public void resumeOrStartConversation(Map<String, Object
> requestDataStore,
                                         Map<String, Object
> sessionDataStore,
                                         String cid) {
      /* Associate the stores with the context and acticate the context */
      * BoundRequest just wraps the two datastores */
      conversationContext.associate(new MutableBoundRequest(requestDataStore, sessionDataStore));
      // Pass the cid in
      conversationContext.activate(cid);
   }

   /* End the conversations, providing the same data store as was used to start */
   /* the request. Any transient conversations will be destroyed, any newly-promoted */
   /* conversations will be placed into the session */
   public void endOrPassivateConversation(Map<String, Object
> requestDataStore,
                                          Map<String, Object
> sessionDataStore) {
      try {
          /* Invalidate the conversation (all transient conversations will be scheduled for
 destruction) */
         conversationContext.invalidate();
        /* Deactivate the conversation, causing all transient conversations to be destroyed */
         conversationContext.deactivate();
      } finally {
        /* Ensure that whatever happens we dissociate to prevent memory leaks*/
         conversationContext.dissociate(new MutableBoundRequest(requestDataStore, sessionDataStore));
      }
   }
```

The conversation context also offers a number of properties which control the behavior of conversation expiration (after this period of inactivity the conversation will be ended and destroyed by the container), and the duration of lock timeouts (the conversation context ensures that a single thread is accessing any bean instances by locking access, if a lock can't be obtained after a certain time Weld will error rather than continue to wait for the lock). Additionally, you can alter the name of the parameter used to transfer the conversation id (by default, `cid`).

Weld also introduces the notion of a `ManagedConversation`, which extends the `Conversation` interface with the ability to lock, unlock and touch (update the last used timestamp) a conversation. Finally, all non-transient conversations in a session can be obtained from the conversation context, as can the current conversation.

Note

Weld's conversations are not assigned ids until they become non-transient.

Configuration

20.1. Excluding classes from scanning and deployment

Weld allows you to exclude classes in your archive from scanning, having container lifecycle events fired, and being deployed as beans.

In this tutorial, we'll explore this feature via an example; a more formal specification can be found in the xsd, located at *http://jboss.org/schema/weld/beans_1_1.xsd*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:weld="http://jboss.org/schema/weld/beans"
       xsi:schemaLocation="
          http://java.sun.com/xml/ns/javaee http://docs.jboss.org/cdi/beans_1_0.xsd
          http://jboss.org/schema/weld/beans http://jboss.org/schema/weld/beans_1_1.xsd">

    <weld:scan>

        <!-- Don't deploy the classes for the swing app! -->
        <weld:exclude name="com.acme.swing.**" />

        <!-- Don't include GWT support if GWT is not installed -->
        <weld:exclude name="com.acme.gwt.**">
            <weld:if-class-available name="!com.google.GWT"/>
        </weld:exclude>

        <!--
            Exclude classes which end in Blether if the system property verbosity is set to low
            i.e.
              java ... -Dverbosity=low
        -->
        <weld:exclude pattern="^(.*)Blether$">
            <weld:if-system-property name="verbosity" value="low"/>
        </weld:exclude>

       <!--
            Don't include JSF support if Wicket classes are present, and the viewlayer system
            property is not set
        -->
        <weld:exclude name="com.acme.jsf.**">
            <weld:if-class-available name="org.apahce.wicket.Wicket"/>
            <weld:if-system-property name="!viewlayer"/>
        </weld:exclude>
    </weld:scan>

</beans
>
```

Dans cet exemple, nous montrons les cas d'utilisation les plus communs pour avoir un contrôle fin sur les classes que Weld va rechercher. Le premier filtre exclut toutes les classes du package `com.acme.swing`, et dans la plupart des cas cela sera suffisant pour vos besoins.

Toutefois, il est parfois utile d'être capable d'activer le filtre en fonction de l'environnement utilisé. Dans ce cas, Weld vous permet d'activer (ou désactiver) un filtre basé sur les propriétés système ou sur la disponibilité d'une classe. Le deuxième filtre montre le cas d'utilisation de la désactivation de la recherche de certaines classes en fonction des capacités de l'environnement que vous déployez - dans ce cas nous excluons la prise en charge de GWT s'il n'est pas installé.

