Pro JPA 2 dans Java EE 8

Un guide détaillé des API de persistance Java

-

Troisième édition

-

Mike Keith
Merrick Schincariol
Massimo Nardone

Pro JPA 2 dans Java EE 8

Un guide détaillé de Java API de persistance

Troisième édition

Mike Keith
Merrick Schincariol
Massimo Nardone

Pro JPA 2 dans Java EE 8: un guide détaillé des API de persistance Java

Mike Keith
Ottawa, Ontario, Canada

Merrick Schincariol
Almonte, Ontario, Canada

Massimo Nardone
Helsinki, Finlande

ISBN-13 (pbk): 978-1-4842-3419-8 ISBN-13 (électronique): 978-1-4842-3420-4
https://doi.org/10.1007/978-1-4842-3420-4

Numéro de contrôle de la Bibliothèque du Congrès: 2018932342

Directeur général: Welmoed Spahr
Directeur éditorial: Todd Green
Responsable des acquisitions: Steve Anglin
Éditeur du développement: Matthew Moodie
Réviseur technique: Mario Faliero
Éditeur coordinateur: Mark Powers
Rédacteur en chef: Kezia Endsley

Distribué au commerce du livre dans le monde entier par Springer Science + Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Téléphone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny @ springer-sbm.com ou visitez www.springeronline.com. Apress Media, LLC est une LLC californienne et le seul membre (propriétaire) est Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc est une Société du **Delaware** .

Pour plus d'informations sur les traductions, veuillez envoyer un e-mail à rights@apress.com ou visiter http://www.apress.com/droits-permissions.

Les titres Apress peuvent être achetés en vrac à des fins académiques, professionnelles ou promotionnelles. Versions de livres électroniques et des licences sont également disponibles pour la plupart des titres. Pour plus d'informations, reportez-vous à nos ventes en bloc imprimées et électroniques page Web à http://www.apress.com/bulk-sales.

Tout code source ou autre matériel supplémentaire référencé par l'auteur dans ce livre est disponible pour lecteurs sur GitHub via la page produit du livre, disponible à l'adresse www.apress.com/9781484234198. Pour plus des informations détaillées, veuillez visiter http://www.apress.com/source-code.

Imprimé sur papier sans acide

À ma femme Darleen, la mère parfaite, et à Cierra, Ariana,
Jeremy et Emma, qui égayent ma vie et me font
efforcez-vous d'être une meilleure personne.

-Mike

À Anthony, dont la créativité sans bornes continue de m'inspirer.
À Evan, dont l'enthousiasme bruyant me motive à
nouveaux défis. À Kate, qui prouve que la taille n'est pas un objet quand
vous avez la bonne attitude. Je vous aime tous.

—Merrick

Je voudrais dédier ce livre à la mémoire de ma bien-aimée
feu mère Maria Augusta Ciniglio. Merci maman pour tout le
de grandes choses que tu m'as apprises, pour avoir fait de moi une bonne personne, pour
me faisant étudier pour devenir informaticien, et pour le
grands souvenirs que tu m'as laissé. Vous serez aimé et regretté à jamais.
Je t'aime maman. DÉCHIRURE.

—Massimo

Table des matières

À propos des auteurs xvii

À propos du réviseur technique xix

Remerciements xxi

Bases de données relationnelles 2

Mappage objet-relationnel 3

Le décalage d'impédance 4

Prise en charge Java pour la persistance 11

Solutions propriétaires 11

JDBC 13

Enterprise JavaBeans 13

Objets de données Java 15

Pourquoi une autre norme? 16

L'API Java Persistence 17

Historique de la spécification 17

Aperçu 21

Résumé 24

Présentation de l'entité 25

Persistabilité 26

Identité 26

Transactionnalité 27

Granularité 27

Métadonnées d'entité 28

Annotations 28

XML 30

Configuration par exception 30

Création d'une entité 31

Gestionnaire d'entités 34

Obtention d'un Entity Manager 36

Persistance d'une entité 37
Recherche d'une entité 38
Suppression d'une entité 39
Mise à jour d'une entité 40
Transactions 41
Requêtes 42
Mettre tous ensemble 44
Emballage 47
Unité de persistance 47
Archive de persistance 48
Résumé 49

Modèles de composants d'application 54
Session Beans 56
Beans session apatrides 57
Beans session avec état 61
Beans session singleton 65
Servlets 67
Gestion des dépendances et CDI 69
Recherche de dépendances 70
Injection de dépendance 72
Déclaration des dépendances 74

CDI et injection contextuelle 78
Haricots CDI 78
Injection et résolution 79
Portées et contextes 80
Injection qualifiée 81
Méthodes et champs du producteur 82
Utilisation des méthodes Producer avec les ressources JPA 83
Gestion des transactions 85
Revue de transaction 85
Transactions d'entreprise en Java 86
Mettre tous ensemble 95
Définition du composant 96
Définition de l'interface utilisateur 97
Emballage 98
Résumé 99

Annotations de persistance 102
Accès à l'état de l'entité 103
Accès sur le terrain 104
Accès à la propriété 105

Accès mixte 106
Mappage à une table 108
Mappage de types simples 110
Mappages de colonnes 111
Récupération paresseuse 113
Grands objets 114
Types énumérés 115
Types temporels 118
État transitoire 119

Page 8

Mappage de la clé primaire 120
Remplacer la colonne de clé primaire 120
Types de clé primaire 121
Génération d'identifiant 121
Des relations 129
Concepts relationnels 129
Présentation des mappages 132
Associations à valeur unique 133
Associations valorisées par la collection 140
Relations paresseuses 148
Objets intégrés 149
Résumé 154

Relations et collections d'éléments 157
Utilisation de différents types de collection 161
Ensembles ou collections 162
Listes 162
Plans 167
Doublons 185
Valeurs nulles 187
Les meilleures pratiques 188
Résumé 189

Contextes de persistance 191
Gestionnaires d'entités 192
Gestionnaires d'entités gérées par conteneurs 192
Gestionnaires d'entités gérées par les applications 198
Gestion des transactions 201
Gestion des transactions JTA 202
Transactions de ressources locales 218
Annulation de transaction et état de l'entité 221

Choisir un Entity Manager 224
Opérations du gestionnaire d'entités 225
Persistance d'une entité 225
Recherche d'une entité 227
Suppression d'une entité 228
Opérations en cascade 229
Effacement du contexte de persistance 234
Synchronisation avec la base de données 234
Détachement et fusion 238
Détachement 238
Fusion d'entités détachées 241
Travailler avec des entités détachées 246
Résumé 267

Java Persistence Query Language 270
Commencer 270
Filtrage des résultats 271
Projection des résultats 272
Jointures entre entités 272
Requêtes agrégées 273
Paramètres de requête 273
Définition des requêtes 274
Définition de requête dynamique 275
Définition de requête nommée 278
Requêtes nommées dynamiques 280
Types de paramètres 282
Exécution de requêtes 285
Utilisation des résultats de requête 287
Résultats de la requête de flux 288
Requête de pagination 293

Requêtes et modifications non validées 296
Délai d'expiration des requêtes 299
Mise à jour et suppression en masse 300
Utilisation de la mise à jour et de la suppression en masse 301
Suppression groupée et relations 304
Conseils de requête 305
Bonnes pratiques relatives aux requêtes 307
Requêtes nommées 307
Rapport de requêtes 308
Conseils du fournisseur 308

Haricots apatrides 309
Mise à jour et suppression en masse 309
Différences entre les fournisseurs 310
Résumé 310

Présentation de JPQL 313
Terminologie 314
Exemple de modèle de données 315
Exemple d'application 316
Sélectionner les requêtes 319
Clause SELECT 321
Clause FROM 325
Clause WHERE 336
Héritage et polymorphisme 344
Expressions scalaires 347
ORDER BY Clause 353
Requêtes agrégées 354
Fonctions d'agrégation 356
Clause GROUP BY 357
Clause HAVING 358
Requêtes de mise à jour 359

Supprimer les requêtes 360
Résumé 361

Aperçu 363
L'API Criteria 364
Types paramétrés 365
Requêtes dynamiques 366
Création de requêtes API de critères 370
Création d'une définition de requête 370
Structure basique 372
Objets critères et mutabilité 373
Racines de requête et expressions de chemin 374
La clause SELECT 377
La clause FROM 382
La clause WHERE 384
Construire des expressions 385
La clause ORDER BY 401
Les clauses GROUP BY et HAVING 402
Mise à jour et suppression en masse 403
Définitions de requêtes fortement typées 405
L'API Metamodel 405
Présentation de l'API fortement typée 407
Le métamodèle canonique 409
Choisir le bon type de requête 412

