Session Beans et Timer Service

Java EE 6 - Sommaire

- Introduction
- Session beans
  - Stateless Beans
  - Stateful Beans
  - Singletons
    - Initialisation
    - Chaînage
    - Concurrence d'accès
  - Modèle de développement
    - Interfaces et classe de bean

Java EE 6 - Sommaire

- Vue cliente
- Descripteur de déploiement
- Variables d'environnement de contexte
- Appels asynchrones
- Conteneur embarqué
- Timer Service
  - Expressions
  - Création automatique des timers
  - Création programmative des timers

Introduction

- Les entités JPA encapsulent les données, le mapping relationnel et parfois la logique de validation
- La couche métier gère ces entités ou objets persistents par les sessions beans
- Cette séparation logique suit le paradigme de ”séparation des responsabilités” ou séparation en couche, où chaque couche recoupe aussi peu que possible les autres couches

Introduction

- Les session beans sont de trois types : Stateless, Stateful et Singleton
- Les Singleton Session Beans sont apparus avec la spécification EJB 3.1

Session Beans

- Les sessions beans sont parfaits pour implémenter la logique métier, les processus et les traitements
- Il faut choisir parmi les trois types de Session beans en fonction des besoins

Session Beans

- Stateless : ces session beans ne maintiennent aucun état entre deux appels d'un même client. Ils sont utiles pour les tâches qui peuvent se gérer en un seul appel. Ce sont les plus performants des Session Beans car les instances deviennent immédiatement disponibles pour un autre appel

Session Beans

- Stateful : ces session beans maintiennent un état entre deux appels d'un même client, c'est à dire que le session bean retrouve la valeur de ses attributs. La sauvegarde et mise à jour de ces attributs est gérée automatiquement par le conteneur, mais cela demande des ressources supplémentaires par rapport aux stateless session beans. Ces beans sont utiles pour les tâches qui doivent se réaliser en plusieurs étapes

Session Beans

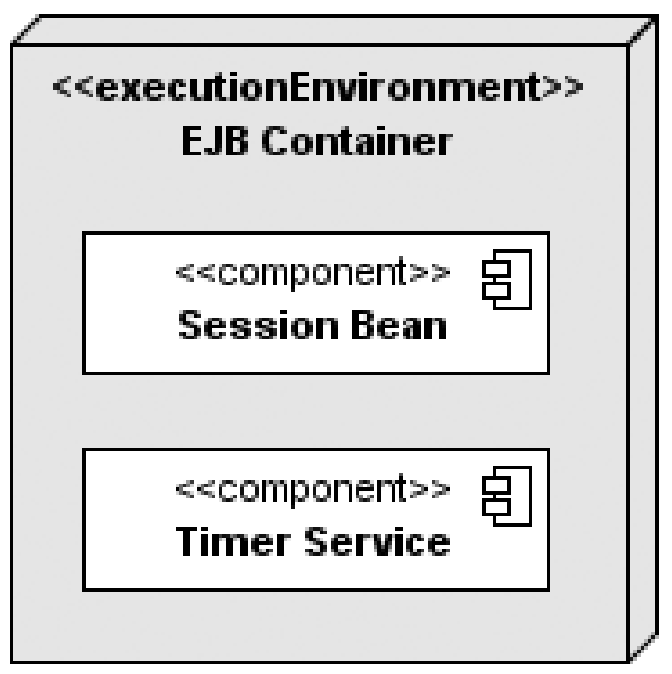
- Singletons : ces beans suivent le design pattern singleton, c'est à dire que le conteneur fait en sorte qu'une seule instance de ce bean existe pour toute l'application

Session Beans

- Les trois types de session beans ont des fonctionnalités spécifiques, mais partagent aussi beaucoup de similarités :
  - Le même mode de programmation
  - Peuvent avoir des interfaces locale et distante ou aucune
  - Doivent être packagés dans une archive (jar, war ou ear) pour être gérés par le conteneur
  - Le conteneur gère leur cycle de vie, les transactions, intercepteurs, sécurité, ...

Session Beans

- Vue d'ensemble du conteneur d'EJB



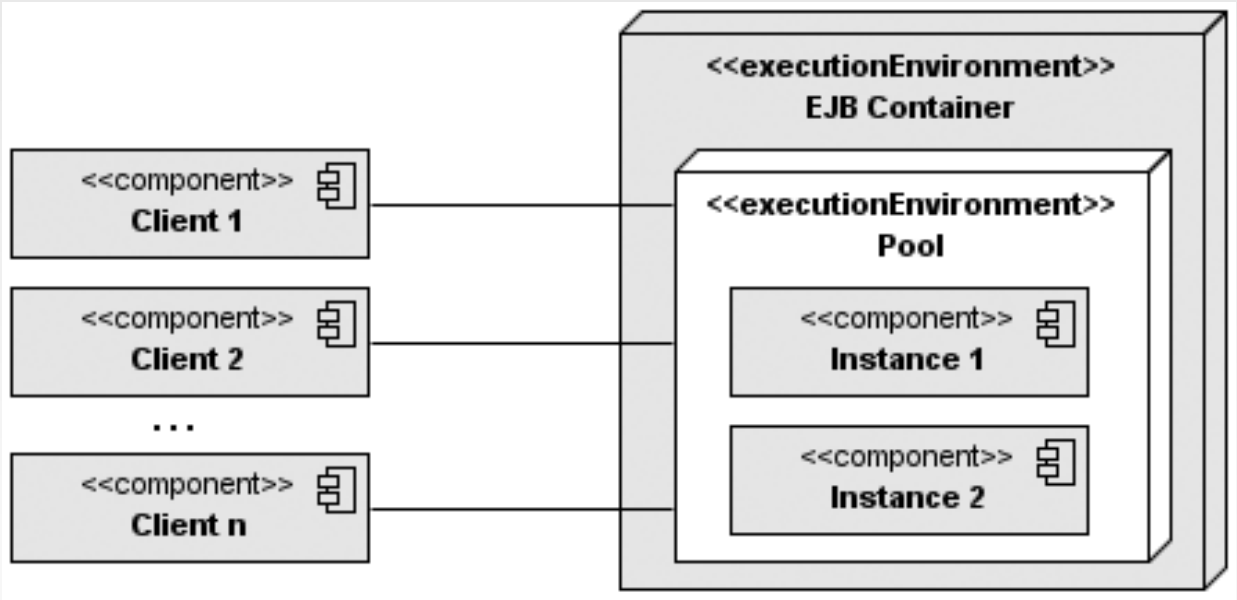
Stateless Beans

- Les stateless beans sont les plus connus et les plus populaires du fait de leur simplicité, leur puissance et leur efficacité
- Les tâches des session beans doivent pouvoir être réalisées en un seul appel de méthode

```
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy series created by Douglas Adams.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
book.setIllustrations(false);
statelessComponent.persistToDatabase(book);
```