Note

Remarquez comment nous utilisons un caractère ! sur le `nom` de l'attribut pour inverser la condition d'activation. Vous pouvez inverser une condition d'activation de cette façon.

The third filter uses a regular expression to select the classes to match (normally we use simple name-based patterns, as they don't require us to escape the period which delimits the package hierarchy).

Note

If you specify just a system property name, Weld will activate the filter if that system property has been set (with any value). If you also specify the system property value, then Weld will only activate the filter if the system property's value matches exactly.

Le quatrième filtre montre une configuration avancée, où nous utilisons les conditions d'activation multiples pour décider quel filtre activer.

You can combine as many activation conditions as you like (*all* must be true for the filter to be activated). If you want to a filter that is active if *any* of the activation conditions are true, then you need multiple identical filters, each with different activation conditions.

Astuce

En général, la sémantique définie par les "pattern sets" d'Ant (http://ant.apache.org/manual/dirtasks.html # patternset) sont suivies.

Annexe A. Intégrer Weld dans d'autres environnements

Si vous voulez utiliser Weld dans d'autres environnement, vous devrez fournir certaines informations à Weld le SPI d'intégration. Dans cette annexe, nous allons examiner brièvement les étapes nécessaires.

Services d'Entreprise

Si vous voulez juste utiliser les managed beans, et ne pas profiter des services d'entreprise (injection de ressources EE, injection CDI dans des classes de composants EE, évènements transactionnels, prise en charge des services CDI dans les EJBs) et des déploiements non-flat, alors le support générique de servlet fournit par l'extension "Weld : Servlet" sera suffisant, et fonctionnera dans n'importe quel conteneur prenant en charge l'API Servlet.

Tous les SPIs et APIs décris ont une JavaDoc complète, qui explique clairement et en détail le contrat entre le conteneur et Weld.

A.1. Le SPI de Weld

Le SPI de Weld est localisé dans le module `weld-spi`, et packagé en tant que `weld-spi.jar`. Quelques SPIs sont optionnels, et doivent seulement être implémentés si vous avez besoin de surchager le comportement par défaut ; les autres sont requis.

Toutes les interfaces dans le SPI prennent en charge le pattern décorateur et fournissent une classe `Forwarding` localisée dans le sous package `helpers`. En outre, couramment utilisés, les classes utilitaires, et les implémentations standards sont aussi localisées dans le sous package `helpers`.

Weld prend en charge des environnements multiples. Un environnement est défini par une implémentation de l'interface `Environment`. Un nombre d'environnements standards sont intégrés, et décrits par l'énumération `Environments`. D'autres environnements ont besoin de la présence de certains services (par exemple, un conteneur de Servlet n'a pas besoin des services transaction, EJB ou JPA). Par défaut, un environnement EE est supposé, mais vous pouvez ajuster cet environnement en appelant `bootstrap.setEnvironment()`.

Weld utilise un registre de services génériquement typés pour permettre aux services d'être enregistrés. Tous les services implémentent l'interface `Service`. Le registre de service permet aux services d'être ajoutés et retrouvés.

A.1.1. Structure de déploiement

Une application est souvent composée de plusieurs modules. Par exemple, un déploiement Java EE peut contenir des modules EJB (contenant la logique métier) et des modules war (contenant l'interface utilisateur). Un conteneur peut imposer certaines règles d'*accessibilité* qui limite la visibilité des classes entre les modules. CDI permet d'appliquer ces règles à la résolution des beans et des méthodes d'observation. Comme les règles d'accessibilité changent d'un conteneur à l'autre, Weld requiert que le conteneur *décrive* la structure de déploiement, via le SPI `Deployment`.

The CDI specification discusses *Bean Deployment Archives* (BDAs)—archives which are marked as containing beans which should be deployed to the CDI container, and made available for injection and resolution. Weld reuses this description of *Bean Deployment Archives* in its deployment structure SPI. Each deployment exposes the BDAs which it contains; each BDA may also reference other which it can access. Together, the transitive closure of this graph forms the beans which are deployed in the application.