Résumé 413

Chapitre 10: Mappage relationnel d'objet avancé 415

Noms des tables et des colonnes 416

Conversion de l'état de l'entité 418

Création d'un convertisseur 418

Conversion d'attributs déclaratifs 420

Conversion automatique 423

Convertisseurs et requêtes 424

Objets intégrés complexes 425

Mappages intégrés avancés 425

Remplacement des relations intégrées 427

Clés primaires composées 429

Classe d'identité 430

Classe d'identification intégrée 432

Identificateurs dérivés 434

Règles de base pour les identificateurs dérivés 435

Clé primaire partagée 436

Attributs mappés multiples 439

Utilisation d'EmbeddedId 440

Éléments de mappage avancés 443

Mappages en lecture seule 444

Optionalité 445

Relations avancées 446

Utilisation des tables de jointure 446

Éviter les tables de jointure 447

Colonnes de jointure composées 449

Suppression des orphelins 451

Cartographie de l'état de la relation 453

Tables multiples 456

Héritage 461

Hiérarchies de classe 461

Modèles d'héritage 466

Héritage mixte 477

Résumé 480

Requêtes SQL 483
Requêtes natives et JDBC 484
Définition et exécution de requêtes SQL 487
Mappage de l'ensemble de résultats SQL 491
Liaison de paramètres 500
Procédures stockées 500
Graphiques d'entité 505
Annotations du graphe d'entité 507
API Entity Graph 516
Gestion des graphiques d'entité 519
Utilisation des graphiques d'entité 522
Résumé 525

Rappels de cycle de vie 527
Événements du cycle de vie 528
Méthodes de rappel 529
Écouteurs d'entités 531
Événements d'héritage et de cycle de vie 536
Validation 542
Utilisation des contraintes 543
Appel de la validation 545
Groupes de validation 546
Création de nouvelles contraintes 548
Validation dans JPA 551
Activation de la validation 552
Définition des groupes de validation du cycle de vie 553

Concurrence 555
Opérations d'entité 555
Accès aux entités 555
Actualisation de l'état de l'entité 555
Verrouillage 559
Verrouillage optimiste 560
Verrouillage pessimiste 574
Mise en cache 580
Tri des couches 580
Cache partagé 582
Classes d'utilité 589
PersistenceUtil 589
PersistenceUnitUtil 590

Résumé 591

Le puzzle des métadonnées ... 593

Le fichier de mappage ... 596

Désactivation des annotations ... 598

Valeurs par défaut de l'unité de persistance ... 601

Mappage des valeurs par défaut des fichiers ... 606

Requêtes et générateurs ... 609

Classes gérées et mappages ... 617

Convertisseurs ... 652

Résumé ... 654

Configuration des unités de persistance ... 656

Nom de l'unité de persistance ... 656

Type de transaction ... 657

Fournisseur de persistance ... 658

La source de données ... 659

Fichiers de mappage ... 662

Classes gérées ... 663

Mode de cache partagé ... 667

Mode de validation ... 668

Ajout de propriétés ... 668

Construction et déploiement ... 669

Déploiement Classpath ... 669

Options d'emballage ... 670

Portée de l'unité de persistance ... 676

En dehors du serveur ... 677

Configuration de l'unité de persistance ... 678

Spécification des propriétés au moment de l'exécution ... 680

Chemin de classe système ... 681

Génération de schéma ... 682

Le processus de génération ... 683

Propriétés de déploiement ... 684

Propriétés d'exécution ... 689

Cartographie des annotations utilisées par la génération de schéma ... 689

Contraintes uniques ... 690

Contraintes nulles ... 691

Index ... 692

Contraintes de clé étrangère ... 692

Colonnes basées sur des chaînes ... 694

Colonnes à virgule flottante ... 695

Définition de la colonne ... 696

Résumé ... 697

Test des applications d'entreprise 699
Terminologie 700
Test en dehors du serveur 702
JUnit 703

Test unitaire 704
Entités de test 704
Test des entités dans les composants 706
Le gestionnaire d'entités dans les tests unitaires 709
Test d'intégration 713
Utilisation de Entity Manager 714
Composants et persistance 722
Cadres de test 736
Les meilleures pratiques 738
Résumé 739
Index 741

à propos des auteurs

Mike Keith était le responsable de la co-spécification pour JPA 1.0 et un membre des groupes d'experts JPA 2.0 et JPA 2.1. Il s'assied sur un certain nombre d'autres groupes d'experts Java Community Process et le groupe d'experts des entreprises (EEG) de l'OSGi Alliance. Il est titulaire d'une maîtrise en informatique de Université Carleton, et possède plus de 20 ans d'expérience dans recherche et pratique sur la persistance et les systèmes distribués. Il a écrit des articles et des articles sur JPA et a parlé à de nombreuses conférences à travers le monde. Il est employé comme architecte chez Oracle à Ottawa, au Canada, et est marié avec quatre enfants et deux chiens.

Merrick Schincariol est ingénieur conseil chez Oracle, spécialisée dans les technologies middleware. Il a un baccalauréat of Science diplôme en informatique de Lakehead Université, et a plus d'une décennie d'expérience dans le développement de logiciels d'entreprise. Il en a dépensé conseil en temps dans l'entreprise et les affaires pré-Java champs d'intelligence avant de passer à l'écriture de Java et J2EE applications. Son expérience des systèmes à grande échelle et la conception de l'entrepôt de données lui a donné une expérience perspective sur les logiciels d'entreprise, qui ont ensuite propulsé lui à faire des travaux d'implémentation de conteneurs Java EE.

Massimo Nardone a plus de 24 ans d'expérience en sécurité, développement Web / mobile, cloud et informatique architecture. Ses véritables passions informatiques sont la sécurité et Android.

Il a programmé et enseigné aux autres comment programme avec Android, Perl, PHP, Java, VB, Python, C / C ++, et MySQL depuis plus de 20 ans.

Il est titulaire d'une maîtrise ès sciences en informatique Science de l'Université de Salerne, Italie. Il a travaillé en tant que chef de projet, ingénieur logiciel, ingénieur de recherche, Architecte en chef de la sécurité, responsable de la sécurité de l'information, Auditeur PCI / SCADA et Architecte principal responsable de la sécurité informatique / Cloud / SCADA pour de nombreuses années.

Ses compétences techniques incluent la sécurité, Android, le cloud, Java, MySQL, Drupal, Cobol, Perl, développement Web et mobile, MongoDB, D3, Joomla, Couchbase, C / C ++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

Il a travaillé comme conférencier invité et superviseur au laboratoire de réseautage de la Université de technologie d'Helsinki (Université Aalto). Il détient également quatre internationaux brevets (dans les domaines PKI, SIP, SAML et Proxy).

Il travaille actuellement en tant que Chief Information Security Office (CISO) pour Cargotec Oyj et

est membre du conseil d'administration de la section ISACA Finlande.

Massimo a révisé plus de 45 livres informatiques pour différents éditeurs et est le co-auteur de *Pro Android Games* (Apress, 2015).

À propos du réviseur technique

Mario Faliero est ingénieur en télécommunications et entrepreneur. Il a plus de dix ans d'expérience avec ingénierie matérielle de radiofréquence. Mario a de nombreux expérience en codage numérique, en utilisant des langages de script (MATLAB et Python) et les langages compilés (C / C ++ et Java). Il a été responsable du développement de outils d'évaluation électromagnétique pour l'espace et le commercial applications. Mario a obtenu sa maîtrise de la Université de Sienne.

Page 20

Remerciements

Un grand merci à ma merveilleuse famille - ma femme Pia et mes enfants Luna, Leo et Neve - pour m'avoir soutenu dans l'élaboration de ce livre. Tu es le plus bel aspect de ma vie.

Je tiens à remercier ma défunte mère bien-aimée Maria Augusta Ciniglio qui a toujours m'a tellement soutenu et aimé. Je t'aimerai et tu me manquerai pour toujours, ma très chère maman.

Je dois aussi remercier mon père bien-aimé Giuseppe et mes frères Mario et Roberto pour votre amour sans fin et pour être le meilleur papa et les meilleurs frères du monde.

Ce livre est également dédié au docteur Antonio Catapano, pour être un si grand personne avec un grand coeur et prenant soin de moi et de ma mère. À ma belle-sœur Susanna Cennamo, à mes chères cousines Rosaria Scudieri, Pina et Elisa Franzese, et Francesco Ciniglio, pour m'aimer et me soutenir ainsi que ma mère comme nul autre. Vers Pertti et Marianna Kantola, pour m'avoir appris à être un bon programmeur, en prenant soin de moi, et me traitant comme leur fils. À Antti, Piia et Daniela Jalonen pour avoir été formidables et des amis solidaires, ainsi qu'à Anton Jalonen, qui deviendra un excellent logiciel ingénieur. Anton, que ce livre soit une inspiration pour votre grand avenir informatique.