Stateless Beans

- Le conteneur de session beans gère généralement un pool d'instances partagées entre tous les clients



Stateless Beans

- Un Stateless Session Bean est une simple classe Java (POJO) annotée par @javax.ejb.Stateless
- Du fait de la gestion par le conteneur, on peut utiliser l'injection de dépendances, par exemple avec l'annotation @PersistenceContext
- Le contexte est transactionnel pour les Session Beans, ce qui signifie que chaque appel de méthode du Session Bean s'exécution dans le cadre d'une transaction

Stateless Beans

```
@Stateless
public class ItemEJB {
        @PersistenceContext(unitName = "chapter07PU")
        private EntityManager em;

        public List<Book> findBooks() {
            Query query = em.createNamedQuery("findAllBooks");
            return query.getResultList();
        }
        public List<CD> findCDs() {
            Query query = em.createNamedQuery("findAllCDs");
            return query.getResultList();
        }
        public Book createBook(Book book) {
            em.persist(book);
            return book;
        }
        public CD createCD(CD cd) {
            em.persist(cd);
            return cd;
        }
    }
```

Stateful Beans

- Les Stateful beans, du fait qu'ils conservent un état entre deux appels d'un même client, sont utiles pour des tâches qui se réalisent en plusieurs étapes
- L'exemple classique est celui du panier d'un site de e-commerce : un client choisit un livre et l'ajoute à son panier, puis choisit un autre livre et l'ajoute au panier. A la fin le client paye pour le contenu de son panier. Le panier doit garder le contenu sélectionné par le client

Stateful Beans

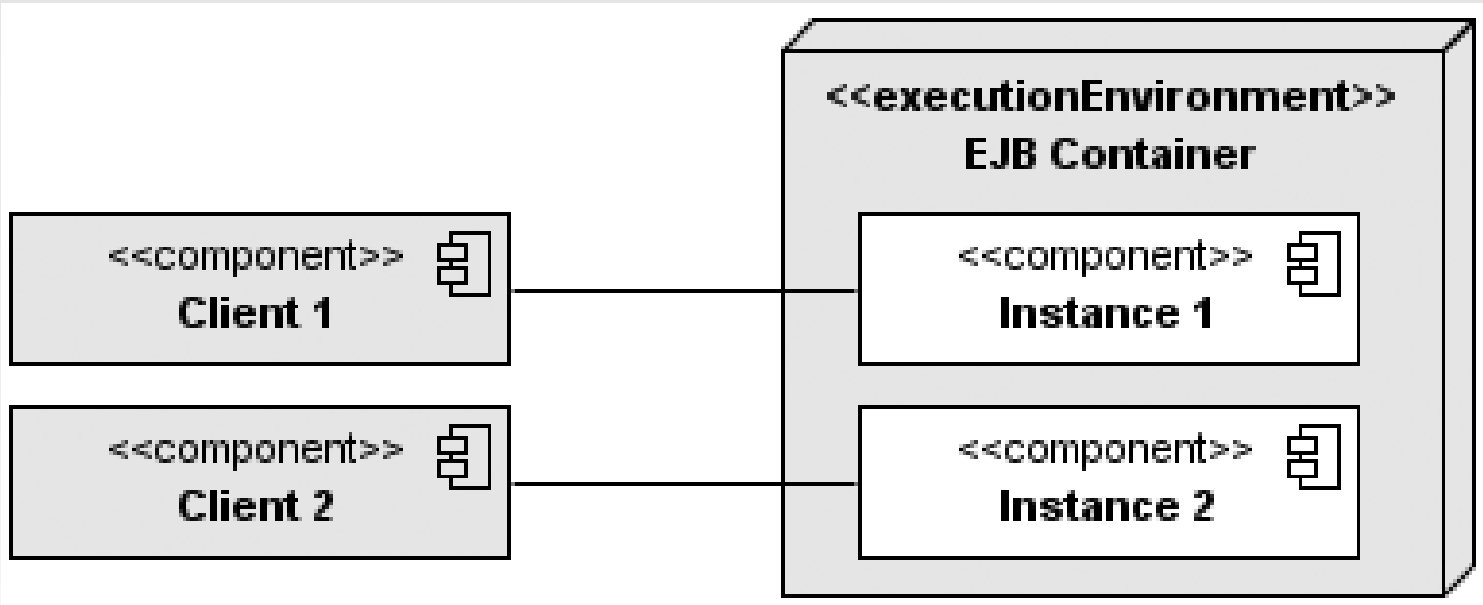
```
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy series created by Douglas Adams.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
book.setIllustrations(false);
statefullComponent.addBookToShoppingCart(book);

book.setTitle("The Robots of Dawn");
book.setPrice(18.25F);
book.setDescription("Isaac Asimov's Robot Series");
book.setIsbn("0-553-29949-2");
book.setNbOfPage(276);
book.setIllustrations(false);
statefullComponent.addBookToShoppingCart(book);

statefullComponent.checkOutShoppingCart();
```

Stateful Beans

- Lien entre stateful bean et client



Stateful Beans

- Pour ne pas avoir autant d'instances de stateful beans que de clients connectés, un mécanisme de nettoyage temporaire de la mémoire entre deux appels est utilisé : passivation et activation. Ce mécanisme est géré automatiquement par le conteneur.
- Du fait de leur lien 1-1 avec le client et du besoin de les passiver-activer, les stateful beans ont un prix en terme de performances
- On devra parfois libérer des ressources (connexions) à la passivation et les réactiver à l'activation

Stateful Beans

- Un Stateful Session Bean est une simple classe Java (POJO) annotée par @javax.ejb.Stateful
- On peut aider le conteneur avec les annotations suivantes :
  - @javax.ejb.StatefulTimeout(delayInMilliseconds) pour définir le timeout de la conversation avec le client
  - @javax.ejb.Remove pour préciser qu'après l'appel de méthode l'instance peut être supprimée de la mémoire

Stateful Beans

```
@Stateful
@StatefulTimeout(20000)
public class ShoppingCartEJB {
    private List<Item> cartItems = new ArrayList<Item>();
    public void addItem(Item item) {
        if (!cartItems.contains(item))
            cartItems.add(item);
    }
    public void removeItem(Item item) {
          if (cartItems.contains(item))
          cartItems.remove(item);
  }
  public Float getTotal() {
      if (cartItems == null || cartItems.isEmpty())
          return 0f;
      Float total = 0f;
      for (Item cartItem : cartItems) {
          total += (cartItem.getPrice());
      }
      return total;
  }
```

