7

Obtenir et manipuler le gestionnaire d'entités

La maîtrise des métadonnées est une chose, l'utilisation correcte d'une API en est une autre. Dès que vous sortez des exemples simples et que vous essayez d'utiliser Java Persistence dans une application quelque peu complexe, la gestion des instances de EntityManager, EntityTransaction dans un environnement SE et même parfois EntityManagerFactory et Ejb3Configuration devient cruciale.

Les questions qui viennent assez vite à l'esprit sont les suivantes :

- Où puis-je obtenir mon gestionnaire d'entité ?
- Quelle est sa durée de vie ?
- Y a-t-il des différences selon que mon application est exécutée dans un conteneur EJB ou dans un environnement Java SE ?

Ce chapitre se penche sur les concepts d'architecture autour de Java Persistence selon le type d'environnement. Nous aborderons un exemple d'utilisation de Java Persistence dans un batch qui nécessite un raisonnement légèrement différent. Enfin, nous énoncerons la liste des exceptions que vous pouvez rencontrer en utilisant Java Persistence.

Définitions et contextes de persistance

Utilisé tout au long des chapitres précédents, le gestionnaire d'entités est le point de jonction transparent entre votre application et la base de données.

Définitions

Un gestionnaire d'entités a une durée de vie relativement courte, en accord avec un traitement métier, et, surtout, est threadsafe. Cet ensemble de notions (gestionnaire d'entités + traitement métier ou cas d'utilisation) définit le contexte de persistance. Le gestionnaire d'entités représente un cache de premier niveau. Il contient un ensemble restreint d'entités et n'est accessible que par un thread.

Il ne faut pas considérer le gestionnaire d'entités comme un cache global. Si deux traitements, ou threads, parallèles venaient à utiliser un même gestionnaire d'entités, Java Persistence ne pourrait garantir les données qu'il contient et cela pourrait engendrer des corruptions de données.

L'EntityManagerFactory dispense les gestionnaires d'entités. Celle-ci est construite *via* une instance d'Ejb3Configuration qui est l'analyseur des métadonnées et qui restera transparent pour l'utilisateur. Une EntityManagerFactory est couplée à une base de données particulière et à un ensemble d'entités. Cela représente l'unité de persistance (*persistence unit*). Rappelez-vous que tant que le gestionnaire d'entités est valide et que vos entités y sont attachées, celles-ci sont surveillées et gérées par le gestionnaire d'entités. Par conséquent, une fois le contexte de persistance terminé, le gestionnaire d'entités n'est plus valide, et les entités sont donc automatiquement détachées. Les entités détachées peuvent être exploitées par un autre tiers, modifiées, puis revenir pour être mergées dans un nouveau contexte de persistance (souvenez-vous de la méthode `em.merge(detachedEntity)`).

Revenons au cycle de vie du gestionnaire d'entités. Nous avons dit que celui-ci était lié à un traitement métier finement défini (court), communément appelé unité de travail.

Contextes de persistance

Au niveau de votre conception, c'est la notion de contexte de persistance qui va être importante.

Il existe deux types de contextes de persistance : le contexte de persistance porté par la transaction et le contexte de persistance étendu.

Contexte de persistance porté par la transaction

Il s'agit de la première possibilité conceptuelle concernant le cycle de vie du contexte de persistance. Il peut vivre aussi longtemps qu'une transaction et se terminer lorsque la transaction s'achève.

Seuls les contextes de persistance gérés par un serveur d'applications peuvent être portés par la transaction. La transaction peut être démarrée par le conteneur si vous utilisez la démarcation déclarative (annotations), ou manuellement dans le cas d'une démarcation programmatique. Dans les deux cas, le conteneur EJB se charge d'associer le contexte de persistance, et donc le gestionnaire d'entités, à la transaction en cours.

Contexte de persistance étendu

Les contextes de persistance étendus peuvent être utilisés si vous devez les conserver sur plusieurs transactions. Pensez, par exemple, à une saisie sur plusieurs formulaires Web que vous souhaiteriez valider en toute dernière page. Cette notion est principalement utilisée pour implémenter une conversation. Nous effectuons une démonstration complète de ce type de contexte de persistance plus loin dans ce chapitre.

Ce type de contexte est aussi le seul type de contexte de persistance disponible pour les environnements autonomes.

Mise en œuvre des différentes possibilités

Nous allons décrire l'utilisation du gestionnaire d'entités selon divers environnements possibles.

Environnement autonome SE

L'environnement autonome est celui qui est le plus indépendant et surtout indépendant de la plate-forme entreprise. Ce qui signifie que vous n'avez accès à aucun service, ni JNDI, ni datasource, il vous faut gérer plus de choses.

Le premier point concerne la configuration. Jusque-là, grâce à JBoss intégré, nous disposions notamment d'une datasource. Il va nous falloir dans un premier temps reconfigurer notre application.

Packaging

Souvenez-vous qu'au chapitre 2 nous avons détaillé scrupuleusement l'arborescence à employer pour utiliser Java Persistence. Dans un environnement autonome, le packaging est sensiblement le même, sauf que vous n'avez besoin que des bibliothèques relatives à l'implémentation de Java Persistence (Hibernate, hibernate-annotations, etc.). Celles dédiées à JBoss intégré peuvent donc être supprimées.

Le point clé reste la présence de META-INF/persistence.xml dans votre classpath.

Il va vous falloir configurer le pool de connexions manuellement, étant donné que vous n'avez plus de datasource à disposition. Pour cela, référez-vous aux tableaux présentés au chapitre 2, ainsi qu'au descriptif relatif aux pools de connexions abordé au chapitre 10.

L'exemple suivant est l'écriture la plus simple possible :

```
<persistence>
  <persistence-unit name="eyrollesEntityManager">
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.HSQLDialect"/>
```

```
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.connection.driver_class"
        value="org.hsqldb.jdbcDriver"/>
      <property name="hibernate.connection.url"
        value="jdbc:hsqldb:."/>
      <property name="hibernate.connection.user" value="sa"/>
    </properties>
  </persistence-unit>
</persistence>
```

Exploiter un fichier *Hibernate.cfg.xml*

Il est possible d'exploiter un fichier de configuration globale Hibernate en paramétrant :

```
<property name="hibernate.ejb.cfgfile" value="/hibernate.cfg.xml"/>
```

EntityManagerFactory

Par définition, dans un environnement autonome, aucun conteneur ne peut nous aider. Il nous faut donc initialiser Java Persistence manuellement. Cela passe par la création de l'EntityManagerFactory, sans laquelle nous ne pouvons obtenir de gestionnaire d'entités.

L'EntityManagerFactory est lourd à créer, demande beaucoup de ressources et peut être accédé par plusieurs threads. Une variable *static* dans une classe utilitaire est un bon moyen de gérer cet aspect :

```
public class JavaPersistenceUtil {
private static EntityManagerFactory emf;
  static {
    emf =
Persistence.createEntityManagerFactory("eyrollesEntityManager");
  }

  public static EntityManagerFactory getEmf() {
    return emf;
  }

}
```

La méthode `createEntityManagerFactory(«name»)` de la classe `javax.persistence.Persistence` prend en argument de nom de l'unité de persistance que vous souhaitez initialiser. Nous l'invoquons ici dans un bloc statique, qui ne sera exécuté qu'une fois.

En début de projet, posez-vous la question de savoir si vous aurez besoin de plusieurs unités de persistance, auquel cas modifiez votre classe utilitaire pour qu'elle puisse gérer un pool de connexions à *n* base de données.

Vous avez désormais un accès aisé à l'EntityManagerFactory et pouvez obtenir un gestionnaire d'entités à tout moment grâce à :

```
EntityManager em = JavaPersistenceUtil
  .getEmf().createEntityManager();
```

Transaction

Dans un environnement autonome, la gestion des transactions est effectuée par l'application. L'interface EntityTransaction vous permet de démarrer et achever la transaction à laquelle le gestionnaire d'entités prend part. Dans un environnement avec conteneur, l'EntityTransaction est directement liée à la UserTransaction de JTA.