Pour décrire la structure de déploiement à Weld, le conteneur doit fournir une implémentation de `Deployment`. `Deployment.getBeanDeploymentArchives()` permet à Weld de découvrir les modules qui constituent l'application. La spécification CDI permet aussi aux beans d'être spécifiés par programmation comme faisant parti du déploiement de beans. Ces beans peuvent, ou non, être dans une BDA existante. Pour cette raison, Weld appellera `Deployment.loadBeanDeploymentArchive(Class clazz)` pour chaque bean décrit par programmation.

Comme les beans décrit par programmation peuvent provenir de BDAs supplémentaires étant ajouté au graphe, Weld explorera la structure BDA chaque fois qu'un BDA inconnu sera retourné par `Deployment.loadBeanDeploymentArchive`.

BDAs virtuelles

Dans un conteneur stricte, chaque BDA peut avoir besoin de spécifier explicitement à quelles autres BDAs elle peut accéder. Cependant, beaucoup de conteneurs permettront un mécanisme simple pour rendre les BDAs accessibles bi-directionnellement (comme un répertoire de librairies). Dans ce cas, il est permit (et raisonnable) de décrire tout ce type d'archive comme une unique, 'virtuelle' `BeanDeploymentArchive`.

Un conteneur, peut, par exemple, utiliser une structure d'accès à plat pour l'application. Dans ce cas, un unique `BeanDeploymentArchive` sera attaché au `Deployment`.

`BeanDeploymentArchive` fournit trois méthodes qui permettent à leur contenu d'être découverts par Weld—`BeanDeploymentArchive.getBeanClasses()` doit retourner toutes les classes dans le BDA, `BeanDeploymentArchive.getBeansXml()` doit retourner une structure de données représentant le descripteur de déploiement `beans.xml` pour l'archive, et `BeanDeploymentArchive.getEjbs()` doit fournir un descripteur d'EJB pour tous les EJBs dans le BDA, ou une liste vide si ce n'est pas une archive EJB.

Pour aider les intégrateurs de conteneur, Weld fournit un parseur intégré `beans.xml`. Pour parser un `beans.xml` dans la structure de données requise par `BeanDeploymentArchive`, le conteneur devrait appeler `Bootstrap.parseBeansXml(URL)`. Weld peut aussi parser plusieurs fichiers `beans.xml`, en les fusionnant pour devenir une structure de données unique. Cela peut être fait en appelant `Bootstrap.parseBeansXml(Iterable<URL>)`.

BDA X may also reference another BDA Y whose beans can be resolved by, and injected into, any bean in BDA X. These are the accessible BDAs, and every BDA that is directly accessible by BDA X should be returned. A BDA will also have BDAs which are accessible transitively, and the transitive closure of the sub-graph of BDA X describes all the beans resolvable by BDA X.

Matching the classloader structure for the deployment

En pratique, vous pouvez regarder la structure de déploiement représentée par `Deployment`, et le graphe du BDA virtuel comme exemple pour la structure du classloader pour le déploiement. Si une classe depuis BDA X peut être chargée par une dans BDA Y, elle est accessible, et par conséquent les BDAs accessibles par BDA Y doivent inclurent BDA X.

Pour spécifier les BDAs directement accessibles, le conteneur doit fournir une implémentation de `BeanDeploymentArchive.getBeanDeploymentArchives()`.

Note

Weld permet au conteneur de décrire un graphe circulaire, et convertira un graphe en arbre dans le cadre du processus de déploiement.

Certains services sont fournis pour l'ensemble du déploiement, tandis que certains sont fournis par BDA. Les services BDA sont fournis en utilisant `BeanDeploymentArchive.getServices()` et s'applique seulement au DBA sur lequel ils sont fournis.

Le contrat de `Deployment` nécessite que le conteneur spécifie les extensions portables (voir chapitre 12 des spécifications CDI) qui seront chargées par l'application. Pour aider l'intégration au conteneur, Weld fournit la méthode `Bootstrap.loadExtensions(ClassLoader)` qui chargera les extensions pour le classloader spécifié.

A.1.2. Descripteurs EJB

Weld délègue la découverte des beans EJB 3 au conteneur ce qui permet de ne pas dupliquer la travail fait par le conteneur EJB, et respecter n'importe quelle extension propriétaire de la définition EJB.

Le `EjbDescriptor` doit retourner les méta-données pertinentes comme définies dans la spécification EJB. Chaque interface métier d'un bean session doit être décrite en utilisant un `BusinessInterfaceDescriptor`.