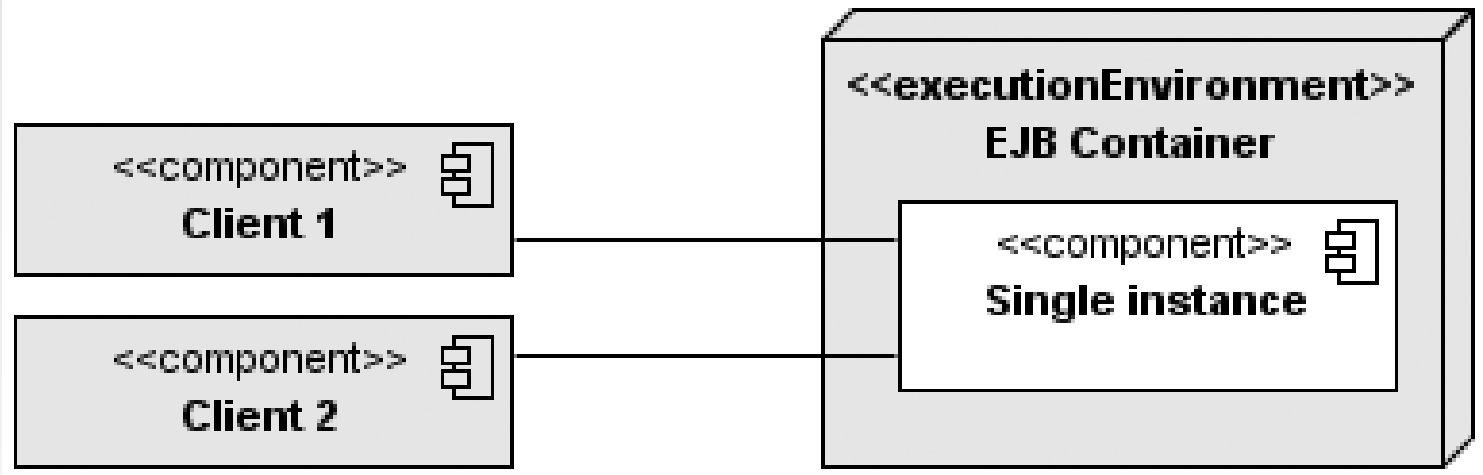
```
  @Remove
  public void checkout() {
      // Business logic
      cartItems.clear();
  }
  @Remove
  public void empty() {
      cartItems.clear();
  }
}
```

Singletons

- Un sigleton est un session bean qui possède une seule instance par application
- Il arrive très souvent qu'une application ait besoin d'une seule instance d'un objet : une souris, un gestionnaire de fenêtres, une file d'attente d'impression, un système de fichiers, un système de cache, …
- Les singletons maintiennent leur état entre les appels clients

Singletons

- Lien entre singleton et client



Singletons

- Pour écrire un singleton session bean, il suffit d'annoter avec @javax.ejb.Singleton une simple classe Java (POJO)

```
@Singleton
public class CacheEJB {
    private Map<Long, Object> cache = new HashMap<Long, Object>();
    public void addToCache(Long id, Object object) {
        if (!cache.containsKey(id))
            cache.put(id, object);
    }
    public void removeFromCache(Long id) {
        if (cache.containsKey(id))
            cache.remove(id);
    }
    public Object getFromCache(Long id) {
        if (cache.containsKey(id))
            return cache.get(id);
        else
                return null;
        }
    }
```

Singletons

- Les singletons ont quelques fonctionnalités supplémentaires par rapport aux stateless et stateful beans : ils peuvent être initialisés, chaînés et on peut personnaliser leur accès concurrent

Singletons : Initialisation

- L'initialisation d'un singleton, en fonction de sa fonction spécifique, peut prendre du temps, par exemple pour le système de cache qui irait mettre en mémoire des milliers d'objets de la base de données
- Pour éviter de faire attendre le premier client qui déclenche l'initialisation, il est possible de demander au conteneur d'initialiser un singleton au démarrage de l'application avec l'annotation @javax.ejb.Startup

Singletons : Initialisation

```
@Singleton
@Startup
public class CacheEJB {
    // ...
}
```

Singletons : Chaînage

- Dans certains cas, quand il y a plusieurs singletons, l'ordre d'initialisation peut être important
- Par exemple si le CacheEJB doit stocker des données qui viennent d'un autre singleton (CountryCodeEJB qui retourne les code ISO des pays), il faut que l'autre singleton soit initialisé avant
- L'annotation @javax.ejb.DependsOn exprime cette dépendance

Singletons : Chaînage

```
@Singleton
public class CountryCodeEJB {
    ...
}

@Singleton
public class ZipCodeEJB {
    ...
}

@DependsOn("CountryCodeEJB", "ZipCodeEJB")
@Startup
@Singleton
public class CacheEJB {
    ...
}
```

Singletons : Concurrence d'accès

- Il y a une seule instance pour tous les clients. Il peut donc y avoir un accès concurrent. L'accès concurrent peut être contrôlé avec l'annotation @javax.ejb.ConcurrencyManagement de trois façons différentes :
  - Container-managed concurrency (CMC) : l'accès concurrent se base sur les métadonnées (annotations et/ou description XML)
  - Bean-managed concurrency (BMC) : le conteneur délègue la responsabilité au bean

Singletons : Concurrence d'accès

- Concurrence d'accès interdite : si un client appelle une méthode déjà en cours d'utilisation par un autre client, l'appel renverra une ConcurrentAccessException
- Si aucun mode de concurrence d'accès n'est spécifié, le mode container-managed (CMC) est choisi
- Un singleton ne peut choisir qu'un seul mode

Singletons : Concurrence d'accès

- Container-managed concurrency
  - Dans ce mode, vous utilisez l'annotation @Lock avec la valeur READ (partagé) ou WRITE (exclusif) pour préciser comment le conteneur doit gérer la concurrence d'accès. **Si rien n'est précisé, WRITE est choisi**
  - @Lock(LockType.WRITE) : ne permet pas qu'une méthode soit accessible par deux clients simultanément
  - @Lock(LockType.READ) : plusieurs clients peuvent appeler simultanément la méthode