Complétons notre exemple de code avec la gestion des transactions :

```
public void test() throws Exception
{
  EntityManager em =
    JavaPersistenceUtil.getEmf().createEntityManager();

  EntityTransaction tx = em.getTransaction();

  tx.begin();
  Team team = new Team("cascade test team");
  Player player = new Player ("cascade player test");
  em.persist(team);
  tx.commit();
  em.close();
}
```

Notez la présence de l'appel à `em.close()`. Nous n'avons jamais vu cette ligne dans les chapitres précédents, car JBoss Intégré se chargeait seul de fermer le gestionnaire d'entités.

Un dernier aspect, des plus contraignants dans un environnement autonome, est la gestion des exceptions.

Gestion des exceptions

Dans un environnement avec conteneur, la gestion des ressources est optimisée, comme nous le verrons plus tard avec les exemples dédiés au beans session. Ce n'est pas le cas dans un environnement autonome ou lorsque vous gérez Java Persistence manuellement. En cas d'exceptions, il est indispensable de vous assurer que le code relâche les ressources.

Notre code se complexifie donc substantiellement :

```
public void test() throws Exception
{
  EntityManager em = null;

  EntityTransaction tx = null;
  try{
    em =  JavaPersistenceUtil.getEmf().createEntityManager();
    tx =  em.getTransaction();
    tx.begin();
    Team team = new Team("cascade test team");
    Player player = new Player ("cascade player test");
    School school = new School ("cascade school test");
```

```
      Coach coach= new Coach ("cascade test coach");
      player.setSchool(school);
      team.getPlayers().add(player);
      team.setCoach(coach);
      em.persist(team);
      tx.commit();
    } catch (RuntimeException ex) {
      try {
        tx.rollback();
      } catch (RuntimeException rbEx) {
      }
        throw ex;
      } finally {
        em.close();
      }
  }
```

La différence de taille, par rapport aux exemples que nous avons abordés jusque-là avec JBoss intégré, est que le code a en charge l'initialisation de Java Persistence (de l'Entity-ManagerFactory). Pour ce qui est des exceptions, nous y reviendrons très rapidement.

Environnement entreprise EE

Que ce soit *via* le conteneur léger JBoss intégré ou l'utilisation d'un serveur d'applications complet, comme JBoss Application Server, les services offerts par la plate-forme Java EE apportent énormément en termes de fonctionnalités, de robustesse et de simplification de code.

JBoss intégré permet d'aborder les services qui impactent directement et favorablement l'utilisation de Java Persistence, à savoir JNDI, JTA et ce que nous pouvons appeler de manière générique les initialiseurs de services.

Au-delà de ces services, la gestion déclarative des transactions et des contextes de persistance simplifie encore d'un niveau votre code.

Packaging

Le packaging est celui normalisé développé au chapitre 2. Le point important est la déclaration de la datasource JTA :

```
<persistence>
  <persistence-unit name="eyrollesEntityManager">
    <jta-data-source>java:/myDS</jta-data-source>
      <properties>
         <property name="hibernate.hbm2ddl.auto"
            value="create-drop"/>
         <property name="hibernate.dialect"
            value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.show_sql" value="true"/>
         <property name="jboss.entity.manager.jndi.name"
```

```
            value="java:/EntityManagers/eyrollesEntityManager"/>
      </properties>
   </persistence-unit>
</persistence>
```

Grâce à cette configuration, le conteneur exploite son gestionnaire de ressources et y inclut la datasource exploitée par Java Persistence.

Exploitation minimale de l'environnement

Cette exploitation consiste à ne tirer profit que de l'initialisation transparente de l'EntityManagerFactory et à l'exploitation de JNDI et JTA. C'est ce que nous avons fait dans les chapitres précédents en utilisant JBoss intégré et JUnit :

```
EntityManager em = (EntityManager) new InitialContext()
  .lookup("java:/EntityManagers/eyrollesEntityManager");

TransactionManager tm = (TransactionManager) new InitialContext()
  .lookup("java:/TransactionManager");

  tm.begin();

  Team team = new Team("cascade test team");
  Player player = new Player ("cascade player test");
  …
  em.persist(team);
  tm.commit();
```

Ce code souffre d'un défaut de gestion d'exception. Nous devons appliquer la même attention que dans notre environnement SE :

```
EntityManager em = null;

TransactionManager tm = null;

try{
  em = (EntityManager) new InitialContext()
    .lookup("java:/EntityManagers/eyrollesEntityManager");

  tm = (TransactionManager) new InitialContext()
    .lookup("java:/TransactionManager");
  tm.begin();
  Team team = new Team("cascade test team");
  Player player = new Player ("cascade player test");
  …
  em.persist(team);
  tm.commit();
} catch (RuntimeException ex) {
  try {
    tm.rollback();
  } catch (RuntimeException rbEx) {
  }
```

```
  throw ex;
}
```

Par rapport à l'environnement autonome, les différences principales sont les suivantes :

- Nous déléguons au conteneur la responsabilité de démarrer Java Persistence.
- Nous exploitons JTA et non plus les transactions JDBC (encapsulées dans EntityTransaction) *via* `em.getTransaction()`, le conteneur faisant le lien de manière transparente entre le gestionnaire d'entités, la datasource et JTA.
- Nous n'avons plus à nous soucier de fermer le gestionnaire d'entités dans un block *finally* : c'est fait automatiquement par le conteneur.

Ce sont des différences de taille pour l'architecture technique de votre application mais qui n'ont que peu d'impact sur le code.

Tester un bean session depuis JUnit

Avant de vous pencher sur l'utilisation des beans session, voici le code intégré aux tests JUnit pour tester les beans session. Tout d'abord, dans le cas d'une utilisation de JBoss intégré, pensez à compléter la méthode `deploy()` pour déployer les classes et interfaces composant votre couche service à base de beans session (ici `TeamManager` et `TeamManagerBmt`) :

```
public static void deploy()
{
  jar = AssembledContextFactory
    .getInstance().create("Ch7TestCase.jar");

  jar.addClass(Team.class);
  jar.addClass(Coach.class);
  jar.addClass(Player.class);
  jar.addClass(School.class);
  jar.addClass(TeamManager.class);
  jar.addClass(TeamManagerBeanBmt.class);
  jar.addResource("myDS-ds.xml");
  jar.mkdir("META-INF")
    .addResource("eyrolles-persistence.xml", "persistence.xml");

  try{
    Bootstrap.getInstance().deploy(jar);
  }
  catch (DeploymentException e) {
    throw new RuntimeException("Unable to deploy", e);
  }
}
```

Les méthodes tests en elles-mêmes suivent le modèle suivant. On consulte le registre JNDI pour exploiter les beans session et on invoque les méthodes que l'on souhaite tester :

```
public void testSessionBeanBmt() throws Exception{
  InitialContext ctx = new InitialContext();
  TeamManager teamManager = (TeamManager) ctx
    .lookup("TeamManagerBeanBmt/local");

List<Team> teams = teamManager.getAllTeams();
}
```

Beans Session et BMT

Nous allons désormais utiliser des beans session que nous testerons *via* JUnit grâce à la méthode évoquée précédemment.

Imaginons une interface de services centrés sur l'entité `Team`, avec une première méthode de recherche ainsi qu'une récupération par identifiant :

```
@Local
public interface TeamManager
{
  List<Team> getAllTeams() ;
  Team getTeam(Integer id);
}
```

L'objet de ce livre n'étant pas la spécification EJB3 dans son intégralité, nous ne détaillerons que le minimum concernant les beans session.

Dans notre exemple, l'interface métier qu'implémentera notre bean session est locale. Elle est annotée avec @Local.