A.1.3. Services d'injection et résolution de ressource EE

Tous les services de ressource EE sont des services par BDA, et peuvent être fournis en utilisant l'une des deux méthodes. La méthode utilisés est à la discrétion de l'intégrateur.

The integrator may choose to provide all EE resource injection services themselves, using another library or framework. In this case the integrator should use the EE environment, and implement the *Section A.1.8, « Services d'injection »* SPI.

Alternatively, the integrator may choose to use CDI to provide EE resource injection. In this case, the `EE_INJECT` environment should be used, and the integrator should implement the *Section A.1.4, « Services EJB » [144]*, *Section A.1.7, « Services de ressource »* and *Section A.1.5, « Services JPA »*.

Important

CDI only provides annotation-based EE resource injection; if you wish to provide deployment descriptor (e.g. `ejb-jar.xml`) injection, you must use *Section A.1.8, « Services d'injection »*.

Si le conteneur effectue l'injection de ressource EE, les ressources injectées doivent être sérialisables. Si l'injection de ressource EE est fournie par Weld, les ressources résolues doivent être sérialisables.

Astuce

Si vous utilisez un environnement non-EE, vous devez alors implémenter n'importe quel SPI de service EE, et Weld fournira les fonctionnalités associées. Il n'est pas nécessaire d'implémenter ces services que vous n'avez pas besoin !

A.1.4. Services EJB

Les services EJB sont divisés en deux interfaces qui sont toutes les deux par BDA.

`EJBServices` est utilisé pour résoudre les EJBs locaux utilisé pour sauvegarder les beans session, et doivent toujours être fournis dans un environnement EE. `EJBServices.resolveEjb(EjbDescriptor ejbDescriptor)` retourne un wrapper—`SessionObjectReference`—atour de la référence EJB. Ce wrapper permet à Weld de demander une référence qui implémente l'interface métier donnée, et, dans le cas des SFSBs, à la fois de demander la suppression de l'EJB dans le conteneur et de demander si l'EJB a été supprimé précédemment.

`EJBResolutionServices.resolveEjb(InjectionPoint ij)` allows the resolution of `@EJB` (for injection into managed beans). This service is not required if the implementation of *Section A.1.8, « Services d'injection »* takes care of `@EJB` injection.

A.1.5. Services JPA

Comme la résolution EJB qui est déléguée au conteneur, la résolution de `@PersistenceContext` pour l'injection dans dans beans managés (avec le `InjectionPoint` fourni), est déléguée au conteneur.

To allow JPA integration, the `JpaServices` interface should be implemented. This service is not required if the implementation of *Section A.1.8, « Services d'injection »* takes care of `@PersistenceContext` injection.

A.1.6. Services de transaction

Weld délègue les activités JTA au conteneur. Le SPI fournit un couple de hooks pour facilement l'atteindre avec l'interface `TransactionServices`.

N'importe quelle implémentation `javax.transaction.Synchronization` peut être passée à la méthode `registerSynchronization()` et l'implémentation du SPI doit immédiatement enregistrer la synchronisation avec la manager de transactions JTA utilisé pour les EJBs.

Pour simplifier la détermination de ce qui est ou non une transaction actuellement actif pour le thread demandeur, la méthode `isTransactionActive()` peut être utilisée. L'implémentation du SPI doit interroger le même manager de transactions JTA utilisé pour les EJBs.

A.1.7. Services de ressource

The resolution of `@Resource` (for injection into managed beans) is delegated to the container. You must provide an implementation of `ResourceServices` which provides these operations. This service is not required if the implementation of *Section A.1.8, « Services d'injection »* takes care of `@Resource` injection.

A.1.8. Services d'injection

Un intégrateur peut souhaiter utiliser `InjectionServices` pour fournir de l'injection par attribut ou par méthode au-delà de ceux fournis par Weld. Une intégration dans un environnement Java EE peut utiliser `InjectionServices` pour fournir l'injection de ressource EE pour les beans managés.

`InjectionServices` fournit un contrat très simple, l'intercepteur `InjectionServices.aroundInject(InjectionContext ic);` sera appelé pour chaque instance que CDI injecte, que ce soit une instance contextuelle, ou une instance non-contextuelle injectée par `InjectionTarget.inject()`.