Singletons : Concurrence d'accès

```
@Singleton
@Lock(LockType.READ)
public class CacheEJB {
    private Map<Long, Object> cache = new HashMap<Long, Object>();
    public void addToCache(Long id, Object object) {
        if (!cache.containsKey(id))
            cache.put(id, object);
    }
    public void removeFromCache(Long id) {
        if (cache.containsKey(id))
            cache.remove(id);
    }
    @AccessTimeout(2000)
    @Lock(LockType.WRITE)
    public Object getFromCache(Long id) {
        if (cache.containsKey(id))
            return cache.get(id);
        else
            return null;
    }
}
```

Singletons : Concurrence d'accès

- Bean-managed concurrency
  - Dans ce mode, le bean gère lui-même la concurrence d'accès. Il peut utiliser les mots-clés **synchronized** et **volatile**

```
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class CacheEJB {
    private Map<Long, Object> cache = new HashMap<Long, Object>();
    public synchronized void addToCache(Long id, Object object) {
        if (!cache.containsKey(id))
            cache.put(id, object);
    }
    public void removeFromCache(Long id) {
        if (cache.containsKey(id))
            cache.remove(id);
    }
    public synchronized Object getFromCache(Long id) {
        if (cache.containsKey(id))
            return cache.get(id);
        else
            return null;
    }
}
```

Singletons : Concurrence d'accès

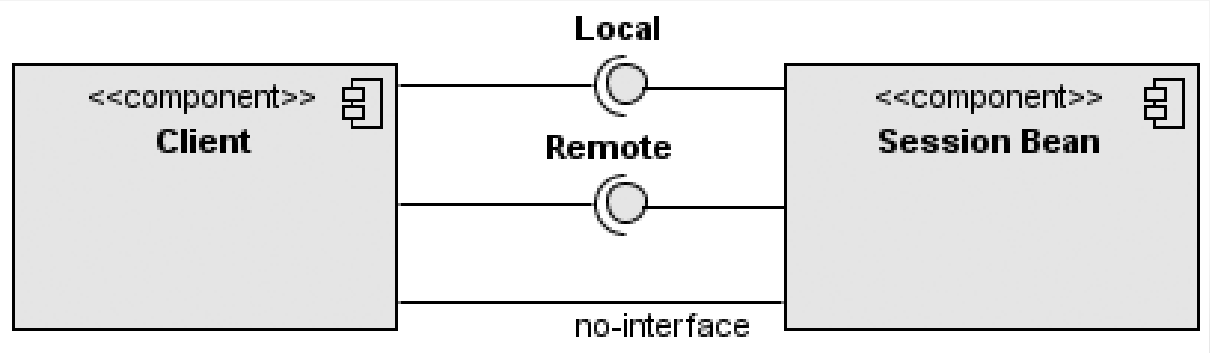
- Concurrence d'accès interdite
  - Dans ce mode, l'exception ConcurrentAccessException sera levée en cas d'accès concurrent
  - Comme pour les autres modes, on peut choisir de l'appliquer à une seule méthode ou au bean entier

Singletons : Concurrence d'accès

```
@Singleton
@Lock(LockType.READ)
public class CacheEJB {
	private Map<Long, Object> cache = new HashMap<Long, Object>();
	@ConcurrencyManagement(ConcurrencyManagementType.CONCURRENCY_NOT_ALLOWED)
	public void addToCache(Long id, Object object) {
	    if (!cache.containsKey(id))
	        cache.put(id, object);
	}
	public void removeFromCache(Long id) {
	    if (cache.containsKey(id))
	        cache.remove(id);
	}
	@AccessTimeout(2000)
	@Lock(LockType.WRITE)
	public Object getFromCache(Long id) {
	    if (cache.containsKey(id))
	        return cache.get(id);
	    else
	        return null;
	}
}
```

Modèle de développement

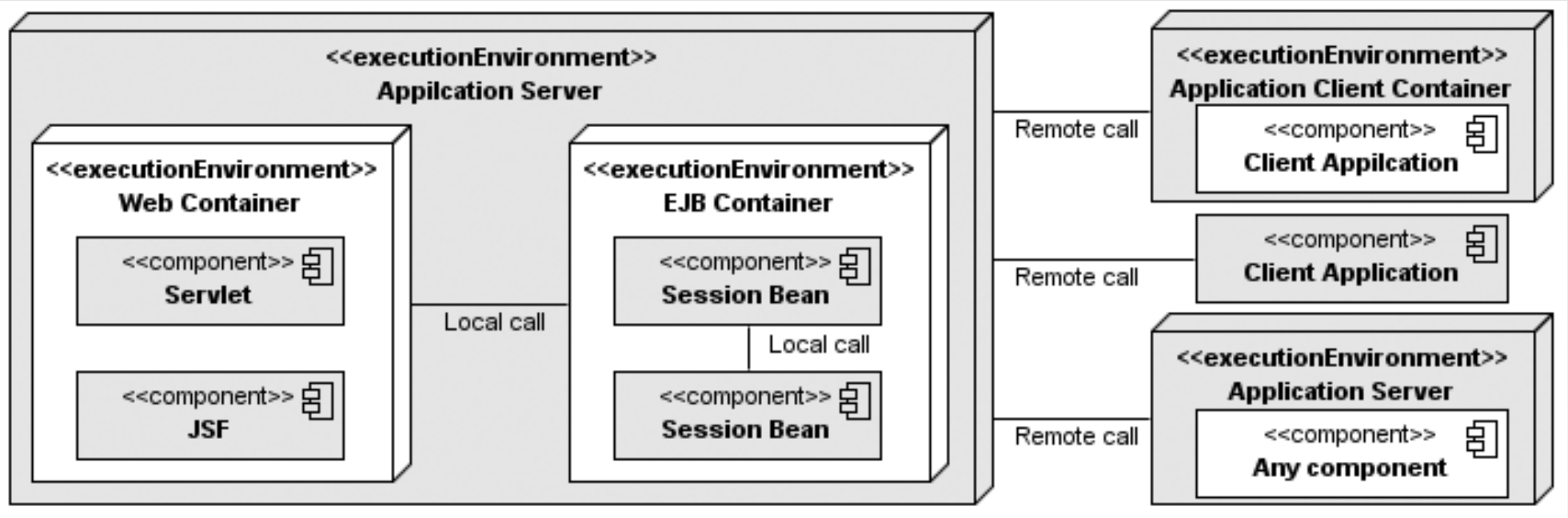
- Interfaces et classe du bean
  - Les session beans ont parfois besoin d'interfaces pour définir la visibilité des méthodes du bean. On peut définir une interface locale, distante ou aucune
  - La classe du bean contient l'implémentation des méthodes et peut implémenter zéro ou plusieurs interfaces. La classe du bean est annotée par @Stateless, @Stateful ou @Singleton