Voyons désormais notre bean session à proprement parler :

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class TeamManagerBeanBmt implements TeamManager {

  @Resource
  UserTransaction ut;

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

  public List<Team> getAllTeams() {
    List<Team> result = null;
    try{
      ut.begin();
      result = em
        .createQuery("select team from Team team")
        .getResultList();
      ut.commit();
    }
    catch (Exception ex) {
      try {
```

```
        ut.rollback();
      }
      catch (Exception rbEx) {
        rbEx.printStackTrace();
      }
      ex.printStackTrace();
    }
    return result;
  }

  public Team getTeam(Integer id) {
    …
  }
}
```

Tout d'abord, notre bean session est sans état. Nous l'annotons donc avec @Stateless. Nous gérons les transactions manuellement au niveau du bean (Bean Managed Transaction, ou BMT), ce qui explique l'annotation @TransactionManagement(TransactionManagementType.BEAN). L'annotation @TransactionManagement accepte deux valeurs pour son membre :

- `TransactionManagementType.BEAN`, pour les BMT.
- `TransactionManagementType.CONTAINER`, pour les CMT, si l'on souhaite déléguer la gestion des transactions au conteneur après avoir opéré une gestion déclarative des transactions, que nous verrons plus tard.

Viennent ensuite deux déclarations cruciales, qui apportent énormément. Il s'agit de la déclaration de variable `UserTransaction ut` (transaction JTA) annotée avec @Resource et de la déclaration d'EntityManager em annotée avec @PersistenceContext(unitName="eyrollesEntityManager"). Ces deux déclarations permettent au conteneur d'injecter la UserTransaction et le gestionnaire d'entités. Elles remplacent l'interrogation manuelle du registre JNDI que nous faisions jusque-là dans nos méthodes JUnit.

L'implémentation de la méthode `getAllTeams()` suit le même schéma que l'exemple précédent. Nous y retrouvons la lourdeur de gestion des exceptions, nécessaire à la robustesse de votre application.

Bean Session et CMT

Dans ce cas de figure, le conteneur gère les transactions. En cas de problème, il effectue un rollback automatique sur la transaction, ce qui simplifie grandement le code :

```
@Stateless
public class TeamManagerBeanCmt implements TeamManager {

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

  public List<Team> getAllTeams() {
    List<Team> result = null;
```

```
    result = em
      .createQuery("select team from Team team")
      .getResultList();
    return result;
  }

  public Team getTeam(Integer id) {
    Team result = null;
    result = em.find(Team.class,id);
    return result;
  }

}
```

En l'absence d'annotation @TransactionManagement, le type de gestion par défaut est `TransactionManagementType.CONTAINER`. Ainsi :

```
@Stateless
public class TeamManagerBeanCmt implements TeamManager {…}
```

équivaut à :

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class TeamManagerBeanCmt implements TeamManager {…}
```

La déclaration de la UserTransaction a disparu. Les transactions étant gérées par le conteneur, nous n'en avons plus besoin ; elles deviennent totalement transparentes.

De même, en l'absence de déclaration de transaction, le comportement équivaut à :

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public List<Team> getAllTeaOms() {…}
```

@TransactionAttribute et TransactionAttributeType

Les différents comportements transactionnels que vous pouvez déclarer sont les suivants :

- `TransactionAttributeType.REQUIRED` : il s'agit de la valeur par défaut. Une méthode doit être invoquée avec un contexte transactionnel. Si ce contexte n'existe pas, le conteneur entame une transaction et y inclut toutes les ressources nécessaires à l'exécution de la méthode. Si cette méthode invoque d'autres composants transactionnels, le contexte transactionnel est propagé. Le conteneur commit la transaction au retour de la méthode, avant que le résultat soit envoyé au client.
- `TransactionAttributeType.NOT_SUPPORTED` : si le client qui invoque la méthode possède un contexte transactionnel, le conteneur suspend ce contexte et le réactive au retour de la méthode. Si ce contexte n'existe pas, aucune transaction n'est entamée. Les ressources utilisées par la méthode ne sont donc pas incluses dans une transaction, et l'autocommit est utilisé.

> **(Suite)**
>
> - `TransactionAttributeType.SUPPORTS` : si le client qui invoque la méthode possède un contexte transactionnel, elle se joint à ce contexte et adopte le même comportement que required. Sinon, c'est le comportement de not_supported qui s'applique. Ce type est rarement utilisé.
> - `TransactionAttributeType.REQUIRES_NEW` : la méthode est toujours exécutée au sein d'un nouveau contexte transactionnel, avec les mêmes conséquences et comportements que required. Si le client qui invoque la méthode possède un contexte transactionnel, le conteneur suspend ce contexte et le réactive au retour de la méthode.
> - `TransactionAttributeType.MANDATORY` : une méthode doit s'inscrire dans le contexte transactionnel du client qui l'invoque. Elle joint alors ce contexte et peut le propager davantage au besoin. S'il n'y a pas de contexte transactionnel, une exception est soulevée.
> - `TransactionAttributeType.NEVER` : c'est l'inverse de mandatory ; une exception est soulevée si le client qui invoque la méthode possède un contexte transactionnel.

Enfin, la gestion des exceptions et surtout leur impact sur la transaction en cours sont automatiquement gérés par le conteneur, celui-ci effectuant un rollback en cas d'exception non applicative (exception n'étant pas annotée avec @ApplicationException) ou en cas d'exception d'application marquée avec le membre rollback = true. Vous pouvez en savoir plus sur la définition d'une exception applicative dans la spécification EJB 3.0. Il s'agit d'un sujet complexe, qu'il est difficile de synthétiser.

En résumé

La plate-forme EE montre ici toute sa puissance tant par rapport à la simplicité du code qu'à la robustesse de la gestion des ressources par le conteneur.

En très peu de lignes de code, vous montez une application qui exploite des sources de données, gère les exceptions, les contextes transactionnels, ce qui n'est pas le cas lorsque vous travaillez avec Java SE. La section suivante, axée sur les conversations, va illustrer un autre avantage qu'a la plate-forme EE sur la plate-forme SE.

Conversation

Une conversation ou transaction applicative est un ensemble d'opérations courtes réalisées par l'interaction entre l'utilisateur et l'application, ce qui est différent de la transaction de base de données ou autre ressource transactionnelle dont nous sommes coutumiers. Dans les applications Web, par exemple, vous pouvez avoir besoin de plusieurs écrans pour couvrir un cas d'utilisation. Il s'agit là d'un cas typique de conversation.

Illustrons le concept de conversation avec notre application exemple de gestion d'équipes sportives, en reprenant les principales entités que nous avons détaillées au cours des chapitres précédents *(voir figure 7.1)* et en rendant l'association entre `Coach` et `Team` bidirectionnelle.

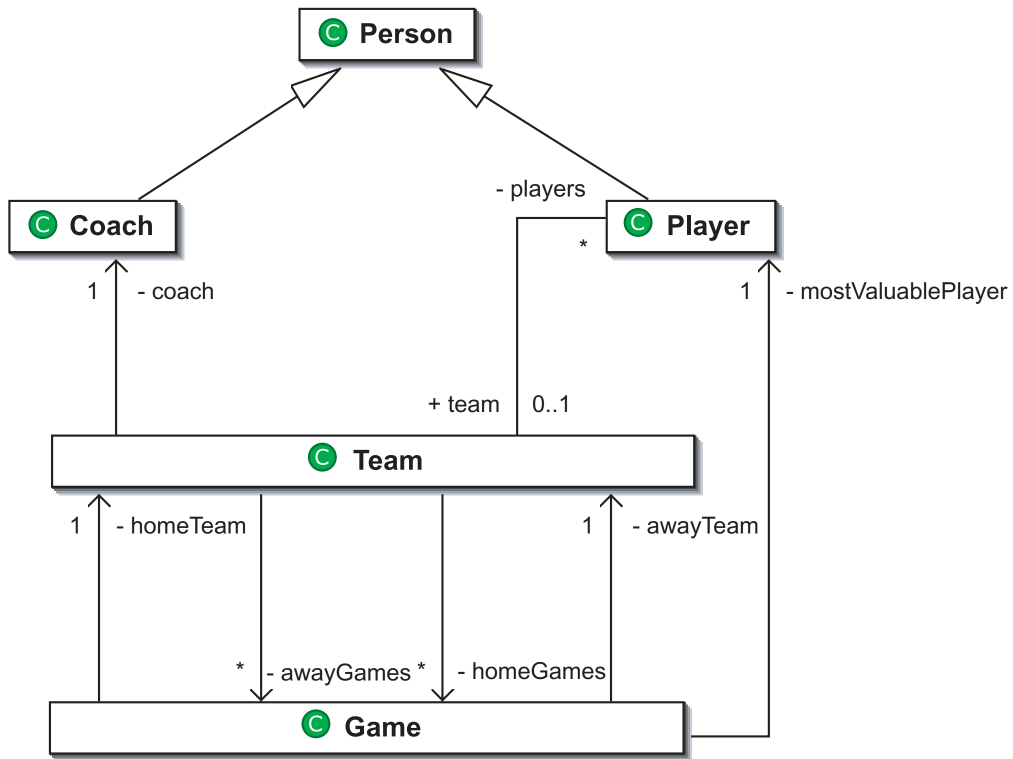


Figure 7-1
Modèle métier

La modification d'une équipe (instance de `Team`) pourrait s'effectuer comme illustré à la figure 7.2.

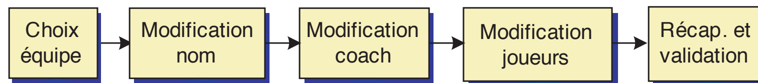


Figure 7-2
Étapes de la conversation

L'action qui marque le début de la conversation consiste en la sélection de l'équipe à modifier. Il n'y a rien de particulier à dire sur cette action, si ce n'est que la récupération de l'identifiant va nous permettre de faire un `em.find(Team.class,id)` :