Je tiens également à remercier Steve Anglin et Matthew Moodie de m'avoir donné l'opportunité pour écrire ce livre. Un merci spécial, comme d'habitude, à Mark Powers pour avoir fait un si grand travail et me soutenir pendant le processus éditorial.

Enfin, je tiens à remercier Mario Faliero, un bon ami et le réviseur technique de ce livre, pour m'avoir aidé à faire un meilleur livre.

Page 21

CHAPITRE 1

introduction

Les applications d'entreprise sont définies par leur besoin de collecter, traiter, transformer et rapport sur de grandes quantités d'informations. Et, bien sûr, ces informations doivent être conservées quelque part. Le stockage et la récupération de données est une activité de plusieurs milliards de dollars, en partie par la croissance du marché des bases de données ainsi que par l'émergence du stockage cloud prestations de service. Malgré toutes les technologies disponibles pour la gestion des données, les concepteurs d'applications passent encore beaucoup de temps à essayer de déplacer efficacement leurs données vers et depuis le stockage.

Malgré le succès de la plate-forme Java en travaillant avec des systèmes de base de données, longtemps, il a souffert du même problème qui a tourmenté d'autres objets orientés langages de programmation. Déplacement des données entre un système de base de données et le le modèle objet d'une application Java était beaucoup plus difficile qu'il ne le fallait. Développeurs Java soit a écrit beaucoup de code pour convertir les données de ligne et de colonne en objets, soit s'est retrouvé lié à des cadres propriétaires qui ont tenté de leur cacher la base de données. Heureusement, une solution standard, l'API Java Persistence (JPA), a été introduite dans la plateforme pour combler le fossé entre les modèles de domaine orientés objet et les systèmes de bases de données relationnelles.

Ce livre présente la version 2.2 de l'API Java Persistence dans le cadre de Java EE 8 et explore tout ce qu'il a à offrir aux développeurs.

La version de maintenance de JPA 2.2 a commencé en 2017 sous JSR 338 et a finalement été approuvé le 19 juin 2017.

Voici la déclaration officielle de la version de maintenance de Java Persistence 2.2:

«La spécification Java Persistence 2.2 améliore l'API Java Persistence avec prise en charge de la répétition des annotations; injection dans des convertisseurs d'attributs; soutien pour le mappage de java.time.LocalDate, java.time.LocalTime, java.time. Types LocalDateTime, java.time.OffsetTime et java.time.OffsetDateTime; et méthodes pour récupérer les résultats de Query et TypedQuery sous forme de flux. »

ı **Matériel supplémentaire électronique** La version en ligne de ce chapitre (https://doi.org/10.1007/978-1-4842-3420-4_1) contient du matériel supplémentaire, qui est disponible pour les utilisateurs autorisés.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_1

L'un de ses points forts est qu'il peut être inséré dans n'importe quelle couche, niveau ou cadre une application en a besoin. Que vous construisiez des applications client-serveur pour collecter des données de formulaire dans une application Swing ou créer un site Web à l'aide de la dernière application framework, JPA peut vous aider à assurer la persistance plus efficacement.

Pour préparer le terrain pour JPA, ce chapitre fait d'abord un pas en arrière pour montrer où nous en sommes et quels problèmes nous essayons de résoudre. À partir de là, nous examinerons l'histoire de la spécification et vous donner une vue d'ensemble de ce qu'il a à offrir.

Bases de données relationnelles

De nombreuses façons de conserver des données sont venues et disparues au fil des ans, et aucun concept n'a plus rester en puissance que la base de données relationnelle. Même à l'ère du cloud, lorsque le «Big Data» et «NoSQL» font régulièrement la une des journaux, les services de bases de données relationnelles sont cohérents demande pour permettre aux applications d'entreprise d'aujourd'hui de s'exécuter dans le cloud. Alors que la valeur-clé et les magasins NoSQL orientés document ont leur place, les magasins relationnels restent les bases de données polyvalentes populaires existantes, et c'est là que la grande majorité des les données d'entreprise du monde entier sont stockées. Ils sont le point de départ de chaque application d'entreprise et ont souvent une durée de vie qui se prolonge longtemps après la disparition de l'application.

Comprendre les données relationnelles est la clé d'un développement d'entreprise réussi. Développer des applications pour bien fonctionner avec les systèmes de base de données est un obstacle au développement de logiciels. Une bonne partie du succès de Java peut être attribuée à sa adoption généralisée pour la création de systèmes de base de données d'entreprise. À partir de sites Web de consommateurs aux passerelles automatisées, les applications Java sont au cœur des applications d'entreprise

développement. Figure 1-1 montre un exemple de base de données relationnelle d'utilisateur à voiture.

Figure 1-1. *Base de données relationnelle utilisateur à voiture*

Mappage objet-relationnel

«Le modèle de domaine a une classe. La base de données a une table. Ils se ressemblent beaucoup. Il devrait être simple pour convertir automatiquement l'une en l'autre. » C'est une pensée que nous probablement tous eu à un moment ou à un autre lors de l'écriture d'un autre objet d'accès aux données (DAO) pour convertir les ensembles de résultats Java Database Connectivity (JDBC) en quelque chose d'objet-orienté. Le modèle de domaine ressemble suffisamment au modèle relationnel du base de données qu'il semble réclamer un moyen de faire parler les deux modèles.

La technique de combler le fossé entre le modèle objet et le relationnel modèle est connu sous le nom de mappage objet-relationnel, souvent appelé mappage OU ou simplement ORM. Le terme vient de l'idée que nous cartographions en quelque sorte le concepts d'un modèle à l'autre, dans le but d'introduire un médiateur gérer la transformation automatique de l'un à l'autre.

Avant d'entrer dans les spécificités du mappage objet-relationnel, définissons un bref manifeste de ce que devrait être la solution idéale.

- *Objets et non tableaux* : les applications doivent être rédigées en termes de le modèle de domaine, non lié au modèle relationnel. Ce doit être possible d'opérer et d'interroger sur le modèle de domaine sans devoir l'exprimer dans le langage relationnel des tableaux, des colonnes et clés étrangères.

- *Commodité, pas ignorance* : les outils de cartographie ne doivent être utilisés que par une personne familière avec la technologie relationnelle. OU cartographie ne vise pas à empêcher les développeurs de comprendre la cartographie problèmes ou pour les cacher complètement. Il est destiné à ceux qui ont une compréhension des problèmes et savoir ce dont ils ont besoin, mais qui ne veux pas avoir à écrire des milliers de lignes de code pour traiter un problème qui a déjà été résolu.

- *Discret, pas transparent* : il n'est pas raisonnable de s'attendre à ce que la persistance soit transparente car une application doit toujours avoir le contrôle des objets qu'il persiste et être conscient du cycle de vie de l'entité. La solution de persistance ne doit pas empiéter sur le le modèle de domaine, cependant, et les classes de domaine ne doivent pas étendre les classes ou implémenter des interfaces afin d'être persistantes.

- *Données héritées, nouveaux objets* : il est beaucoup plus probable qu'une application ciblez un schéma de base de données relationnelle existant plutôt que d'en créer un nouveau. La prise en charge des schémas hérités est l'un des cas d'utilisation les plus pertinents cela se produira, et il est fort possible que de telles bases de données survivent chacun de nous.
- *Assez, mais pas trop* : les développeurs d'entreprise ont des problèmes pour résoudre, et ils ont besoin de fonctionnalités suffisantes pour résoudre ces problèmes. Quoi ils n'aiment pas être obligés de manger un modèle de persévérance lourd qui introduit de gros frais généraux car il résout des problèmes qui beaucoup ne sont même pas d'accord sur *les* problèmes.
- *Local, mais mobile* : une représentation persistante des données ne doivent être modélisés comme un objet distant à part entière. Distribution est quelque chose qui fait partie de l'application et non de la couche de persistance. Les entités qui contiennent l'état persistant, cependant, doit pouvoir se déplacer vers la couche qui en a besoin pour que si une application est distribuée, les entités prendront en charge et non inhiber une architecture particulière.
- *API standard, avec implémentations enfichables* : grandes entreprises avec des applications importantes ne veulent pas risquer d'être couplées au produit- bibliothèques et interfaces spécifiques. En ne dépendant que de défini interfaces standard, l'application est découplée des API propriétaires et peut changer d'implémentation si une autre devient plus appropriée.