Le `InjectionContext` peut être utilisé pour découvrir des informations supplémentaires sur l'injection étant effectuée, incluant la `target` étant injectée. `ic.proceed()` doit être appelée pour effectuer l'injection à la façon CDI, et pour appeler les méthodes d'initialisation.

A.1.9. Services de sécurité

Afin d'obtenir le `Principal` représentant l'identité de l'appelant courant, le conteneur doit fournir une implémentation de `SecurityServices`.

A.1.10. Services pour Bean Validation

Afin d'obtenir la `ValidatorFactory` par défaut pour le déploiement de l'application, le conteneur doit fournir une implémentation de `ValidationServices`.

A.1.11. Identifier le BDA en cours d'appréhension

Quand un client effectue une requête à une application qui utilise Weld, la requête peut être adressée à n'importe quels des BDAs de l'application déployée. Pour permettre à Weld de servir correctement la requête, il a besoin de savoir quelle BDA est adressé par la requête. Lorsque cela est possible, Weld fournira un certain contexte, mais son utilisation par l'intégrateur est optionnelle.

Note

La plupart des conteneurs de Servlet utilisé un classloader par war, cela permet de fournir un bon moyen d'identifier la BDA utilisée par les requêtes web.

Quand Weld a besoin d'identifier la BDA, il utilisera l'un de ces services, selon ce qui est utilisé par la requête.

`ServletServices.getBeanDeploymentArchive(ServletContext ctx)`
: Identifier le war en utilisation. Le `ServletContext` est fournit pour un contexte supplémentaire.

A.1.12. Le stockage de beans

Weld utilise une map comme structure pour stocker les instances de bean - `org.jboss.weld.context.api.BeanStore`. Vous pouvez trouver `org.jboss.weld.context.api.helpers.ConcurrentHashMapBeanStore` utile.

A.1.13. Le contexte d'application

Weld s'attend à ce que le Server d'Application ou tout autre conteneur fournisse le stockage de chaque contexte de l'application. Le `org.jboss.weld.context.api.BeanStore` doit être implémenté pour fournir un stockage de portée application.

A.1.14. Initialisation et fermeture

L'interface `org.jboss.weld.bootstrap.api.Bootstrap` définit l'initialisation de Weld, le déploiement de beans et la validation de beans. Pour démarrer Weld, vous devez créer une instance de `org.jboss.weld.bootstrap.WeldBeansBootstrap` (qui implémente `Boostrap`), lui donner les services à utiliser, et ensuite demander que le conteneur démarrer.

Le démarrage est divisé en phases, l'initialisation du conteneur, le déploiement des beans, la validation des beans et la fermeture. L'initialisation créera un manager, et ajoutera les contextes intégrés, et examinera la structure de déploiement. Le déploiement des beans déploiera n'importe quels beans (définis en utilisant les annotations, la programmation, ou ceux intégrés). La validation des beans validera tous les beans.

Pour initialiser le conteneur, vous devez appeler `Bootstrap.startInitialization()`. Avant d'appeler `startInitialization()`, vous devez enregistrer tous les services nécessaire à l'environnement. Vous pouvez faire cela en appelant, par exemple, `bootstrap.getServices().add(JpaServices.class, new MyJpaServices())`. Vous devez aussi fournir le stockage de beans pour le contexte d'application.

Une fois `startInitialization()` appelé, le `Manager` pour chaque BDA peut être obtenu en appelant `Bootstrap.getManager(BeanDeploymentArchive bda)`.

Pour déployer les beans découverts, appelez `Bootstrap.deployBeans()`.

Pour valider les beans déployés, appelez `Bootstrap.validateBeans()`.

Pour placer le conteneur dans un état où il pourra servir les requêtes, appelez `Bootstrap.endInitialization()`.

Pour étendre le conteneur, appelez `Bootstrap.shutdown()`. Ceci permet au conteneur d'effectuer toutes les opérations de nettoyage nécessaires.

A.1.15. Chargement de ressource

Weld a besoin de charger des classes et des ressources du classpath à différents moments. Par défaut, ils sont chargés depuis le Thread Context ClassLoader si disponible, s'il ne s'agit pas du même classloader utilisé pour charger Weld, cependant cela peut ne pas être correct pour quelques environnements. Dans ce cas, vous pouvez implémenter `org.jboss.weld.spi.ResourceLoader`.