```
public Team getTeam(Integer id) {
  Team result = em.find(Team.class,id);
  return result;
}
```

Un simple lien hypertexte permet d'envoyer l'information sur l'identifiant au serveur, comme le montre la figure 7.3.

Figure 7-3
Choix de l'équipe à modifier

La première étape, dite de modification, porte sur le nom de l'équipe. Le nom courant est renseigné dans le champ dédié (voir figure 7.4), et l'utilisateur peut choisir de le modifier.

À partir de ce moment, il est intéressant de se poser la question de ce que « contient » la vue. Est-ce l'objet persistant, une copie (sous forme de Data Transfer Object) ou un objet détaché ?

Figure 7-4
Modification du nom de l'équipe

Lors de la soumission du formulaire, l'objet soumis à la conversation possède ainsi un nom potentiellement modifié.

La deuxième étape propose la modification du coach *(voir figure 7.5)*. Il est nécessaire d'afficher la liste des coachs sans équipe, comme le montre la méthode suivante, ainsi que le coach courant :

```
public List<Coach> getFreeCoachs() {
  List<Coach> results = em
    .createQuery("from Coach c where c.team is null")
    .getResultList();
  return results;
}
```

Devons-nous recharger notre équipe et stocker les modifications déjà effectuées à un endroit précis ou réutiliser l'instance chargée lors de la première étape ? Qu'en est-il des associations non chargées ? Quel est l'état du gestionnaire d'entités à cet instant ? Comme nous le voyons, les questions s'accumulent.

Figure 7-5
Modification du coach

Lors de la soumission du formulaire, l'association entre l'équipe et son coach peut être modifiée. Cela engendre trois conséquences :

- L'équipe est associée au nouveau coach choisi.
- Le nouveau coach est associé à l'équipe.
- L'ancien coach n'a plus d'équipe.

Plaçons toute cette logique dans notre setter `team.setCoach()` :

```
public void setCoach(Coach c) {
  // pas d'ancien coach
  if (getCoach() == null && c != null){
    this.coach = c;
    c.setTeam(this);
  }
  else if (getCoach() != null && c == null){
    getCoach().setTeam(null);
    this.coach = c;
  }
  else if (getCoach() != null && c != null){
    if (!(getCoach().equals(c))){
      getCoach().setTeam(null);
      c.setTeam(this);
      this.coach = c;
    }
  }
}
```

L'avant-dernière étape traite des joueurs. Une fois encore nous devons précharger les joueurs libres, comme l'indique la méthode suivante :

```
public List<Player> getFreePlayers() {
  List<Player> results = em
    .createQuery("from Player p where p.team is null")
    .getResultList();
return results;
}
```

Il nous faut en outre présélectionner les joueurs courants *(voir figure 7.6)*. Ne nous attardons pas sur la suppression des doublons, car ce n'est pas notre préoccupation ici. Les mêmes questions que celles soulevées au sujet du coach apparaissent.

Figure 7-6
Modification des joueurs

La méthode `team.addPlayer(Player p)` garantit la cohérence de nos instances en gérant les deux extrémités de l'association :

```
public void addPlayer(Player p){
  if (p.getTeam() != null)
    p.getTeam().getPlayers().remove(p);
  p.setTeam(this);
  this.getPlayers().add(p);
  // vous pouvez encore améliorer ce code et le rendre plus robuste
  // avec des exceptions applicatives par exemple
}
```

La dernière étape affiche un résumé de l'équipe, avec les modifications saisies au cours des étapes précédentes *(voir figure 7.7)*. Lors de la validation, la totalité des modifications doit être rendue persistante.

Figure 7-7
Récapitulatif de l'équipe avant validation

Nous allons décrire deux moyens de gérer une telle situation. Avant cela, il est primordial de comprendre comment la base de données et le gestionnaire d'entités se synchronisent entre eux.

Synchronisation entre le gestionnaire d'entités et la base de données : flush

Lorsque le gestionnaire d'entités le nécessite, l'état de la base de données est synchronisé avec l'état des objets gérés par le gestionnaire d'entités en mémoire. Ce mécanisme, appelé flush, peut être paramétré selon un FlushModeType.

Il serait en effet préjudiciable pour les performances que chaque modification sur un objet soit propagée en base de données en temps réel et au fil de l'eau. Il est de loin préférable de regrouper les modifications et de les exécuter aux bons moments.

Selon la spécification Java Persistence, il existe deux FlushModeType :

- `FlushModeType.AUTO` (défaut) : Le flush est effectué au commit et avant l'exécution de certaines requêtes afin de garantir la validité du résultat.
- `FlushModeType.COMMIT` : Le flush est effectué au commit de la transaction.

Le FlushMode peut être modifié sur le gestionnaire d'entités (`em.setFlushMode(FlushModeType.X)`) ou sur une Query particulière (`query.setFlushMode(FlushModeType.X)`).

> **Classe *org.hibernate.flushMode* et automatismes pour le flush d'Hibernate**
>
> Hibernate propose davantage de possibilités. Voici les différents modes de synchronisation entre la session Hibernate et la base de données (le `flushMode.MANUAL` est primordial pour la suite de nos démonstrations) :
>
> - `flushMode.COMMIT`. Le flush est effectué au commit de la transaction.
> - `flushMode.AUTO` (défaut). Le flush est effectué au commit et avant l'exécution de certaines requêtes afin de garantir la validité du résultat.
> - `flushMode.ALWAYS`. La synchronisation se fait avant chaque exécution de requête. Ce mode pouvant affecter les performances, il est déconseillé de changer le FlushMode sans raison valable. Seul le `flushMode.AUTO` garantit de ne pas récupérer dans les résultats de requête des données obsolètes par rapport à l'état de la session.
> - `flushMode.MANUAL`. La base de données n'est pas automatiquement synchronisée avec la session Hibernate. Pour la synchroniser, il faut appeler explicitement `session.flush()`.

Il est important de bien comprendre les conséquences du flush, surtout pendant les phases de développement, car le débogage en dépend. En effet, vous ne voyez les ordres SQL qu'à l'appel du flush, et les exceptions potentielles peuvent n'être levées qu'à ce moment. Il est utile de rappeler que les ordres SQL s'exécutent au sein d'une transaction et que les résultats ne sont visibles de l'extérieur qu'au commit de cette transaction.

En d'autres termes, même si des update, delete et insert sont visibles sur les traces, les modifications ne sont pas consultables par votre client de base de données. Il faut pour cela attendre le commit de la transaction.

Prenons un exemple :

```
Player player = new Player();←❶
tm.begin();
em.persist(player);←❷
player.setName("zidane");←❸
//em.flush();←❹
Query q = em.createQuery("select coach from Coach coach");
//Query q = em.createQuery("select player from Player player");
List results = q.getResultList();←❺
tm.commit();←❻
```

La première ligne instancie une entité. La ligne ❷ rend l'instance persistante. À partir de ce moment, toute modification de l'objet est enregistrée par le gestionnaire d'entités.

Nous apportons une modification (repère ❸), effectuons une requête sur la classe (repère ❺) puis validons la transaction (repère ❻).

Nous sommes conscients que la modification apportée sur l'instance de `Player` pourrait avoir un impact sur le résultat de la requête si celle-ci cible les entités de type `Player`. Vérifions cela en effectuant tout d'abord une requête sur un autre type d'entité `Coach`.

Voici les logs provoqués par le code précédent :