Cela semble être un ensemble d'exigences quelque peu exigeantes, mais un né à la fois de l'expérience pratique et de la nécessité. Les applications d'entreprise ont besoins de persistance très spécifiques, et cette liste d'achats d'articles est une représentation de l'expérience de la communauté des entreprises.

Le décalage d'impédance

Les partisans du mappage objet-relationnel décrivent souvent la différence entre l'objet modèle et le modèle relationnel comme le décalage d'impédance entre les deux. C'est une description appropriée parce que le défi de la correspondance entre les deux ne réside pas dans le similitudes entre les deux, mais dans les concepts de chacun pour lesquels il n'y a pas de logique équivalent dans l'autre.

Dans les sections suivantes, nous présentons quelques modèles de domaine orientés objet de base et une variété de modèles relationnels pour conserver le même ensemble de données. Comme vous le verrez, le le défi du mappage objet-relationnel n'est pas tant la complexité d'un seul mappage mais qu'il y a tellement de mappages possibles. Le but n'est pas d'expliquer comment aller d'un point à un autre, mais pour comprendre les routes à emprunter arriver à une destination prévue.

Représentation de classe

Commençons cette discussion par une classe simple. Figure 1-2 montre une classe d'employé avec quatre attributs: ID de l'employé, nom de l'employé, date de début et salaire actuel.

Figure 1-2. *La classe Employé*

Considérons maintenant le modèle relationnel illustré à la figure 1-3 . La représentation idéale de cette classe dans la base de données correspond au scénario (A). Chaque champ dans les cartes de classe directement dans une colonne du tableau. L'ID d'employé devient la clé primaire. Avec le à l'exception de quelques légères différences de dénomination, il s'agit d'un mappage simple.

Figure 1-3. *Trois scénarios pour stocker les données des employés*

Dans le scénario (B), nous voyons que la date de début de l'employé est en fait stockée comme trois colonnes séparées, une pour le jour, le mois et l'année. Rappelez-vous que la classe a utilisé un objet Date pour représenter cette valeur. Parce que les schémas de base de données sont beaucoup plus difficiles changer, si la classe est forcée d'adopter la même stratégie de stockage afin de rester cohérent avec le modèle relationnel? Considérez également l'inverse du problème, dans lequel la classe avait utilisé trois champs et la table utilisait une seule colonne de date. Même un seul champ devient complexe à mapper lorsque la base de données et le modèle objet diffèrent représentation.

Les informations salariales sont considérées comme commercialement sensibles, il peut donc être imprudent de placez la valeur du salaire directement dans le tableau EMP, qui peut être utilisé pour un certain nombre de fins. Dans le scénario (C), la table EMP a été fractionnée afin que les informations de salaire soient stocké dans une table EMP_SAL distincte. Cela permet à l'administrateur de la base de données de restreindre SÉLECTIONNEZ l'accès aux informations salariales aux utilisateurs qui en ont réellement besoin. Avec un tel un mappage, même une opération de magasin unique pour la classe Employee nécessite désormais des insertions ou met à jour deux tables différentes.

De toute évidence, même le stockage des données d'une seule classe dans une base de données peut être un défi exercice. Nous nous préoccupons de ces scénarios car de vrais schémas de base de données dans les systèmes de production n'ont jamais été conçus avec des modèles d'objets à l'esprit. La règle de Dans les applications d'entreprise, les besoins de la base de données l'emportent sur les L'application. En fait, il existe généralement de nombreuses applications, certaines orientées objet et certains basés sur SQL (Structured Query Language), qui récupèrent et stockent des données dans une seule base de données. La dépendance de plusieurs applications sur la même base de données signifie que la modification de la base de données affecterait chacune des applications, clairement une option indésirable et potentiellement coûteuse. C'est au modèle objet de s'adapter et trouvez des moyens de travailler avec le schéma de base de données sans laisser la conception physique maîtriser le modèle d'application logique.

Des relations

Les objets existent rarement de manière isolée. Tout comme les relations dans une base de données, les classes de domaine dépendent et s'associent à d'autres classes de domaine. Considérez l'employé classe introduite dans la figure 1-2. Il existe de nombreux concepts de domaine qui pourraient être associés avec un employé, mais pour l'instant introduisons la classe de domaine Address, pour laquelle un L'employé peut avoir au plus une instance. Nous disons dans ce cas que l'employé a un-relation à un avec l'adresse, représentée dans le langage de modélisation unifié (UML) modèle par la notation 0..1. La figure 1-4 illustre cette relation.

Figure 1-4. *La relation employé-adresse*

Nous avons discuté de différents scénarios pour représenter l'état de l'employé dans le précédent section, et de même, il existe plusieurs approches pour représenter une relation dans un schéma de base de données. Figure 1-5 montre trois scénarios différents pour un one-to-one relation entre un employé et une adresse.

Figure 1-5. *Trois scénarios pour relier les données des employés et des adresses*

Le bloc de construction des relations dans la base de données est la clé étrangère. Chaque scénario implique des relations de clé étrangère entre les différentes tables, mais pour être une relation de clé étrangère, la table cible doit avoir une clé primaire. Et donc avant nous même arriver à associer les employés et les adresses les uns avec les autres, nous avons un problème. le La classe de domaine Address n'a pas d'identifiant, mais la table dans laquelle elle serait stockée doit en avoir une si elle veut faire partie des relations. Nous pourrions construire une clé primaire de tous des colonnes de la table ADDRESS, mais cela est considéré comme une mauvaise pratique. Par conséquent, l'ID La colonne est introduite, et le mappage relationnel de l'objet devra s'adapter d'une certaine manière.

Le scénario (A) de la figure 1-5 montre la cartographie idéale de cette relation. La table EMP a une clé étrangère à la table ADDRESS stockée dans la colonne ADDRESS_ID. Si l'employé class détient une instance de la classe Address, la valeur de clé primaire de l'adresse peut être défini pendant les opérations de stockage lorsqu'une ligne EMPLOYEE est écrite.

Et pourtant, considérons le scénario (B), qui n'est que légèrement différent mais tout à coup beaucoup plus complexe. Dans le modèle de domaine, une instance Address ne conservait pas le Instance de l'employé qui en était propriétaire, mais la clé primaire de l'employé doit être stockée dans la table ADRESSE. Le mappage objet-relationnel doit soit tenir compte de cette incompatibilité entre la classe de domaine et la table ou une référence à l'employé devra être ajouté pour chaque adresse.

Pour aggraver les choses, le scénario (C) introduit une table de jointure pour relier le PEM et Tables ADRESSE. Au lieu de stocker les clés étrangères directement dans l'une des tables de domaine, la table de jointure tient la paire de clés. Chaque opération de base de données impliquant les deux les tables doivent maintenant traverser la table de jointure et la garder cohérente. Nous pourrions introduire un EmployeeAddress classe d'association dans le modèle de domaine pour compenser, mais que va à l'encontre de la représentation logique que nous essayons d'obtenir.

Les relations présentent un défi dans toute solution de mappage objet-relationnel. Cette l'introduction ne couvrait que les relations individuelles, et pourtant nous avons été confrontés à la besoin de clés primaires absentes du modèle objet et possibilité de devoir introduire des relations supplémentaires dans le modèle ou même des classes associées pour compenser schéma de base de données.

Héritage

Un élément déterminant d'un modèle de domaine orienté objet est l'opportunité d'introduire relations généralisées entre des classes similaires. L'héritage est le moyen naturel d'exprimer ces relations et permet un polymorphisme dans l'application. Revisitons le Classe d'employés illustrée à la figure 1-2 et imaginez une entreprise qui a besoin de se distinguer entre les employés à temps plein et à temps partiel. Les employés à temps partiel travaillent pour une heure taux, tandis que les employés à temps plein reçoivent un salaire. C'est une bonne opportunité pour héritage, déplacement des informations de salaire vers PartTimeEmployee et FullTimeEmployee sous-classes. La figure 1-6 montre cette disposition.

Figure 1-6. *Relations successorales entre temps plein et temps partiel des employés*

L'héritage pose un véritable problème pour le mappage objet-relationnel. Nous ne sommes pas traitant plus d'une situation dans laquelle il y a un mappage naturel d'une classe à une table. Considérez les modèles relationnels illustrés à la figure 1-7 . Encore une fois, trois différents des stratégies pour conserver le même ensemble de données sont démontrées.

Figure 1-7. *Stratégies d'héritage dans un modèle relationnel*

Sans doute la solution la plus simple pour quelqu'un qui mappe une structure d'héritage à un base de données serait de mettre toutes les données nécessaires pour chaque classe (y compris les classes parentes) dans des tableaux séparés. Cette stratégie est illustrée par le scénario (A) de la figure 1 à 7. Notez que il n'y a pas de relation entre les tables (c'est-à-dire que chaque table est indépendante des autres). Cela signifie que les requêtes sur ces tables sont désormais beaucoup plus compliquées si l'utilisateur doit fonctionner à la fois pour les employés à temps plein et à temps partiel en une seule étape.