A.2. Le contrat avec le conteneur

Il y a un nombre d'exigences que Weld place sur le conteneur pour fonctionner correctement qui ne relève pas la mise en oeuvre d'API.

Isolation du classloader
: Si vous intégrez Weld dans un environnement qui supporte le déploiement de plusieurs applications, vous devez activer, automatiquement, ou au travers d'une configuration utilisateur, l'isolation du classloader pour chaque application CDI.

Servlet
: Si vous intégrez Weld dans un environnement Servlet, vous devez enregistrer `org.jboss.weld.servlet.WeldListener` en tant que Servlet listener, automatiquement ou au travers d'une configuration utilisateur, pour chaque application CDI qui utilise Servlet.

  Vous devez vous assurer que `WeldListener.contextInitialized()` est appelé après que les beans soit complètement déployés (`Bootstrap.deployBeans()` a été appelé).

JSF
: Si vous intégrez Weld dans un environnement JSF, vous devez enregistrer `org.jboss.weld.jsf.WeldPhaseListener` en tant que phase listener.

  Si vous intégrez Weld dans un environnement JSF, vous devez enregistrer `org.jboss.weld.el.WeldELContextListener` en tant que EL Context listener.

  Si vous intégrez Weld dans un environnement JSF, vous devez enregistrer `org.jboss.weld.jsf.ConversationAwareViewHandler` en tant que delegating view handler.

  Si vous intégrez Weld dans un environnement JSF, vous devez obtenir le manager de beans du module et ensuite appeler `BeanManager.wrapExpressionFactory()`, en passant

`Application.getExpressionFactory()` comme argument. L'expression factory wrappée doit être utilisée dans toutes les évaluations d'expression EL effectuées par JSF dans l'application web.

Si vous intégrez Weld dans un environnement JSF, vous devez obtenir le manager de beans du module et ensuite appeler `BeanManager.getELResolver()`. Le EL resolver retourné doit être enregistré avec JSF pour cette application web.

Astuce

Il existe plusieurs façons d'obtenir le manager de beans du module. Vous pouvez appeler `Bootstrap.getManager()`, en passant dans la BDA de ce module. Sinon, vous pouvez utilise l'injection dans les classes de composant Java EE, et rechercher le manager de beans dans JNDI.

Si vous intégrez Weld dans un environnement JSF, vous devez enregistrer `org.jboss.weld.servlet.ConversationPropagationFilter` en tant que Servlet listener, automatiquement ou au travers d'une configuration utilisateur, pour chaque application CDI qui utilise JSF. Ce filtre peut être enregistré sans danger pour tous les déploiements de Servlet.

Note

Weld prend seulement en charger les versions JSF 1.2 et au dessus.

JSP

Si vous intégrez Weld dans un environnement JSP, vous devez enregistrer `org.jboss.weld.el.WeldELContextListener` en tant que EL Context listener.

Si vous intégrez Weld dans un environnement JSP, vous devez obtenir le manager de beans du module et ensuite appeler `BeanManager.wrapExpressionFactory()` en passant `Application.getExpressionFactory()` comme argument. La expression factory wrappée doit être utilisée dans toutes les évaluations d'expression EL effectuées par JSP.

Si vous intégrez Weld dans un environnement JSP, vous devez obtenir le manager de beans du module et ensuite appeler `BeanManager.getELResolver()`. Le EL resolver retourné doit être enregistrer avec JSP pour cette application web.

Astuce

Il existe plusieurs façons d'obtenir le manager de beans du module. Vous pouvez appeler `Bootstrap.getManager()`, en passant dans la BDA de ce module. Sinon, vous pouvez utilise l'injection dans les classes de composant Java EE, et rechercher le manager de beans dans JNDI.

Intercepteur de Bean Session

Si vous intégrez Weld dans un environnement EJB, vous devez enregistrer la méthode `aroundInvoke` de `org.jboss.weld.ejb.SessionBeanInterceptor` en tant qu'intercepteur EJB around-invoke pour tous les EJBs dans l'application, automatiquement ou au travers d'une configuration utilisateur, pour chaque application CDI qui utilise des beans entreprise. Si vous utilisez un environnement EJB 3.1, vous devez aussi enregistrer cette méthode en tant qu'intercepteur around-timeout.