```
insert into Player (id, name, height, team_id) values (null, ?, ?, ?)
call identity()
select coach0_.id as id1_, coach0_.name as name1_ from Coach coach0_
update Player set name=?, height=?, team_id=? where id=?
```

Nous voyons que le flush est exécuté en toute fin, au commit de la transaction.

Effectuons le même test, avec cette fois-ci une requête ciblant le type `Player` :

```
insert into Player (id, name, height, team_id) values (null, ?, ?, ?)
call identity()
update Player set name=?, height=?, team_id=? where id=?
select player0_.id as id2_, player0_.name as name2_, player0_.height as
height2_, player0_.team_id as team4_2_ from Player player0_
```

Ici, le flush intervient juste avant l'exécution de la requête.

Nous constatons que les impacts sur les ordres SQL sont sensibles à l'exécution d'une requête et au commit, ce qui n'a rien pour nous surprendre. Pendant les phases de développement, vous pouvez activer la ligne ❹ pour vérifier et déboguer.

Souvenez-vous que `FlushModeType.AUTO` invoque le flush au commit et lors de l'exécution de certaines requêtes.

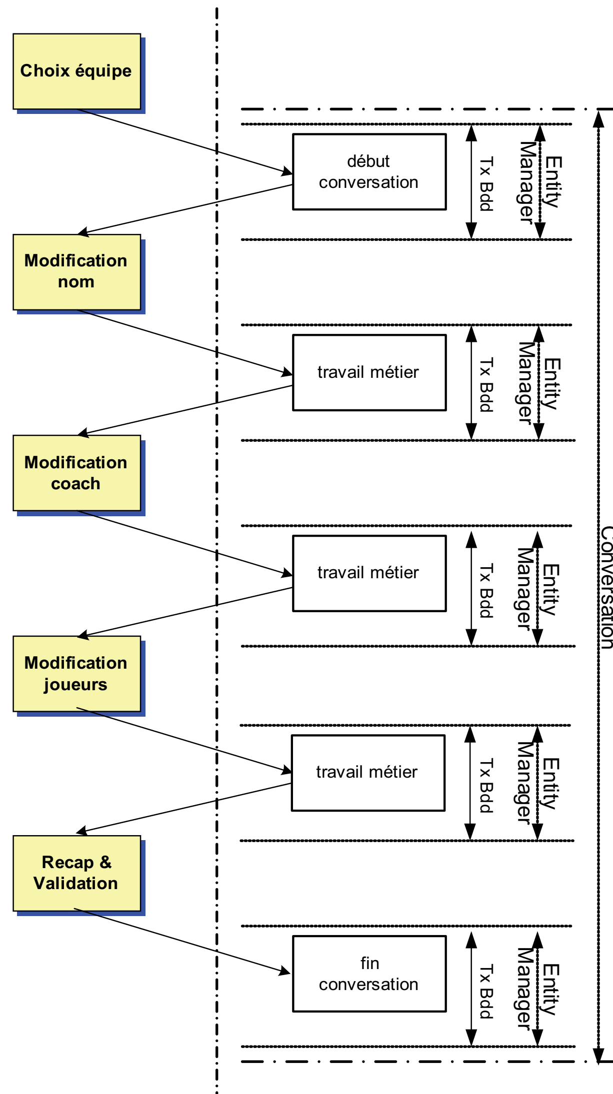
La notion de flush est primordiale pour la suite.

Gestionnaires d'entités multiples et objets détachés

Le premier moyen de gérer une conversation consiste à utiliser un nouveau gestionnaire d'entités à chaque étape. Les objets chargés à chaque étape sont donc détachés et doivent être mergés d'étape en étape si nécessaire, c'est-à-dire si la couche de persistance est utilisée.

La figure 7.8 illustre ce principe.

Figure 7-8
Gestionnaires d'entités multiples par conversation



Voici le code client simulant cette conversation :

```
public void testConversation() throws Exception
{
  initDatas();
  InitialContext ctx = new InitialContext();
  ConversationDemo teamManager = (ConversationDemo) ctx
    .lookup("ConversationDemoDetached/local");
```

```
    // étape zéro, on récupère toutes les équipes
    // pour faire notre choix
    List<Team> teams = teamManager.getAllTeams();

    //début de la conversation, on démarre de l'id
    //de l'équipe à modifier
    int teamToModifyId = teams.get(0).getId();

    // étape 1 : récupération de l'équipe
    Team teamToModify = teamManager.getTeam(teamToModifyId);

    // étape 2 : modification du nom
    teamToModify.setName("new name");

    // étape 3 : modification du coach
    List<Coach> freeCoachs = teamManager.getFreeCoachs();
    Coach previousCoach = teamToModify.getCoach();🡐❶
    previousCoach.setName("une petite modification");
    Coach nextCoach = freeCoachs.get(0);
    nextCoach.setName("I'm your new coach");
    teamToModify.setCoach(nextCoach);

    // étape 4 : modfication de l'effectif jouers
    List<Player> freePlayers = teamManager.getFreePlayers();
    teamToModify.addPlayer(freePlayers.get(0)); 🡐❷

    // étape finale : validation
    teamManager.validateModification(teamToModify);
}
```

Souvenez-vous que le but est de mettre à jour la base de données (insert, update, delete) uniquement en fin de conversation. Les transactions de chacune des étapes apparaissant sur la figure, excepté la dernière, et ne sont donc autorisées qu'à effectuer des lectures (select). Ces lectures pourraient être utiles si nous souhaitions initialiser des associations non chargées, configurées en lazy. Dans notre exemple, cela pourrait être utile pour éviter qu'une exception ne soit soulevée en ❶ et ❷.

Continuons dans ce raisonnement : pour initialiser des associations non chargées, il faudrait merger ou réattacher les instances racines (ici `teamToModify`) possédant de telles associations.

Réattacher une entité est possible avec les API spécifiques d'Hibernate (`session.update()`, `session.lock()`) mais n'est pas possible avec Java Persistence. En effet, la méthode `merge()` ne réattache pas mais renvoie une copie attachée. Cela change considérablement les signatures de vos méthodes.

Même dans notre cas très simple, si nous tentions une invocation de `merge()` avant les lignes ❶ et ❷, un update serait déclenché lors de son invocation puisque nous avons déjà modifié le nom de l'équipe.

Avec Java Persistence, la solution fiable est donc de précharger le réseau d'entités dès la première étape. Pour cela, remplacez :

```
public Team getTeam(Integer id) {
  Team result = em.find(Team.class,id);
  return result;
}
```

par :

```
public Team getTeam(Integer id) {
  StringBuffer queryString = new StringBuffer();
  queryString.append("select team from Team team ")
    .append("left join fetch team.coach ")
    .append("left join fetch team.players ")
    .append("where team.id = :theId");
  Query q = em.createQuery(queryString.toString());
  q.setParameter("theId", id);
  Team result = (Team)q.getSingleResult();
  return result;
}
```

L'étape finale est relativement simple puisqu'une ligne suffit à rendre persistantes toutes les modifications. N'oubliez pas d'activer `cascade=CascadeType.MERGE` sur les associations concernées :

```
teamManager.validateModification(teamToModify);
```

qui invoque :