Une alternative efficace mais dénormalisée consiste à placer toutes les données requises pour chaque classe dans le modèle dans une seule table. Cela rend les requêtes très faciles, mais notez le

structure du tableau présenté dans le scénario (B) de la figure 1-7. Il y a une nouvelle colonne, TYPE, qui n'existe dans aucune partie du modèle de domaine. La colonne TYPE indique si l'employé est à temps partiel ou à temps plein. Cette information doit maintenant être interprété par une solution de mappage objet-relationnel pour savoir quel type de classe de domaine pour instancier une ligne donnée de la table.

Piste 31

Le scénario (C) va encore plus loin, normalisant cette fois les données dans des tables pour les employés à temps plein et à temps partiel. Contrairement au scénario (A), cependant, ces tableaux sont liés par une table EMP commune qui stocke toutes les données communes aux deux employés les types. Cela peut sembler excessif pour une seule colonne de données supplémentaires, mais un vrai schéma avec de nombreuses colonnes spécifiques à chaque type d'employé utiliserait probablement ce type de tableau structure. Il présente les données sous une forme logique et simplifie également les requêtes en permettant les tables à réunir. Malheureusement, ce qui fonctionne bien pour la base de données fonctionne ne fonctionne pas nécessairement bien pour un modèle objet mappé à un tel schéma. Même sans associations à d'autres classes, le mappage objet-relationnel de la classe de domaine doit prenez maintenant en compte les jointures entre plusieurs tables. Quand vous commencez à considérer l'abstrait superclasses ou classes parentes qui ne sont pas persistantes, l'héritage devient rapidement un problème complexe dans le mappage objet-relationnel. Non seulement le stockage de les données de classe mais les relations de table complexes sont également difficiles à interroger efficacement.

Prise en charge de Java pour la persistance

Depuis les débuts de la plate-forme Java, des interfaces de programmation existent pour fournir des passerelles dans la base de données et extraire de nombreux éléments spécifiques au domaine les exigences de persistance des applications métier. Les prochaines sections traitent des et les anciennes solutions de persistance Java et leur rôle dans les applications d'entreprise.

L'API Java Persistence se compose de quatre zones:

- L'API Java Persistence
- L'API Java Persistence Criteria
- Le langage de requête
- Métadonnées de mappage objet-relationnel

Solutions propriétaires

Il peut être surprenant d'apprendre que les solutions de cartographie relationnelle objet ont été autour depuis longtemps; plus longtemps que le langage Java lui-même. Produits tels que Oracle TopLink a fait ses débuts dans le monde Smalltalk avant de passer à Java. Un grand l'ironie dans l'histoire des solutions de persistance Java est que l'une des premières implémentations des beans entité a en fait été démontré en ajoutant une couche de bean entité supplémentaire sur Objets mappés TopLink.

Piste 32

Les deux API de persistance propriétaires les plus populaires étaient TopLink dans la publicité space et Hibernate dans la communauté open source. Produits commerciaux comme TopLink

étaient disponibles dans les premiers jours de Java et ont réussi, mais les techniques étaient
jamais standardisé pour la plate-forme Java. C'était plus tard, quand upstart open source
les solutions de mappage objet-relationnel telles que Hibernate sont devenues populaires, qui changent
autour de la persistance dans la plate-forme Java s'est produite, conduisant à une convergence vers
mappage objet-relationnel comme solution préférée.

Ces deux produits et d'autres pourraient être intégrés à toutes les principales applications
serveurs et a fourni aux applications toutes les fonctionnalités de persistance dont elles avaient besoin.
Les développeurs d'applications acceptaient d'utiliser un produit tiers pour leur persistance
besoins, d'autant plus qu'aucune norme commune et équivalente n'était en vue.

Mappeurs de données

Une approche partielle pour résoudre le problème relationnel objet consistait à utiliser des mappeurs de données.[1]
Le modèle de mappeur de données se situe dans l'espace entre le JDBC ordinaire (voir la section «JDBC»)
et une solution complète de mappage objet-relationnel car le développeur de l'application est
responsable de la création des chaînes SQL brutes pour mapper l'objet aux tables de la base de données, mais
un framework personnalisé ou prêt à l'emploi est généralement utilisé pour appeler le SQL à partir des données
méthodes de mappage. Le framework aide également avec d'autres choses comme le mappage de l'ensemble de résultats
et les paramètres de l'instruction SQL. Le framework de mappage de données le plus populaire était Apache
iBatis (maintenant nommé MyBatis et hébergé chez Google Code). Il a gagné une communauté importante
et se trouve toujours dans un certain nombre d'applications.

Le plus grand avantage de l'utilisation d'une stratégie de cartographie des données comme MyBatis est que
l'application a un contrôle complet sur le SQL qui est envoyé à la base de données. Stockée
les procédures et toutes les fonctionnalités SQL disponibles à partir du pilote sont toutes équitables, et
la surcharge ajoutée par le framework est plus petite que lors de l'utilisation d'un ORM à part entière
cadre. Cependant, le principal inconvénient de pouvoir écrire du SQL personnalisé est que
il doit être maintenu. Toute modification apportée au modèle objet peut avoir des répercussions
sur le modèle de données et peut-être entraîner une perte importante de SQL pendant le développement. UNE
Le cadre minimaliste ouvre également la porte au développeur pour créer de nouvelles fonctionnalités en tant que
les exigences des applications augmentent, conduisant finalement à une réinvention de la roue ORM. Les données
les mappeurs peuvent toujours avoir une place dans certaines applications s'ils sont sûrs que leurs besoins ne seront pas
vont au-delà du simple mappage, ou s'ils nécessitent un SQL très explicite qui ne peut pas être généré.

[1] Fowler, Martin. *Modèles d'architecture d'application d'entreprise* , Addison-Wesley, 2003.

Piste 33

JDBC

La deuxième version de la plate-forme Java, Java Development Kit (JDK) 1.1, publiée dans
1997, a inauguré le premier support majeur de la persistance des bases de données avec JDBC. C'était
créé comme une version spécifique à Java de son prédécesseur plus générique, la base de données d'objets
Spécification de connectivité (ODBC), un standard pour accéder à n'importe quelle base de données relationnelle
depuis n'importe quelle langue ou plateforme. Offrant une abstraction simple et portable du
interfaces de programmation client propriétaires offertes par les fournisseurs de bases de données, JDBC permet
Programmes Java pour interagir pleinement avec la base de données. Cette interaction repose fortement sur
SQL, offrant aux développeurs la possibilité d'écrire des requêtes et des instructions de manipulation de données
dans le langage de la base de données, mais exécuté et traité à l'aide d'un simple Java
modèle de programmation.

L'ironie de JDBC est que, bien que les interfaces de programmation soient portables, le
Le langage SQL ne l'est pas. Malgré les nombreuses tentatives de standardisation, il est encore rare d'écrire
SQL de toute complexité qui fonctionnera inchangé sur deux plates-formes de base de données principales. Même
où les dialectes SQL sont similaires, chaque base de données fonctionne différemment selon le
structure de la requête, nécessitant un réglage spécifique au fournisseur dans la plupart des cas.

Il y a aussi le problème du couplage étroit entre la source Java et le texte SQL.

Les développeurs sont constamment tentés par l'attrait des requêtes SQL prêtes à l'emploi.
construit dynamiquement au moment de l'exécution ou simplement stocké dans des variables ou des champs. C'est un
modèle de programmation très attractif jusqu'au jour où vous vous rendez compte que l'application a
pour prendre en charge un nouveau fournisseur de base de données qui ne prend pas en charge le dialecte SQL que vous avez
utilisé.

Même avec du texte SQL relégué aux fichiers de propriétés ou à d'autres métadonnées d'application,
vient un moment où travailler avec JDBC ne se sent pas seulement mal, mais devient simplement un
exercice fastidieux consistant à prendre des données tabulaires de lignes et de colonnes et à devoir continuellement
le convertir d'avant en arrière en objets. L'application possède un modèle objet - pourquoi
doit être si difficile à utiliser avec la base de données?