Important

You must register the `SessionBeanInterceptor` as the inner most interceptor in the stack for all EJBs.

Le `weld-core.jar`

Weld peut résider dans un classloader isolé, ou un classloader partagé. Si vous choisissez d'utiliser un classloader isolé, le `SingletonProvider` par défaut, `IsolatedStaticSingletonProvider`, peut être utilisé. Si vous choisissez d'utiliser un classloader partagé, vous aurez alors besoin de choisir une autre stratégie.

Vous pouvez fournir votre propre implémentation de `Singleton` et `SingletonProvider`, et les enregistrer pour les utiliser en utilisant `SingletonProvider.initialize(SingletonProvider provider)`.

Weld fournit également une implémentation de Thread Context Classloader par stratégie d'application, via le `TCCLSingletonProvider`.

Lier le manager dans JNDI

Vous devez lier le manager de beans pour l'archive de déploiement de beans dans JNDI à `java:comp/BeanManager`. Le type doit être `javax.enterprise.inject.spi.BeanManager`. Pour obtenir le bon manager de beans pour l'archive de déploiement de beans, vous devez appeler `bootstrap.getBeanManager(beanDeploymentArchive)`.

Effectuer l'injection CDI sur des classes de composant Java EE

La spécification CDI exige que le conteneur fournisse l'injection dans des ressources non contextuelles pour toutes les classes de composant Java EE. Weld délègue cette responsabilité au conteneur. Ceci peut être réalisé en utilisant le SPI défini de CDI `InjectionTarget`. De plus, vous devez effectuer cette opération sur le bon manager de bean de l'archive de déploiement de bean contenant la classe du composant EE.

La spécification CDI exige aussi qu'un évènement `ProcessInjectionTarget` soit levé pour chaque classe de composant Java EE. De plus, si un observateur appelle `ProcessInjectionTarget.setInjectionTarget()`, le conteneur doit utiliser la target d'injection *spécifiée* pour effectuer l'injection.

Pour aider l'ingrateur, Weld fournit `WeldManager.fireProcessInjectionTarget()` qui retourne le `InjectionTarget` à utiliser.

```
// Fire ProcessInjectionTarget, returning the InjectionTarget
// to use
InjectionTarget it = weldBeanManager.fireProcessInjectionTarget(clazz);

// Per instance required, create the creational context
CreationalContext<?> cc = beanManager.createCreationalContext(null);

// Produce the instance, performing any constructor injection required
Object instance = it.produce();

// Perform injection and call initializers
it.inject(instance, cc);

// Call the post-construct callback
it.postConstruct(instance);

// Call the pre-destroy callback
```

```
it.preDestroy(instance);

// Clean up the instance
it.dispose();
cc.release();
```

Le conteneur peut intercaler d'autres opérations entre ces appels. De plus, l'intégrateur peut choisir d'implémenter n'importe quelles de ces appels d'une autre façon en s'assurant que le contrat est rempli.

Lors de l'exécution des injections dans les EJBs, vous devez utiliser le SPI Weld défini, `WeldManager`. De plus, vous devez effectuer cette opération sur le bon manager de beans du l'archive de déploiement de beans contenant l'EJB.

```
// Obtain the EjbDescriptor for the EJB
// You may choose to use this utility method to get the descriptor
EjbDescriptor<?> ejbDescriptor = beanManager.getEjbDescriptor(ejbName);

// Get an the Bean object
Bean<?> bean = beanManager.getBean(ejbDescriptor);

// Create the injection target
InjectionTarget it = deploymentBeanManager.createInjectionTarget(ejbDescriptor);

// Per instance required, create the creational context
CreationalContext<?> cc = deploymentBeanManager.createCreationalContext(bean);

// Perform injection and call initializers
it.inject(instance, cc);

// You may choose to have CDI call the post construct and pre destroy
// lifecycle callbacks

// Call the post-construct callback
it.postConstruct(instance);

// Call the pre-destroy callback
it.preDestroy(instance);

// Clean up the instance
it.dispose();
cc.release();
```