```
public void validateModification(Team team) {
  em.merge(team);
}
```

Dans le cas présent, le préchargement du graphe d'objets est facile. Pour des cas d'utilisation plus complexes et des graphes d'objets plus lourds, il devient délicat de prévoir de manière efficace ce qu'il faut charger. Nous risquons d'aboutir soit à un chargement trop large, et donc pénalisant pour les performances, soit à un chargement trop restreint, et donc à des risques de LazyInitialisationException difficiles à maîtriser et à localiser. Souvenez-vous toutefois qu'en utilisant directement les API d'Hibernate, vous pouvez réattacher les entités pour charger de telles associations, même si cela complexifie davantage le code.

Nous voyons donc que l'utilisation d'entités détachées est vite source de complications et est souvent synonyme de casse-tête.

Mise en place d'un contexte de persistance étendu

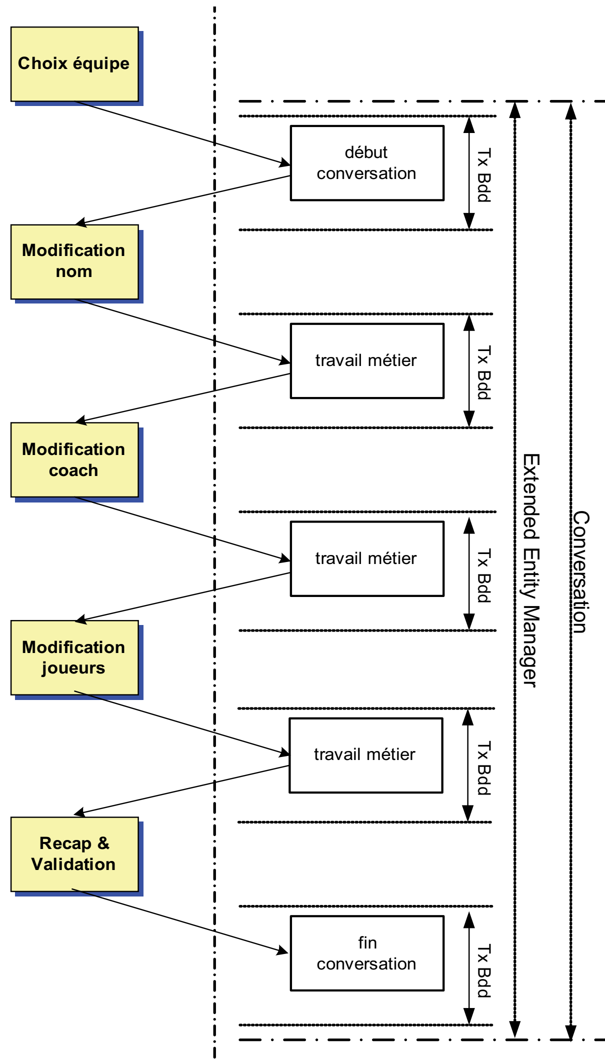
Une autre manière de procéder pour traiter une conversation consiste à garder le gestionnaire d'entités en vie pendant les cinq étapes. Cette méthode est aussi appelée *mise en place d'un contexte de persistance étendu*.

Nous allons en faire la démonstration dans un environnement EE à base d'EJB session.

La figure 7.9 illustre la possibilité d'exploiter un seul gestionnaire d'entités.

Figure 7-9

Gestionnaire d'entités étendu et conversation



Notez qu'il est communément plus correct de parler de contexte de persistance étendu plutôt que de gestionnaire d'entités étendu, comme à la figure 7.9.

La conversation sera composée de *n* transactions avec la base de données, une par étape, chaque étape pouvant être traitée par une HTTPRequest par exemple. Étant donné la durée de vie du gestio nnaire d'entités, nous ne pouvons nous contenter d'un contexte de persistance couplé au contexte transactionnel. Il nous faut donc l'étendre, et nous utiliserons un contexte de persistance étendu.

La notion de conversation demande de ne valider les changements qu'à la validation (dernier écran). Il faut qu'aucune mise à jour dans la base de données n'ait lieu pendant les étapes précédant la validation.

En résumé, nous avons :

- Une ouverture de gestionnaire d'entités en début de conversation.
- Une transaction en lecture si nécessaire pour les étapes intermédiaires.
- Aucun update, delete ou insert n'est « généré » avant la fin de la conversation. C'est là la fonctionnalité pivot de l'implémentation de conversation par un gestionnaire d'entités étendu. Souvenez-vous que seul le flush peut provoquer les ordres SQL que nous souhaitons éviter.
- Un flush à la fin de la conversation suivi d'un commit de la transaction base de données.

Avec cette méthode, tant que vous êtes en invocation locale (dans la même JVM), vous n'avez pas besoin de vous soucier des LazyInitializationException, détachement/réattachement, préchargement ultra-rigoureux du graphe d'entités pour un cas d'utilisation donné puisque les entités sont constamment surveillées par le gestionnaire d'entités d'origine. Tout changement apporté à ces objets est propagé au flush, donc à la validation de la conversation, et l'accès à des associations lazy est résolu de manière transparente tant que vous y accédez depuis la même JVM.

Problématiques

La théorie que nous venons de développer pose deux problématiques. La première est que si le cycle de vie du gestionnaire d'entités est long, il faut le « stocker » quelque part. Dans un environnement EE, nous disposons des beans session avec état qui seront parfaits et fortement recommandés.

La seconde problématique est que nous avons besoin de flusher manuellement en fin de conversation et que la spécification ne l'autorise pas. Souvenez-vous que la spécification ne propose que deux modes : auto et commit. Cependant, Hibernate propose son propre mode manual.

Implémentation exploitant le *FlushMode.MANUAL* spécifique d'Hibernate

Voici notre bean session avec état exploitant la spécificité d'Hibernate :

```
@Stateful
public class ConversationDemoExtended implements ConversationDemo {

  @PersistenceContext(
    unitName = "eyrollesEntityManager",
    type = PersistenceContextType.EXTENDED,
    properties = @PersistenceProperty(
      name="org.hibernate.flushMode",
      value="MANUAL")
```

```
    )
    private EntityManager em;

    public List<Team> getAllTeams() {
      List<Team> result = null;
      result = em
        .createQuery("select team from Team team")
        .getResultList();
      return result;
    }

    public Team getTeam(Integer id) {
      Team result = em.find(Team.class,id);
      return result;
    }

    public List<Coach> getFreeCoachs() {
      List<Coach> results = em
        .createQuery("from Coach c where c.team is null")
        .getResultList();
      return results;
    }

    public List<Player> getFreePlayers() {
      List<Player> results = em
        .createQuery("from Player p where p.team is null")
        .getResultList();
      return results;
    }

    @Remove
    public void validateModification(Team team) {
      em.flush();
    }
  }
```

Tout d'abord, notez que le bean session est annoté avec @Stateful. En l'absence d'annotation @TransactionAttribute, toutes les méthodes suivent le comportement REQUIRED, donc un begin en début de méthode et un commit en fin de méthode si des ressources transactionnelles entrent en jeu, avec potentiellement récupération ou propagation du contexte transactionnel.

L'une des clés de notre implémentation étant de garder en vie le contexte de persistance le temps de la conversation, il nous faut ensuite déclarer notre contexte de persistance comme étendu et basculer dans le mode flush manuel spécifique d'Hibernate, ce qui est fait *via* :

```
    @PersistenceContext(
      unitName = "eyrollesEntityManager",
      type = PersistenceContextType.EXTENDED,
      properties = @PersistenceProperty(
```

```
        name="org.hibernate.flushMode",
        value="MANUAL")
    )
```

À partir de ce moment :

- Le contexte de persistance n'est plus couplé au contexte transactionnel mais au cycle de vie du bean session.
- Le flush n'est plus géré de manière automatique par le conteneur mais par l'application elle-même *via* l'invocation de `em.flush()`.

Tant que nous sommes dans la même JVM, nous pouvons accéder aux associations non chargées. Cela explique pourquoi la méthode `getTeam(Integer id)` n'exécute plus de requête préchargeant l'intégralité du graphe d'entités. En exécutant le test, vous verrez, qu'au niveau de votre méthode JUnit la résolution des associations non chargées se fait de manière transparente. Cependant, pour des soucis d'optimisation évoqués au chapitre 5, il vous faudra probablement étudier les différentes possibilités.

Remarquez l'implémentation de `validateModification()` :