Enterprise JavaBeans

La première version de la plate-forme Java 2 Enterprise Edition (J2EE) a introduit un nouveau
solution de persistance Java sous la forme du bean entité, partie intégrante de l'Enterprise
Famille de composants JavaBean (EJB). Destiné à isoler complètement les développeurs de
traitant directement de la persistance, il a introduit une approche basée sur l'interface où le
La classe de bean concret n'a jamais été directement utilisée par le code client. Au lieu de cela, un haricot spécialisé

Piste 34

le compilateur a généré une implémentation de l'interface bean pour faciliter de telles choses
comme persistance, sécurité et gestion des transactions, en déléguant la logique métier à
l'implémentation du bean entité. Les beans entité ont été configurés à l'aide d'une combinaison de
descripteurs de déploiement XML standard et spécifiques au fournisseur, qui sont devenus notoires pour
leur complexité et leur verbosité.

Il est probablement juste de dire que les beans entité ont été sur-conçus pour le problème
ils essayaient de résoudre, mais ironiquement, la première version de la technologie manquait de nombreux
fonctionnalités nécessaires pour mettre en œuvre des applications professionnelles réalistes. Relations entre
les entités devaient être gérées par l'application, nécessitant le stockage des champs de clé étrangère
et géré sur la classe de haricots. Le mappage réel du bean entité à la base de données
a été entièrement réalisé en utilisant des configurations spécifiques au fournisseur, tout comme la définition des trouveurs
(le terme du bean entité pour les requêtes). Enfin, les beans entité ont été modélisés comme des objets distants
qui utilisaient RMI et CORBA, introduisant une surcharge du réseau et des restrictions qui devraient
n'ont jamais été ajoutés à un objet persistant pour commencer. Le bean entité a vraiment commencé
en résolvant le problème des composants persistants distribués, un remède pour lequel il y avait
pas de maladie, laissant derrière le cas commun de persistant léger accessible localement
objets.

La spécification EJB 2.0 a résolu de nombreux problèmes identifiés au début
communiqués. La notion de beans entité gérés par des conteneurs a été introduite, où bean
les classes sont devenues abstraites et le serveur était responsable de la génération d'une sous-classe à
gérer les données persistantes. Les interfaces locales et les relations gérées par le conteneur étaient
introduit, permettant de définir des associations entre les beans entité et automatiquement
maintenu cohérent par le serveur. Cette version a également vu l'introduction d'Enterprise
JavaBeans Query Language (EJB QL), un langage de requête conçu pour fonctionner avec des entités
qui pourrait être compilé de manière portable dans n'importe quel dialecte SQL.

Malgré les améliorations apportées avec EJB 2.0, un problème majeur demeure:
complexité excessive. La spécification supposait que les outils de développement isoleraient
le développeur du défi de la configuration et de la gestion des nombreux artefacts
requis pour chaque grain. Malheureusement, ces outils ont mis trop de temps à se matérialiser, et donc
le fardeau retombait carrément sur les épaules des développeurs, même si la taille et la portée de l'EJB
les demandes ont augmenté. Les développeurs se sentaient abandonnés dans une mer de complexité sans
a promis une infrastructure pour les maintenir à flot.

Piste 35

Objets de données Java

En partie à cause de certains des échecs du modèle de persistance EJB et d'une certaine frustration ne pas avoir une API de persistance standardisée satisfaisante, une autre spécification de persistance a été tenté. Java Data Objects (JDO) a été inspiré et pris en charge principalement par les fournisseurs de bases de données orientées objet (OODB) et n'a jamais vraiment été adopté par le communauté de programmation grand public. Il obligeait les fournisseurs à améliorer le bytecode de objets de domaine pour produire des fichiers de classe compatibles binaires entre tous les fournisseurs, et les produits de chaque fournisseur conforme devaient être capables de produire et de consommer leur. JDO disposait également d'un langage de requête résolument orienté objet par nature, qui ne convenait pas aux utilisateurs de bases de données relationnelles, qui étaient majorité.

JDO a atteint le statut d'être une extension du JDK, mais n'est jamais devenu un partie intégrante de la plate-forme Java d'entreprise. Il avait de nombreuses fonctionnalités et était adopté par une petite communauté d'utilisateurs dévoués qui s'en tenaient à lui et essayaient désespérément de promouvoir. Malheureusement, les principaux fournisseurs commerciaux ne partageaient pas le même point de vue comment un cadre de persistance doit être mis en œuvre. Peu ont soutenu la spécification, on parlait donc de JDO, mais rarement utilisé.

Certains pourraient soutenir qu'il était en avance sur son temps et que sa dépendance au bytecode l'amélioration a entraîné une stigmatisation inéquitable. C'était probablement vrai, et si c'était le cas été introduit trois ans plus tard, il aurait pu être mieux accepté par un développeur communauté qui ne pense plus à l'utilisation de frameworks largement utilisés de l'amélioration du bytecode. Une fois le mouvement de persistance EJB 3.0 en mouvement, cependant, et les principaux fournisseurs se sont tous inscrits pour faire partie de la nouvelle entreprise standard de persistance, l'écriture était sur le mur pour JDO. Les gens se sont rapidement plaints Sun qu'ils avaient maintenant deux spécifications de persistance: une qui faisait partie de son entreprise plate-forme et a également travaillé dans Java SE, et une qui n'était normalisée que pour Java SE. Peu de temps après, Sun a annoncé que JDO serait réduit aux spécifications mode de maintenance et que JPA tirerait à la fois de JDO et des fournisseurs de persistance et devenir la norme unique prise en charge à l'avenir.

Piste 36

Pourquoi une autre norme?

Les développeurs de logiciels savaient ce qu'ils voulaient, mais beaucoup ne pouvaient pas le trouver dans normes, ils ont donc décidé de chercher ailleurs. Ce qu'ils ont trouvé était une gamme de cadres de persistance, à la fois commerciaux et open source. Beaucoup de produits qui

mis en œuvre ces technologies ont adopté un modèle de persistance qui n'empiète pas sur
les objets du domaine. Pour ces produits, la persistance n'a pas été intrusive pour l'entreprise
objets en ce que, contrairement aux beans entité, ils n'avaient pas besoin d'être conscients de la technologie qui
les persistait. Ils n'ont eu à implémenter aucun type d'interface ni à étendre
une classe spéciale. Le développeur pourrait simplement traiter l'objet persistant comme n'importe quel autre
Objet Java, puis mappez-le à un magasin persistant et utilisez une API de persistance pour le conserver.
Comme les objets étaient des objets Java normaux, ce modèle de persistance est devenu connu
comme persistance POJO (Plain Old Java Object).

À mesure que Hibernate, TopLink et d'autres API de persistance se sont intégrés dans
applications et répondait parfaitement aux besoins de l'application, la question était souvent
a demandé: «Pourquoi mettre à jour la norme EJB pour qu'elle corresponde déjà à ce que ces produits
fait? Pourquoi ne pas continuer à utiliser ces produits comme cela se fait depuis des années, ou
pourquoi ne pas simplement standardiser sur un produit open source comme Hibernate? » Il y a
en fait de nombreuses raisons pour lesquelles cela n'a pas pu être fait - et ce serait une mauvaise idée même si
pourrait.

Une norme va bien plus loin qu'un produit, et un seul produit (même un produit
réussie comme Hibernate ou TopLink) ne peut pas incarner une spécification, même si elle peut
mettre en œuvre un. Au fond, l'intention d'une spécification est qu'elle soit implémentée
par différents fournisseurs et qu'il a différents produits offrent des interfaces standard et
sémantique qui peut être assumée par les applications sans coupler l'application à aucun
un produit.

Lier un standard à un projet open source comme Hibernate serait problématique
pour le standard et probablement encore pire pour le projet Hibernate. Imaginez un
spécification basée sur une version ou un point de contrôle spécifique de la base de code d'un
projet open source, et à quel point ce serait déroutant. Imaginez maintenant un open source
projet logiciel (OSS) qui ne pouvait pas changer ou qui ne pouvait changer que dans des versions discrètes
contrôlé par un comité spécial tous les deux ans, par opposition aux changements
décidé par le projet lui-même. Hibernate, et en fait tout projet open source,
probablement étouffé.

Piste 37

Bien que la normalisation puisse ne pas être appréciée par le consultant ou les cinq
boutique de logiciels, pour une entreprise, c'est énorme. Les technologies logicielles sont un gros investissement pour
la plupart des services informatiques d'entreprise, et le risque doit être mesuré lorsque de grosses sommes d'argent
sont impliqués. L'utilisation d'une technologie standard réduit considérablement ce risque et permet
société pour pouvoir changer de fournisseur si le choix initial ne répond pas aux
avoir besoin.

Outre la portabilité, la valeur de la standardisation d'une technologie se manifeste dans toutes sortes
d'autres domaines également. L'éducation, les modèles de conception et la communication avec l'industrie ne sont que
certains des nombreux avantages que les normes apportent à la table.