Modèle de développement

- Le besoin en interfaces dépend de la visibilité qu'on veut donner au bean :
  - Pour invoquer un bean depuis une autre machine virtuelle, il faut que le bean implémente une interface remote
  - Pour invoquer un bean dans la même machine virtuelle, il n'y a pas besoin d'interface. Dans ce cas toutes les méthodes publiques du bean sont visibles. Pour limiter la visibilité, on peut définir une interface local

Modèle de développement



Modèle de développement

```
@Local
public interface ItemLocal {
    List<Book> findBooks();
    List<CD> findCDs();
}
@Remote
public interface ItemRemote {
    List<Book> findBooks();
    List<CD> findCDs();
    Book createBook(Book book);
    CD createCD(CD cd);
}
@Stateless
public class ItemEJB
       implements ItemLocal, ItemRemote {
    ...
}
```

```
public interface ItemLocal {
    List<Book> findBooks();
    List<CD> findCDs();
}
public interface ItemRemote {
    List<Book> findBooks();
    List<CD> findCDs();
    Book createBook(Book book);
    CD createCD(CD cd);
}
@Stateless
@Remote (ItemRemote)
@Local (ItemLocal)
public class ItemEJB
       implements ItemLocal, ItemRemote {
    ...
}
```

Modèle de développement

- En plus des appels distants via RMI, les stateless beans peuvent aussi être invoqués de façon distante en tant que web services SOAP ou REST

```
@Local
public interface ItemLocal {
    List<Book> findBooks();
    List<CD> findCDs();
}
@WebService
public interface ItemWeb {
    List<Book> findBooks();
    List<CD> findCDs();
    Book createBook(Book book);
    CD createCD(CD cd);
}
```

```
@Path(/items)
public interface ItemRest {
    List<Book> findBooks();
}
@Stateless
public class ItemEJB
        implements ItemLocal, ItemWeb, ItemRest {
    ...
}
```

Modèle de développement

- Les contraintes de la classe de bean sont :
  - Avoir une annotation @Stateless, @Stateful ou @Singleton, ou un équivalent XML
  - Implémenter les méthodes de ses interfaces
  - Être public, ne peut pas être final ou abstract
  - Avoir un constructeur public sans argument
  - Ne doit pas avoir de méthode finalize()
  - Les noms des méthodes métier ne peuvent commencer par ejb et ne peuvent être static ou final
  - Les arguments et valeur de retour des méthodes remote doivent être des types RMI valides

Modèle de développement

- Pour interagir avec un EJB, il faut d'abord obtenir une référence sur cet EJB car on n'utilise pas l'opérateur new
- Il y a 2 façon d'obtenir une référence :
  - Par injection de dépendances avec l'annotation @EJB
  - Via une recherche JNDI

Modèle de développement

- Injection de dépendances avec @EJB
  - L'annotation @EJB permet d'injecter la référence sur un Session Bean
  - L'injection de dépendances n'est possible que dans des environnements managés comme les conteneurs d'EJB, conteneurs web et conteneur d'applications clientes

Modèle de développement

- Session Bean sans interface

```
@Stateless
public class ItemEJB {
    ...
}
// Client code
@EJB ItemEJB itemEJB;
```

Modèle de développement

- Session Bean avec interfaces

```
@Stateless
@Remote (ItemRemote)
@Local (ItemLocal)
public class ItemEJB
      implements ItemLocal, ItemRemote {
   ...
}
// Client code
@EJB ItemEJB itemEJB; // Not possible
@EJB ItemLocal itemEJBLocal;
@EJB ItemRemote itemEJBRemote;
```

```
@Stateless
@Remote (ItemRemote)
@Local (ItemLocal)
@LocalBean
public class ItemEJB
      implements ItemLocal, ItemRemote {
   ...
}
// Client code
@EJB ItemEJB itemEJB;
@EJB ItemLocal itemEJBLocal;
@EJB ItemRemote itemEJBRemote;
```

Modèle de développement

- Recherche JNDI
  - Quand un session bean est déployé dans le conteneur, il est automatiquement référencé dans l'arbre JNDI
  - Jusqu'à Java EE 6, le nom n'était pas standardisé et variait d'un serveur à l'autre
  - Java EE 6 définit le nom de la façon suivante :

java:global[/<app-name>]/<module-name>/<bean-name> [!<fully-qualified-interface-name>]

  - <app-name> : pour les ear (par défaut son nom)
  - <module-name> : par défaut le nom de l'archive (jar ou war)

Modèle de développement

- Exemple :
  - ItemEJB = <bean-name>
  - Déployé dans cdbookstore.jar = <module-name>
  - A une remote interface et no-interface (@LocalBean)

```
package com.apress.javaee6;
@Stateless
@LocalBean
@Remote (ItemRemote)
public class ItemEJB implements ItemRemote {
    ...
}
// JNDI name
java:global/cdbookstore/ItemEJB!com.apress.javaee6.ItemEJB
java:global/cdbookstore/ItemEJB!com.apress.javaee6.ItemRemote
```

Modèle de développement

- Recherche JNDI
  - Si le session bean implémente seulement une interface local ou aucune interface, le conteneur ajoute une autre entrée pour trouver le bean :

```
java:global[/<app-name>]/<module-name>/<bean-name>
```

  - Exemple :

```
package com.apress.javaee6;
@Stateless
public class ItemEJB {
    ...
}
// JNDI name
java:global/cdbookstore/ItemEJB
```

Modèle de développement

- SessionContext
  - Les session beans sont des composants qui s'exécutent dans un conteneur. Normalement, ils n'ont pas accès au conteneur ou n'appelent pas de services du conteneur directement (transaction, sécurité, injection de dépendance, …) puisque le conteneur les gère de façon transparente pour le bean. Mais parfois nécessaire d'accéder au conteneur. On peut pour cela utiliser l'interface javax.ejb.SessionContext qui étend EJBContext

Modèle de développement

- Quelques méthodes de SessionContext

Method	Description
`getCallerPrincipal`	Returns the `java.security.Principal` associated with the invocation.
`getRollbackOnly`	Tests whether the current transaction has been marked for rollback.
`getTimerService`	Returns the `javax.ejb.TimerService` interface. Only stateless beans and singletons can use this method. Stateful session beans cannot be timed objects.
`getUserTransaction`	Returns the `javax.transaction.UserTransaction` interface to demarcate transactions. Only session beans with bean-managed transaction (BMT) can use this method.
`isCallerInRole`	Tests whether the caller has a given security role.
`lookup`	Enables the session bean to look up its environment entries in the JNDI naming context.
`setRollbackOnly`	Allows the bean to mark the current transaction for rollback. Only session beans with BMT can use this method.
`wasCancelCalled`	Checks whether a client invoked the `cancel()` method on the client `Future` object corresponding to the currently executing asynchronous business method.