```
@Remove
public void validateModification() {
  em.flush();
}
```

Très simple, étant donné le comportement transactionnel, `em.flush()` est « encapsulé » par une transaction. Depuis le début de la conversation, le gestionnaire d'entités a traqué les modifications apportées aux entités ; un simple flush permet d'exécuter la synchronisation avec la base de données. @Remove stipule le retrait de l'instance de l'EJB session par le conteneur. Cette méthode marque donc aussi la fermeture de son contexte de persistance.

Implémentation officielle et bridée

Si nous souhaitons ne pas avoir recours au mode flush manuel spécifique d'Hibernate, cela veut dire que nous devons trancher entre les modes standards auto et commit. Écartons d'office le mode auto puisque celui-ci peut exécuter un flush avant l'exécution de certaines requêtes.

Il nous reste le mode commit avec la nécessité de ne pas flusher le gestionnaire d'entités et le besoin d'accéder à la base de données en lecture pour la résolution des associations non chargées. L'astuce réside dans le choix de @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED) pour toutes les méthodes, excepté celle de validation.

Pour rappel, avec NOT_SUPPORTED, si un accès à une ressource transactionnelle est requis, celui-ci s'effectue en dehors d'un contexte transactionnel en exploitant la fonctionnalité d'autocommit. Comprenez par là que cet autocommit n'est pas intercepté par le conteneur comme un commit traditionnel et donc que le flush ne sera pas déclenché.

Comme c'est le comportement majoritaire, nous allons annoter la classe. Ainsi toutes les méthodes apporteront ce comportement :

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class ConversationDemoExtended implements ConversationDemo {

  @PersistenceContext(
    unitName = "eyrollesEntityManager",
    type = PersistenceContextType.EXTENDED,
    properties = @PersistenceProperty(
      name="org.hibernate.flushMode",
      value="MANUAL")
  )
  private EntityManager em;

  public List<Team> getAllTeams() {
    List<Team> result = null;
    result = em
      .createQuery("select team from Team team")
      .getResultList();
    return result;
  }

  public Team getTeam(Integer id) {
    Team result = em.find(Team.class,id);
    return result;
  }

  public List<Coach> getFreeCoachs() {
    List<Coach> results = em
      .createQuery("from Coach c where c.team is null")
      .getResultList();
    return results;
  }

  public List<Player> getFreePlayers() {
    List<Player> results = em
      .createQuery("from Player p where p.team is null")
      .getResultList();
    return results;
  }

  @Remove
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void validateModification(Team team) {

  }
}
```

Seule la méthode de validation surcharge ce comportement : on l'annote avec @TransactionAttribute(TransactionAttributeType.REQUIRED). Cette méthode peut être vide, le simple fait que le conteneur commit le contexte transactionnel en fin de méthode suffisant à déclencher le flush.

La grosse limitation de cette implémentation concerne les impacts de la déclaration NOT_SUPPORTED : le contexte transactionnel du client invoquant les méthodes est suspendu et non propagé.

Contexte de persistance étendu dans un environnement autonome Java SE

Dans un environnement autonome java SE, ce modèle peut être implémenté mais ne va pas sans poser problème dans la mesure où, entre chaque étape, le contrôle nous échappe complètement et aucun serveur d'applications n'est présent comme rempart pour la gestion des ressources.

Comme toutes les ressources, la gestion du nombre de connexions est essentielle pour nos applications. Or le gestionnaire d'entités ouvre une connexion JDBC s'il en a besoin et nous ne pouvons nous permettre de laisser une connexion JDBC ouverte *x* minutes. Imaginez, par exemple, que l'utilisateur aille boire un café et revienne dix minutes plus tard. Sa connexion aurait pu être utilisée par quelqu'un d'autre. La gestion des ressources n'est donc pas optimale.

La transaction sous-jacente pose davantage de problèmes, relatifs aux potentiels verrous qu'elle gère.

Pour toutes ces raisons, il est indispensable de fermer la connexion JDBC entre chaque étape. En fonction de l'implémentation de Java Persistence, cela n'est pas toujours simple voire possible. Sachez cependant qu'Hibernate est des plus robustes en matière de gestion des ressources.

Si vous souhaitez implémenter des contextes de persistance étendus dans un environnement java SE, nous vous recommandons d'exploiter directement les API d'Hibernate et non de Java Persistence, qui ne spécifie pas de manière assez sûre le rapport entre le gestionnaire d'entités et la connexion JDBC et surtout sa déconnexion/connexion.

La communauté Hibernate propose des modèles éprouvés pour implémenter le contexte de persistance étendu avec la session Hibernate. Le point de départ est la page : *http://www.hibernate.org/42.html.*

En résumé

La notion de conversation est récurrente dans les applications d'entreprise, où un cas d'utilisation peut rarement être géré en un seul écran ou en un seul aller/retour entre l'utilisateur et le serveur, que l'application soit Web ou non.

Pour implémenter vos conversations, vous partirez probablement d'une logique sans état à base de servlet ou de bean session sans état, comme beaucoup d'architectes ou développeurs en ont pris l'habitude.

Cependant, suite à cette partie, posez-vous les bonnes questions, et mesurez bien la puissance des beans session avec état, qui répondent parfaitement à ce genre de besoin.

Manipulation du gestionnaire d'entités dans un batch

Les batch ont vocation à effectuer des traitements de masse, le plus souvent des insertions ou extractions de données. À ce titre, ils ne profitent que très rarement d'une logique métier. De ce fait, le passage par votre modèle de classes, et donc par Java Persistence, pour ce genre de traitement n'est pas le plus adapté.

Même s'il n'est pas rare de voir Java Persistence utilisé pour les batch, cela peut être catastrophique pour les performances si une gestion adaptée n'est pas envisagée. Nous verrons cependant que l'overhead engendré par Java Persistence est nul par rapport à JDBC, pour peu que l'outil soit bien utilisé.

Il est important de rappeler que Java n'est pas forcément le meilleur langage pour coder des batch. Des outils moins lourds permettent d'insérer ou de mettre à jour des données en masse.

Ajoutons que les outils de mapping objet-relationnel relèvent d'une philosophie et d'une intelligence qui engendrent un léger surcoût en termes de performance. Si ce coût est négligeable en comparaison des garanties et fonctionnalités offertes aux applications complexes, il n'en va pas de même avec les traitements de masse.

Best practice de session dans un batch

Avant de décrire le modèle de manipulation du gestionnaire d'entités dans un batch, voici typiquement ce qu'il ne faut pas faire :

```
EntityManager em = (EntityManager) new InitialContext()
  .lookup("java:/EntityManagers/eyrollesEntityManager");

TransactionManager tm = (TransactionManager) new InitialContext()
  .lookup("java:/TransactionManager");

tm.begin();
for ( int i=0; i<100000; i++ ) {
  Team team = new Team();
  em.persist(team);
}

tm.commit();
```

Avec un tel code, à chaque appel de `em.persist(team)` le cache de premier niveau qu'est le gestionnaire d'entités augmente en taille puisqu'il contient une nouvelle instance. Cela engendre rapidement un manque de mémoire et soulève une OutOfMemoryException.

Insertion optimisée par paquets

Voyons comment améliorer ce code. Définissons d'abord le paramètre `batch_size` (optimisation JDBC de plus bas niveau) entre 10 et 20. Disons, pour simplifier, que cela permet de regrouper les appels JDBC semblables par paquets. Vous pouvez le faire *via* persistence.xml en ajoutant :

```
<property name="hibernate.batch_size" value="20"/>
```

Afin d'éviter que le gestionnaire d'entités augmente indéfiniment, nous allons le vider. Cela n'a aucun impact, puisque le batch n'a plus besoin des objets insérés et que, sans autre valeur ajoutée, il ne tire plus profit du cache de premier niveau. Pour vider le gestionnaire d'entités, nous invoquons simplement `em.clear()`.

Rendre persistante une entité signifie connaître son identifiant. Il faut donc optimiser la génération d'identifiant. Dans notre cas, il faudrait :

- interroger la base de données une première fois pour récupérer la valeur d'identifiant la plus haute ;
- réserver la plage des vingt prochaines valeurs en incrémentant cette valeur en base de données ;
- incrémenter cette valeur en mémoire à chaque appel de `em.persist()`.

Un générateur d'identifiant optimisé par Hibernate va nous permettre d'adopter ce comportement :

```
@Entity
public class Team {
  @Id
  @GeneratedValue(generator="SequenceStyleGenerator")
  @GenericGenerator(name="SequenceStyleGenerator",
    strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
    parameters = {
      @Parameter(name="optimizer", value="hilo"),
      @Parameter(name="initial_value", value="1"),
      @Parameter(name="increment_size", value="20")
    })
  private int id;
  …
}
```

Nous retrouvons les besoins évoqués précédemment. Il s'agit de surcharger les valeurs par défaut de l'implémentation de SEQUENCE proposée par Hibernate. Nous en parlions au chapitre 3 (pour plus de détails, voir le wiki *http://in.relation.to/Bloggers/New323HibernateIdentifierGenerators*).