L'API Java Persistence

L'API Java Persistence est un framework léger basé sur POJO pour la persistance Java.
Bien que le mappage objet-relationnel soit un composant majeur de l'API, il offre également
solutions aux défis architecturaux de l'intégration de la persistance dans
applications de l'entreprise. Les sections suivantes examinent l'évolution de la spécification
et donner un aperçu des principaux aspects de cette technologie.

JPA n'est pas un produit mais simplement une spécification qui ne peut pas assurer la persistance par elle-même.
JPA nécessite bien sûr une base de données pour y persister.

Histoire de la spécification

L'API Java Persistence est remarquable non seulement pour ce qu'elle offre aux développeurs, mais aussi pour la manière dont il est venu à être. Les sections suivantes décrivent la préhistoire de l'objet solutions de persistance relationnelle et genèse de JPA.

EJB 3.0 et JPA 1.0

Après des années de plaintes concernant la complexité de la création d'applications d'entreprise avec Java, «facilité de développement» était le thème de la version de la plate-forme Java EE 5. EJB 3.0 a conduit le charge et a trouvé des moyens de rendre Enterprise JavaBeans plus facile et plus productif à utiliser.

Dans le cas des beans session et des beans orientés message, des solutions à la convivialité les problèmes ont été résolus en supprimant simplement certaines des implémentations les plus lourdes les exigences et laisser les composants ressembler davantage à des objets Java simples.

Piste 38

Dans le cas des beans entité, cependant, un problème plus sérieux existait. Si la définition de «facilité d'utilisation» est de garder les interfaces d'implémentation et les descripteurs hors de l'application code et pour adopter le modèle d'objet naturel du langage Java, comment faites-vous Les beans entité à granularité grossière, pilotés par l'interface et gérés par des conteneurs ressemblent à un modèle de domaine?

La réponse était de recommencer et de laisser les beans entité tranquilles et d'introduire à la place un nouveau modèle de persistance. L'API Java Persistence est née de la reconnaissance du les demandes des praticiens et les solutions propriétaires existantes qu'ils utilisaient pour résoudre leurs problèmes. Ignorer cette expérience aurait été une folie.

Ainsi, les principaux fournisseurs de solutions de cartographie relationnelle objet se sont manifestés et standardisé les meilleures pratiques représentées par leurs produits. Hibernate et TopLink ont été les premiers à se connecter avec les fournisseurs EJB, suivis plus tard par les fournisseurs JDO.

Des années d'expérience dans l'industrie associées à une mission de simplification du développement combiné pour produire la première spécification pour vraiment embrasser la nouvelle programmation modèles proposés par la plate-forme Java SE 5. L'utilisation d'annotations en particulier a conduit à une nouvelle façon d'utiliser la persistance dans des applications qui n'avait jamais été vue auparavant.

La spécification EJB 3.0 résultante, publiée en 2006, a finalement été divisée en trois pièces distinctes et réparties sur trois documents distincts. Le premier document contenait tout le contenu du modèle de composant EJB hérité, et le second décrit le nouveau modèle de composants POJO simplifié. Le troisième était l'API Java Persistence, un spécification autonome décrivant le modèle de persistance dans Java SE et Environnements Java EE.

Figure 1-8 montre JPA dans l'environnement Java EE.

Figure 1-8. *JPA dans l'environnement Java EE*

Piste 39

JPA 2.0

Au démarrage de la première version de JPA, la persistance ORM était déjà
évoluant depuis une décennie. Malheureusement, il n'y a eu qu'une période de temps relativement courte
disponible (environ deux ans) dans le cycle de développement des spécifications pour créer le
spécification initiale, de sorte que toutes les fonctionnalités possibles rencontrées ne pouvaient pas être
inclus dans la première version. Pourtant, un nombre impressionnant de fonctionnalités ont été spécifiées, avec
le reste est laissé pour les versions ultérieures et pour que les fournisseurs prennent en charge dans
en attendant.

La prochaine version, JPA 2.0, est devenue définitive en 2009 et a inclus un certain nombre de fonctionnalités
qui n'étaient pas présents dans la première version, en particulier ceux qui avaient été le plus
demandé par les utilisateurs. Ces nouvelles fonctionnalités comprenaient des capacités de cartographie supplémentaires, flexibles
les moyens de déterminer la manière dont le fournisseur a accédé à l'état de l'entité et les extensions du
Java Persistence Query Language (JP QL). La caractéristique la plus importante était probablement la
API Java Criteria, un moyen par programme de créer des requêtes dynamiques. Cela a principalement permis
frameworks pour utiliser JPA comme moyen de créer du code par programmation pour accéder aux données.

JPA 2.1

La sortie de JPA 2.1 en 2013 a permis à presque toutes les applications basées sur JPA de
être satisfait des fonctionnalités incluses dans la norme sans avoir à revenir au fournisseur
ajouts. Cependant, quel que soit le nombre de fonctionnalités spécifiées, il y aura toujours
être des applications qui nécessitent des capacités supplémentaires pour contourner des circonstances inhabituelles.
La spécification JPA 2.1 a ajouté certaines des fonctionnalités les plus exotiques, comme la cartographie
convertisseurs, prise en charge des procédures stockées et contextes de persistance non synchronisés pour
opérations conversationnelles améliorées. Il a également ajouté la possibilité de créer des graphiques d'entités
et les transmettre aux requêtes, ce qui équivaut à ce que l'on appelle communément un groupe de récupération
contraintes sur l'ensemble d'objets retourné.

JPA 2.2 et EJB 3.2

La version de maintenance JPA 2.2 a été publiée par Oracle en juin 2017. En général, les modifications
JPA 2.2, répertorié dans le fichier changelog, comprenait:

- Possibilité de diffuser le résultat d'une exécution de requête
- @Repeatable pour toutes les annotations pertinentes
- Prise en charge des types de date et d’heure Java 8 de base

Piste 40

- Permettre aux AttributeConverters de prendre en charge l’injection CDI
- Mise à jour du mécanisme de découverte du fournisseur de persistance
- Permettre à toutes les annotations JPA d'être utilisées dans les méta-annotations

Le fichier journal des modifications JPA 2.2 peut être trouvé ici:

https://jcp.org/aboutJava/communityprocess/maintenance/jsr338/ChangeLog-JPA- 2.2-MR.txt

Étant donné que JPA 2.2 n'est qu'une petite version, tout au long de ce livre, nous notons quand les nouvelles fonctionnalités

ajouté dans JPA 2.2 sera décrit, tandis que les autres feront toujours partie de JPA 2.1.

Depuis 2013, une version finale d'EJB 3.2 a également été développée dans le cadre de Java EE 7.

Les nouvelles fonctionnalités de la version Enterprise JavaBeans 3.2 (EJB 3.2) incluaient JNDI et EJB Lite.

JPA et vous

En fin de compte, il se peut qu'il y ait encore une fonctionnalité que vous, ou un autre utilisateur JPA, pourriez recherchez dans la norme qui n'a pas encore été incluse. Si la fonctionnalité est demandée par un nombre suffisant d'utilisateurs, il deviendra probablement éventuellement partie intégrante de la norme, mais cela dépend en partie des développeurs. Si vous pensez qu'une fonctionnalité doit être normalisée, vous devez en parler et en faire la demande auprès de votre fournisseur JPA; vous devriez également contacter le groupe d'experts de la prochaine version de JPA. La communauté contribue à façonner et à conduire le normes, et c'est vous, la communauté, qui devez faire connaître vos besoins.

Notez, cependant, qu'il y aura toujours un sous-ensemble de fonctionnalités rarement utilisées qui probablement jamais en faire la norme simplement parce qu'ils ne sont pas courants assez pour justifier son inclusion. La philosophie bien connue des «besoins des beaucoup "l'emportent sur les" besoins de quelques-uns "(ne prétendez même pas que vous ne connaissez pas épisode exact dans lequel cette philosophie a été exprimée pour la première fois) doit être considérée car chaque nouvelle fonctionnalité ajoute une complexité non nulle à la spécification, le rendant beaucoup plus grand et beaucoup plus difficile à comprendre, à utiliser et à implémenter. La leçon est que même si nous vous demandons votre contribution, tout cela ne peut pas être incorporé dans la spécification.

Aperçu

Le modèle de JPA est simple et élégant, puissant et flexible. Il est naturel d'utiliser et facile à apprendre, surtout si vous avez utilisé l'un des produits de persistance existants sur le marché aujourd'hui sur lequel reposait l'API. La principale API opérationnelle qu'une application sera exposé est contenu dans un petit nombre de classes.