Modèle de développement

- Récupération du SessionContext

```
@Stateless
public class ItemEJB {
    @Resource
    private SessionContext context;
    ...
    public Book createBook(Book book) {
        ...
        if (cantFindAuthor())
            context.setRollbackOnly();
    }
}
```

Modèle de développement

- Descripteur de déploiement XML
  - Java EE 6 utilise le principe de configuration par l'exception, ie la configuration est l'exception à la configuration par défaut
  - Il est possible d'utiliser aussi bien les annotations que des fichiers XML pour définir cette configuration des services (transaction, cycle de vie, sécurité, intercepteurs, concurrence d'accès, ...)
  - Les annotations comme les fichiers XML attachent des informations à une classe, une interface, une méthode ou une variable

Modèle de développement

- Descripteur de déploiement XML
  - Toutes les balises ont leur équivalent XML
  - Si une annotation et un XML équivalent sont tous deux présent, le XML prédomine
  - Le fichier de description s'appelle ejb-jar.xml
  - S'il est déployé dans un fichier jar, il doit être placé dans le dossier META-INF/ejb-jar.xml
  - S'il est déployé dans un fichier war, il doit être placé dans le dossier WEB-INF/ejb-jar.xml

Modèle de développement

- Exemple de descripteur de déploiement

```
<ejb-jar>
   <enterprise-beans>
      <session>
         <ejb-name>ItemEJB</ejb-name>
         <ejb-class>com.apress.javaee6.ItemEJB</ejb-class>
         <local>com.apress.javaee6.ItemLocal</local>
         <remote>com.apress.javaee6.ItemLocal</remote>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
         <env-entry>
            <env-entry-name>aBookTitle</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>Beginning Java EE 6</env-entry-value>
         </env-entry>
      </session>
   </enterprise-beans>
</ejb-jar>
```

Modèle de développement

- Injection de dépendance
  - Ce mécanisme permet de recevoir une référence plutôt que de devoir la rechercher
  - Dans Java EE, le conteneur est capable d'injecter des références à des objets plutôt que d'avoir à rechercher ces références en utilisant JNDI
  - Le conteneur est capable d'injecter différents types de ressources dans des sessions beans en fonction des annotations utilisées

Modèle de développement

- Injection de dépendance
  - @EJB : injecte une référence vers une interface locale ou distante ou la vue de l'EJB dans la variable annotée
  - @PersistenceContext et @PersistenceUnit : injecte une dépendance sur un EntityManager et un EntityManagerFactory respectivement
  - @WebServiceRef : injecte une référence sur un web service
  - @Resource : injecte des ressources comme des datasources JDBC, des contextes de session, des transactions utilisateurs, le timer service, ...

Modèle de développement

- Exemple d'injection de dépendances

```
@Stateless
public class ItemEJB {
    @PersistenceContext(unitName = "chapter07PU")
    private EntityManager em;
    @EJB
    private CustomerEJB customerEJB;
    @WebServiceRef
    private ArtistWebService artistWebService;
    private SessionContext context;
    @Resource
    public void setCtx(SessionContext ctx) {
        this.ctx = ctx;
    }
    ...
}
```

Modèle de développement

- Variables d'environnement de contexte
  - Dans des applications d'entreprise, il y a des situations où certains paramètres changent d'un environnement de déploiement à l'autre (par exemple en fonction de pays où elle est déployée, de la version de l'application, …)

```
@Stateless
public class ItemConverterEJB {
        public Item convertPrice(Item item) {
            item.setPrice(item.getPrice() * 0.80);
            item.setCurrency("Euros");
            return item;
        }
    }
```

Modèle de développement

- Variables d'environnement de contexte
  - Le problème de l'écriture en dur de ces paramètres dans le code, c'est qu'il faut changer le code, recompiler et redéployer pour chaque environnement
  - Une autre possibilité, c'est d'utiliser la base de données pour stocker ces valeurs, mais c'est une consommation de ressources inutiles
  - Le plus simple consiste à stocker ces valeurs dans le descripteur de déploiement. Ces valeurs peuvent ensuite être injectées par injection de dépendances

Modèle de développement

```
<ejb-jar>
   <enterprise-beans>
      <session>
         <ejb-name>ItemConverterEJB</ejb-name>
         <ejb-class>com.apress.javaee6.ItemConverterEJB</ejb-class>
         <env-entry>
            <env-entry-name>currencyEntry</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>Euros</env-entry-value>
         </env-entry>
         <env-entry>
            <env-entry-name>changeRateEntry</env-entry-name>
            <env-entry-type>java.lang.Float</env-entry-type>
            <env-entry-value>0.80</env-entry-value>
         </env-entry>
      </session>
   </enterprise-beans>
</ejb-jar>
```

Modèle de développement

```
@Stateless
public class ItemConverterEJB {
    @Resource(name = "currencyEntry")
    private String currency;

    @Resource(name = "changeRateEntry")
    private Float changeRate;

    public Item convertPrice(Item item) {
        item.setPrice(item.getPrice() * changeRate);
        item.setCurrency(currency);
        return item;
    }
}
```

Appels asynchrones

- Par défaut, les invocations de méthodes de session beans via l'interface local, remote ou sans interface son synchrones, c'est à dire que le client reste bloqué le temps du traitement, puis reçoit la réponse et peut continuer son travail
- Mais il est parfois utile de pouvoir faire des appels asynchrones, notamment pour les traitements longs, par exemple pour demander l'impression d'un document

Appels asynchrones

- Avant EJB 3.1, il était possible d'utiliser les Message Driven Beans et JMS pour faire des appels asychrones. Il fallait créer les objets JMS (factories et distinations), envoyer un message à la destination et développer un MDB pour traiter le message. Cela fonctionne mais reste un processus lourd si vous n'avez besoin que de faire un appel de méthode de façon asychrone

Appels asynchrones

- Depuis EJB 3.1, il est possible de faire un appel de méthode de façon asynchrone simplement en annotant une méthode avec @javax.ejb.Asynchronous

```
@Stateless
public class OrderEJB {
        @Asynchronous
        private void sendEmailOrderComplete(Order order, Customer customer) {
            // Send e-mail
        }
        @Asynchronous
        private void printOrder(Order order) {
            // Print order
        }
    }
```