Avant de vider le gestionnaire d'entités, il faut s'assurer qu'il est synchronisé avec la base de données. Nous forçons donc le flush avec `em.flush()`. Le tout se déroule dans une boucle, qui traite les insertions par paquet de 20. Selon l'importance du graphe d'objets, vous pouvez jouer sur ce paramètre, en prenant soin de surveiller le `batch_size`.

Voici le code optimisé :

```
tm.begin();

for ( int i=0; i<100000; i++ ) {
  Team team = new Team();
  em.persist(team);
  if(i % 20 == 0){
    em.flush();
    em.clear();
  }
}

tm.commit();
```

Scrolling

Le code ci-dessous permet des mises à jour efficaces d'instances persistantes tirant parti de la fonction scroll sur une requête. Là encore, l'objectif est d'éviter de charger la totalité des objets retournés par la requête et de les traiter par paquet de taille raisonnable. Vous optimisez de la sorte le rapport consommation mémoire/nombre de requêtes exécutées.

Java Persistence ne spécifiant pas de méthode exploitant scroll, il est nécessaire d'exploiter directement la session Hibernate :

```
Session session = (Session)em.getDelegate();
tm.begin();
ScrollableResults teams = session.getNamedQuery("GetTeams")
   .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( teams.next() ) {
   Team team = (Team) teams.get(0);
   team.updateStuff(...);
   if ( ++count % 20 == 0 ) {
      //synchronise la base de données avec les mises
      //à jour et libère la mémoire
      session.flush();
      session.clear();
   }
}
tm.commit();
```

Requêtes EJB-QL de type DML (Data Manipulation Language)

Si votre batch est dénué de toute logique métier nécessitant du code java, vous pouvez exécuter des requêtes d'insertion, effacement ou mise à jour massives *via* EJB-QL. Il s'agit d'une traduction directe de JPA-QL vers SQL, ce qui signifie que les entités ne sont pas chargées en mémoire et que vous n'exploitez aucun cache, ni celui du gestionnaire d'entités ni un potentiel cache de second niveau.

La conséquence directe est qu'une telle requête sera ignorée par les caches et donc que ceux-ci pourront se retrouver dans un état incohérent.

Exemple de mise à jour en masse :

```
tm.begin();
Query q = em.createQuery("update Team t set t.name = :param");
q.setParameter("param", "blah");
q.executeUpdate();
tm.commit();
```

Rien de particulier dans cette requête. Pour les requêtes de type DML, il faut simplement les exécuter *via* `executeUpdate()` ; cette méthode renvoie par ailleurs le nombre de lignes impactées en base de données.

Cette requête est la même qu'une requête DML sauf qu'elle exploite les noms d'entités et leurs propriétés au lieu des tables et colonnes. Elle présente le grand avantage de reconnaître l'héritage et saura déclencher plusieurs requêtes au besoin, si vous ciblez une superclasse. UPDATE ne peut prendre qu'un seul type d'entité comme cible ; par contre, vous pouvez exploiter des sous-requêtes avec jointures dans la clause WHERE.

Selon la spécification Java Persistence, le numéro de version n'est pas impacté par de telles requêtes. Cependant, Hibernate vous autorise le mot-clé `versioned`, et incrémente la version des enregistrements impactés :

```
Query q = em.createQuery("update versioned Team t set t.name = :param");
```

Exemple de suppression en masse :

```
tm.begin();
Query q = em.createQuery("delete Team t set t.name = :param");
q.setParameter("param", "blah");
q.executeUpdate();
tm.commit();
```

Les mêmes règles que pour l'update s'appliquent.

Exemple d'insertion en masse :

```
insert into EntityB (propA, propB)
select mustMatchPropAType, mustMatchPropBType
from EntityA e join e.anotherEntity
where anotherEntity.propX = :param
```

Java Persistence ne supporte pas ce genre d'écriture, mais vous pouvez l'exploiter. Hibernate fonctionnant en arrière-plan, cela ne posera pas de problème.

Les prérequis pour exécuter une telle requête sont les suivants :

- Les propriétés ciblées ne doivent pas être celles d'une classe abstraite, ce qui est totalement logique.
- Les types entre le select et l'insert doivent correspondre.
- Les identifiants seront générés selon la stratégie définie par les métadonnées.

En résumé

Grâce à ces deux techniques d'écriture de batch et aux types de requêtes DML, les problèmes de performances devraient être considérablement réduits.

Gardez en tête que chaque technologie possède ses avantages et inconvénients. En tout état de cause, ces méthodes rendent l'utilisation de Java Persistence et d'Hibernate comparable à du JDBC pur.

Interpréter les exceptions

La grande majorité des exceptions soulevées par Java Persistence ne sont pas récupérables et doivent être considérées comme fatales pour le gestionnaire d'entités en cours.

Dans un environnement Java EE, le conteneur se charge de la gestion des ressources et du gestionnaire d'entités. Cependant, dans un environnement Java SE il est important de rappeler la logique globale de manipulation d'un gestionnaire d'entités.

Voici un exemple robuste d'utilisation du gestionnaire d'entités dans un environnement Java SE :

```
try{
  em =  JavaPersistenceUtil.getEmf().createEntityManager();
  tx =  em.getTransaction();
  tx.begin();
  …
  tx.commit();
} catch (RuntimeException ex) {
  try {
    tx.rollback();
  } catch (RuntimeException rbEx) {
  }
    throw ex;
  } finally {
    em.close();
  }
}
```

Les deux points essentiels sont les suivants :

- Effectuer un rollback sur la transaction.
- Fermer le gestionnaire d'entités dans une clause `finally` afin que le gestionnaire d'entités ne puisse être réutilisé et que la connexion JDBC en cours soit rendue au pool de connexions.

Il est indispensable d'identifier les raisons qui ont abouti à une exception. Le tableau 7.1 récapitule les exceptions susceptibles d'être soulevées.

Tableau 7.1 Exceptions Java Persistence

Exception	Description
PersistenceException	Soulevée par le fournisseur de persistance lorsqu'un problème survient. Elle peut être soulevée pour indiquer que l'opération invoquée n'as pu être exécutée à cause d'une erreur inattendue (par exemple, la base de données ne répond pas). Toutes les autres exceptions définies par la spécification héritent de PersistenceException. À l'exception de NoResultException et NonUniqueResultException, elles marquent la transaction en cours pour rollback.
TransactionRequiredException	Soulevée lorsqu'une transaction est requise mais non active.
OptimisticLockException	Soulevée lorsqu'un conflit de verrou optimiste survient. Peut être soulevée par un appel d'une API, au flush ou au commit.
RollbackException	Soulevée lorsqu'il y a échec du commit.
EntityExistsException	Soulevée lors de l'invocation de persist() sur une entité qui existe déjà.
EntityNotFoundException	Soulevée lorsqu'on accède à une entité obtenue *via* getReference() (obtention d'un proxy et non de l'entité réelle) et que cette entité n'existe pas. Soulevée aussi lors de l'invocation de refresh() sur une entité qui n'existe plus.
NonUniqueResultException	Soulevée si l'application invoque Query.getSingleResult() et que la requête renvoie plus d'un résultat. Il s'agit d'une des rares exceptions non bloquantes.
NoResultException	Soulevée si l'application invoque Query.getSingleResult() et que la requête ne renvoie pas de résultat. Il s'agit d'une des rares exceptions non bloquantes.

En résumé

La qualité d'une application se mesure en partie aux indications qu'elle est capable de fournir lorsqu'un problème survient. Pour cela, la gestion et l'interprétation des exceptions sont fondamentales.

Vous disposez des éléments qui vous seront utiles pour identifier et gérer les problèmes pouvant être soulevés par la couche persistance de vos applications.

Conclusion

Ce chapitre a passé en revue les différentes techniques d'obtention/manipulation du gestionnaire d'entités, qui ne devrait plus avoir de secret pour vous. Vous devriez en outre être capable d'interpréter les exceptions soulevées et d'écrire des batch avec Java Persistence de manière optimisée.

Vous découvrirez au chapitre 8 des points d'extension et d'intégration, ainsi que des fonctionnalités très avancées d'Hibernate.