Appels asynchrones

- On a parfois besoin que la méthode asynchrone renvoie un résultat :
  - Une méthode asynchrone peut retourner void ou java.util.concurrent.Future<V>, où V représentes le résultat
  - La classe Future permet d'obtenir le résultat d'une méthode exécutée dans un thread différent
  - Le client peut donc utiliser l'API Future pour obtenir le résultat ou même annuler la demande. On utilise généralement l'implémentation javax.ejb.AsyncResult<V>

Appels asynchrones

```
@Stateless
@Asynchronous
public class OrderEJB {
    @Resource
    SessionContext ctx;
    private void sendEmailOrderComplete(Order order, Customer customer) {
        // Send e-mail
    }
    private void printOrder(Order order) {
        // Print order
    }
    private Future<Integer> sendOrderToWorkflow(Order order) {
        Integer status = 0;
        // processing
        status = 1;
        if (ctx.wasCancelCalled()) {
            return new AsyncResult<Integer>(2);
        }
        // processing
        return new AsyncResult<Integer>(status);
    }
}
```

```
Future<Integer> status = orderEJB.sendOrderToWorkflow (order);
Integer statusValue = status.get();
```

Conteneur embarqué

- Le conteneur de sessions beans offre de nombreux services (transaction, cycle de vie, appels asynchrones, intercepteurs, ...), ce qui permet de se concentrer sur le code métier
- Le problème c'est du coup qu'il faut toujours un conteneur pour exécuter un EJB, même pour le tester
- Avant les EJB 3.1, il fallait ajouter une interface Remote, créer une façade TestEJB qui délègue les appels au bean à tester ou encore utiliser des fonctionnalités spécifiques au serveur

Conteneur embarqué

- Avec EJB 3.1, le problème a été résolu avec la création d'un conteneur embarqué
- EJB 3.1 fournit une API standard pour exécuter des EJBs depuis Java SE. Le package javax.ejb.embeddable permet à un client d'instancier un conteneur d'EJB qui tourne dans sa propre JVM
- Le conteneur embarqué fournit un environnement d'exécution managé qui supporte les mêmes services de base qui existent dans un serveur JEE : transactions, ...

Conteneur embarqué

- Le conteneur embarqué ne doit implémenter que le sous-ensemble d'API EJB Lite. On ne peut donc utiliser les EJB 2.x, les MDBs, ...
- La classe EJBContainer contient une factory method (createEJBContainer()) pour créer une instance du conteneur
- Par défaut, le conteneur recherche dans le classpath du client pour trouver les classes d'EJBs pour l'initialisation

Conteneur embarqué

```
public class ItemEJBTest {
    private static EJBContainer ec;
    private static Context ctx;

    @BeforeClass
    public static void initContainer() throws Exception {
        ec = EJBContainer.createEJBContainer();
        ctx = ec.getContext();
    }

    @AfterClass
    public static void closeContainer() throws Exception {
        ec.close();
    }
```

```
    @Test
    public void createBook() throws Exception {
        // Creates an instance of book
        Book book = new Book();
        book.setTitle("The Hitchhiker's Guide to the Galaxy");
        book.setPrice(12.5F);
        book.setDescription("Science fiction comedy book");
        book.setIsbn("1-84023-742-2");
        book.setNbOfPage(354);
        book.setIllustrations(false);

        // Looks up for the EJB
        ItemEJB bookEJB = (ItemEJB) ➥
            ctx.lookup("java:global/chapter07/ItemEJB");

        // Persists the book to the database
        book = itemEJB.createBook(book);
        assertNotNull("ID should not be null", book.getId());

        // Retrieves all the books from the database
        List<Book> books = itemEJB.findBooks();
        assertNotNull(books);
    }
}
```

Timer Service

- Il est parfois nécessaire dans des applications Java EE de plannifier des tâches à certains moments. Par exemple :
  - envoyer un email à l'anniversaire d'un client,
  - imprimer les statistiques mensuelles,
  - générer des rapports chaque nuit sur les niveaux de stocks,
  - rafraîchir un cache technique toutes les 30 secondes

Timer Service

- EJB 2.1 a introduit un timer service pour ce genre de besoin. Mais comparé à d'autres outils comme le cron Unix ou Quartz, il manquait beaucoup de fonctionnalités au timer service
- La version 3.1 de la spécification EJB est une très grosse amélioration qui s'est inspirée des outils disponibles pour offrir une solution qui est maintenant comparable pour la plupart des besoins

Timer Service

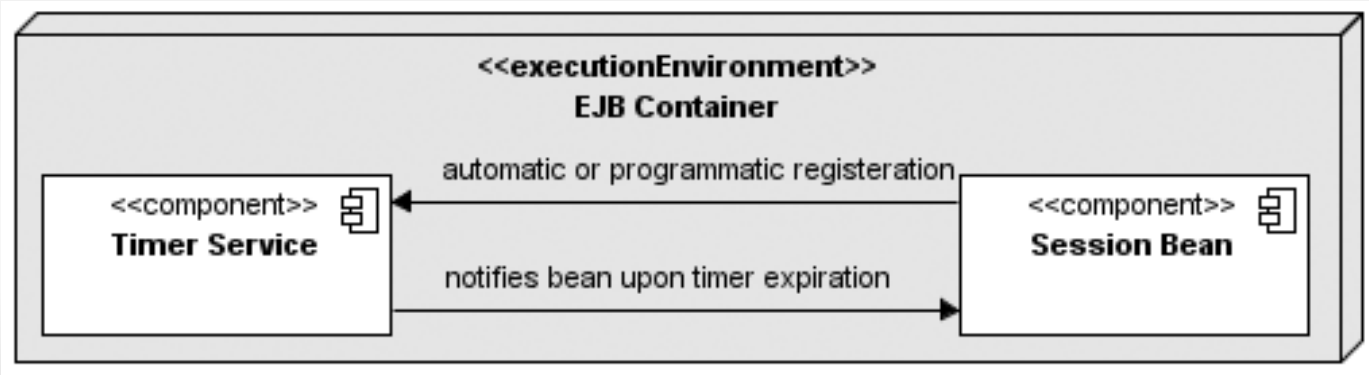
- Le timer service est un service du conteneur qui permet aux EJBs d'être enregistrés pour un appel ultérieur
- Les appels d'EJBs peuvent se faire :
  - à une date et une heure spécifiques,
  - après un certain délai,
  - à intervales réguliers
- Le conteneur retient tous les ”timers” et invoque la méthode de l'EJB quand le ”timer” a expiré

Timer Service

- Il faut donc commencer par créer un timer (automatiquement ou de façon programmative) et enregistrer l'EJB pour un appel ultérieur
- Les timers sont prévus pour des processus longs, donc ils sont par défaut persistants. On peut choisir de les rendre non persistants
- Les beans de type Stateless, Singleton et MDB peuvent être associés à un timer (pas Stateful)

Timer Service

- Les timers peuvent être créés automatiquement par le conteneur au moment du déploiement si l'EJB a une méthode annotée @Schedule
- Les timers peuvent aussi être créés programmativement et associer une méthode qui doit être annotée avec @Timeout



Expressions

Attribute	Description	Possible Values	Default Value
`second`	One or more seconds within a minute	[0,59]	0
`minute`	One or more minutes within an hour	[0,59]	0
`hour`	One or more hours within a day	[0,23]	0
`dayOfMonth`	One or more days within a month	[1,31] or {"1st", "2nd", "3rd", . . . , "30th", "31st"} or {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"} or "Last" (the last day of the month) or -x (means x day(s) before the last day of the month)	*
`month`	One or more months within a year	[1,12] or {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"}	*
`dayOfWeek`	One or more days within a week	[0,7] or {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}—"0" and "7" both refer to Sunday	*
`year`	A particular calendar year	A four-digit calendar year	*
`timezone`	A specific time zone	List of time zones as provided by the zoneinfo (or tz) database	

Expressions

Form	Description	Example
Single value	The attribute has only one possible value.	`year = "2009"` `month= "May"`
Wildcard	This form represents all possible values for a given attribute.	`second = "*"` `dayOfWeek = "*"`

List	The attribute has two or more values separated by a comma.	`year = "2008,2012,2016"`
		`dayOfWeek = "Sat,Sun"`
		`minute = "0-10,30,40"`

Range	The attribute has a range of values separated by a dash.	`second="1-10"` `dayOfWeek = "Mon-Fri"`
Increments	The attribute has a starting point and an interval separated by a forward slash.	`minute = "*/15"` `second = "30/10"`

Expressions

Example	Expression
Every Wednesday at midnight	`dayOfWeek="Wed"`
Every Wednesday at midnight	`second="0", minute="0", hour="0", dayOfMonth="*", month="*", dayOfWeek="Wed", year="*"`
Every weekday at 6:55	`minute="55", hour="6", dayOfWeek="Mon-Fri"`
Every weekday at 6:55 Paris time	`minute="55", hour="6", dayOfWeek="Mon-Fri", timezone="Europe/Paris"`
Every minute of every hour of every day	`minute="*", hour="*"`
Every second of every minute of every hour of every day	`second="*", minute="*", hour="*"`
Every Monday and Friday at 30 seconds past noon	`second="30", hour="12", dayOfWeek="Mon, Fri"`
Every five minutes within the hour	`minute="*/5", hour="*"`
Every five minutes within the hour	`minute="0,5,10,15,20,25,30,35, 40,45,50,55", hour="*"`
The last Monday of December at 3 p.m.	`hour="15", dayOfMonth="Last Mon", month="Dec"`
Three days before the last day of each month at 1 p.m.	`hour="13", dayOfMonth="-3"`
Every other hour within the day starting at noon on the second Tuesday of every month	`hour="12/2", dayOfMonth="2nd Tue"`
Every 14 minutes within the hour, for the hours of 1 and 2 a.m.	`minute = "*/14", hour="1,2"`
Every 14 minutes within the hour, for the hours of 1 and 2 a.m.	`minute = "0,14,28,42,56", hour = "1,2"`
Every 10 seconds within the minute, starting at second 30	`second = "30/10"`
Every 10 seconds within the minute, starting at second 30	`second = "30,40,50"`

Création automatique des timers

- Le conteneur crée un timer pour chaque méthode annotée par @javax.ejb.Schedule ou @Schedules (ou l'équivalent XML dans ejb-jar.xml)
- Par défaut les timers sont persistants, mais on peut demander le contraire

Création automatique des timers

```
@Stateless
public class StatisticsEJB {
    @Schedule(dayOfMonth = "1", hour = "5", minute = "30")
    public void statisticsItemsSold() {
        // ...
    }
    @Schedules({
            @Schedule(hour = "2"),
            @Schedule(hour = "14", dayOfWeek = "Wed")
    })
    public void generateReport() {
        // ...
    }
    @Schedule(minute = "*/10", hour = "*", persistent = false)
    public void refreshCache() {
        // ...
    }
}
```

Création programmative des timers

- Pour créer les timers, l'EJB a besoin d'accéder à l'interface javax.ejb.TimerService en utilisant l'injection de dépendance ou l'EJBContext (EJBContext.getTimerService()), ou encore en utilisant JNDI
- L'interface TimerService propose plusieurs méthodes de création de timers suivant les types de timers

Création programmative de timers

- createTimer : création de timers basés sur les dates, intervales et durées sans expressions
- createSingleActionTimer : crée un timer qui expire à une date précise ou après une certaine durée. Le timer est ensuite supprimé
- createIntervalTimer : crée un timer dont la première expiration est à une date précise et dont les suivantes sont à un intervale donné
- createCalendarTimer : crée un timer basée sur une expression calendaire avec la classe utilitaire ScheduleExpression

Création programmative de timers

```
new ScheduleExpression().dayOfMonth("Mon").month("Jan");
new ScheduleExpression().second("10,30,50").minute("*/5").hour("10-14");
new ScheduleExpression().dayOfWeek("1,5").timezone("Europe/Lisbon");
```

Création programmative de timers

- Les méthodes de TimerService (createSingleActionTimer, …) renvoient un objet de type Timer qui contient des informations sur la date de création, s'il est persistant, … Cet objet permet aussi d'annuler un timer avant son expiration
- Quand le timer expire, la méthode annotée par @Timeout est appelée et reçoit l'objet Timer
- Une seule méthode par bean peut être annotée par @Timeout

Création programmative de timers

```
@Stateless
public class CustomerEJB {
    @Resource
    TimerService timerService;

    @PersistenceContext(unitName = "chapter07PU")
    private EntityManager em;

    public void createCustomer(Customer customer) {
        em.persist(customer);
        ScheduleExpression birthDay = new ScheduleExpression().➥
                dayOfMonth(customer.getBirthDay()).month(customer.getBirthMonth());
        timerService.createCalendarTimer(birthDay, new TimerConfig(customer, true));
    }
    @Timeout
    public void sendBirthdayEmail(Timer timer) {
        Customer customer = (Customer) timer.getInfo();
        // ...
    }
}
```