> Remarque Si vous êtes intéressé par les outils Java Jpa open-source, j'ai fourni un court description de trois des plus populaires dans le cadre du téléchargement gratuit du code source disponible via www.apress.com/9781484234198 .

Persistance POJO

L'aspect le plus important de JPA est peut-être le fait que les objets sont des POJO, ce qui signifie qu'il n'y a rien de spécial dans tout objet qui est rendu persistant. En fait, presque tous l'objet d'application non final existant avec un constructeur par défaut peut être rendu persistant sans même changer une seule ligne de code. Mappage objet-relationnel avec JPA est entièrement basé sur les métadonnées. Cela peut être fait soit en ajoutant des annotations au code, soit en utilisant du XML défini en externe. Les objets persistants sont aussi lourds que les données qui est défini ou mis en correspondance avec eux.

Non intrusivité

L'API de persistance existe en tant que couche distincte des objets persistants. La persistance L'API est appelée par la logique métier de l'application, reçoit les objets de persistance et

est chargé de les opérer. Donc, même si l'application doit être consciente de
l'API de persistance car elle doit y faire appel, les objets persistants eux-mêmes ont besoin
pas être au courant. Parce que l'API n'empiète pas sur le code de l'objet persistant
classes, c'est ce qu'on appelle la persistance non intrusive.

Certaines personnes croient à tort que la persistance non intrusive signifie
que les objets deviennent magiquement persistants, comme les bases de données d'objets d'antan
faire quand une transaction a été validée. Ceci est parfois appelé persistance transparente
et c'est une notion incorrecte qui est encore plus irrationnelle lorsque vous pensez à l'interrogation.
Vous devez disposer d'un moyen de récupérer les objets du magasin de données. Cela nécessite

un objet API distinct et, en fait, certaines bases de données d'objets exigeaient que les utilisateurs invoquent
objets Etendue spéciaux pour émettre des requêtes. Les applications doivent absolument gérer leur
les objets persistants de manière très explicite, et ils nécessitent une API désignée pour le faire.

Requêtes d'objets

Un puissant framework de requêtes offre la possibilité d'interroger les entités et leurs
relations sans avoir à utiliser des clés étrangères concrètes ou des colonnes de base de données. Requêtes
peut être exprimé en JP QL, un langage de requête qui est modelé sur SQL pour sa familiarité
mais n'est pas lié au schéma de base de données ou défini à l'aide de l'API de critères. Utilisation des requêtes
une abstraction de schéma basée sur le modèle d'entité par opposition aux colonnes de
dans laquelle l'entité est stockée. Les entités Java et leurs attributs sont utilisés comme schéma de requête,
il n'est donc pas nécessaire de connaître les informations de mappage de la base de données. Les requêtes seront
éventuellement être traduit par l'implémentation JPA dans le SQL approprié pour le
base de données cible et exécutée sur la base de données.

En général, une entité est un objet de domaine de persistance léger.

En pratique, une entité est une table dans une base de données relationnelle où chaque instance d'entité
correspond à une certaine ligne de ce tableau.

Une requête peut être définie statiquement dans les métadonnées ou créée dynamiquement en passant
critères de requête lors de sa construction. Il est également possible d'échapper à SQL si une requête spéciale
il existe une exigence qui ne peut pas être satisfaite par la génération SQL à partir de la persistance
cadre. Ces requêtes peuvent renvoyer des résultats sous forme d'entités, de projections de
des attributs d'entité spécifiques, ou même des valeurs de fonction d'agrégation, entre autres options. JPA
les requêtes sont des abstractions précieuses qui permettent d'interroger sur le modèle de domaine Java
au lieu de sur des tables de base de données concrètes.

Entités mobiles

Les applications client / serveur et Web et autres architectures distribuées sont clairement
types d'applications les plus populaires dans un monde connecté. Pour reconnaître ce fait
signifie reconnaître que les entités persistantes doivent être mobiles dans le réseau. Objets
doit pouvoir être déplacé d'une machine virtuelle Java (JVM) à une autre, puis
à nouveau, et doit toujours être utilisable par l'application.

Les objets qui quittent la couche de persistance sont appelés détachés. Une caractéristique clé du
le modèle de persistance est la capacité de modifier les entités détachées, puis de les rattacher
à leur retour dans la JVM d'origine. Le modèle de détachement permet de

Piste 43

réconcilier l'état d'une entité en cours de rattachement avec l'état dans lequel elle se trouvait avant elle s'est détaché. Cela permet aux modifications d'entité d'être effectuées hors ligne tout en conservant cohérence des entités face à la concurrence.

Configuration simple

Il existe un grand nombre de fonctionnalités de persistance que la spécification a à offrir et que nous expliquerons dans les chapitres de ce livre. Toutes les fonctionnalités sont configurables via l'utilisation d'annotations, de XML ou d'une combinaison des deux. Les annotations offrent une facilité d'utilisation c'est sans précédent dans l'histoire des métadonnées Java. Ils sont pratiques à écrire et indolore à lire, et ils permettent aux débutants de lancer une application rapidement et facilement. La configuration peut également être effectuée en XML pour ceux qui aiment XML ou souhaitez externaliser les métadonnées du code.

Plus important que le langage des métadonnées est le fait que JPA rend utilisation des valeurs par défaut. Cela signifie que quelle que soit la méthode choisie, la quantité de les métadonnées qui seront nécessaires juste pour fonctionner sont le minimum absolu. Dans certaines cas, si les valeurs par défaut sont assez bonnes, presque aucune métadonnée ne sera requise du tout.

Intégration et testabilité

Les applications multiniveaux hébergées sur un serveur d'applications sont devenues de facto norme pour les architectures d'application. Tester sur un serveur d'applications est un défi que peu apprécient. Cela peut entraîner de la douleur et des difficultés, et il est souvent prohibitif de pratiquer tests unitaires et tests en boîte blanche.

Ceci est résolu en définissant l'API pour qu'elle fonctionne aussi bien à l'extérieur qu'à l'intérieur de l'application serveur. Bien que ce ne soit pas un cas d'utilisation aussi courant, les applications qui s'exécutent sur deux niveaux (l'application communiquant directement avec le niveau base de données) peut utiliser l'API de persistance sans l'existence d'un serveur d'application du tout. Le scénario le plus courant concerne les tests unitaires et des cadres de test automatisés qui peuvent être exécutés facilement et facilement dans Java SE environnements.

Avec l'API Java Persistence, il est désormais possible d'écrire code de persistance et être en mesure de le réutiliser pour des tests en dehors du serveur. En courant à l'intérieur d'un conteneur de serveur, tous les avantages du support de conteneur et une facilité supérieure de utilisez apply, mais avec quelques modifications et un peu de cadre de test, le support est le même l'application peut également être configurée pour s'exécuter en dehors du conteneur.

Piste 44

Résumé

Ce chapitre a présenté une introduction à l'API Java Persistence. Nous avons commencé par un introduction au problème principal auquel sont confrontés les développeurs essayant d'utiliser orienté objet modèles de domaine de concert avec une base de données relationnelle: le décalage d'impédance. À démontrer la complexité de combler le fossé, nous avons présenté trois petits modèles d'objets et neuf façons différentes de représenter la même information. Nous avons exploré chacun un peu et expliqué comment le mappage d'objets à différentes configurations de table peut entraîner des différences, non seulement dans la manière dont les données évoluent dans la base de données, mais aussi dans le coût des les opérations de base de données sont et comment l'application fonctionne.

Nous avons ensuite présenté un aperçu de certaines des solutions propriétaires et des normes de persistance, en regardant JDBC, EJB et JDO. Dans chaque cas, nous avons examiné le l'évolution de la norme et ses lacunes. Vous avez acquis des informations générales sur

aspects particuliers du problème de persistance qui ont été appris en cours de route.

Nous avons conclu le chapitre par un bref aperçu de JPA. Nous avons regardé l'histoire de la spécification et les fournisseurs qui se sont réunis pour le créer. Nous avons ensuite examiné le rôle il joue dans le développement d'applications d'entreprise et a donné une introduction à certains des caractéristiques offertes par la spécification.

Dans le chapitre suivant, vous vous mouillerez les pieds avec JPA en faisant une visite éclair de la les bases et la création d'une application Java Enterprise simple dans le processus.

CHAPITRE 2

Commencer

L'un des principaux objectifs de JPA était qu'il soit simple à utiliser et facile à comprendre. Bien que son domaine problématique ne puisse être banalisé ou édulcoré, la technologie qui permet aux développeurs de gérer cela peut être simple et intuitif. Ce chapitre montre à quel point il peut être facile de développer et d'utiliser des entités.

Nous commençons par décrire les caractéristiques de base des entités et passons en revue les exigences qu'une classe d'entité doit suivre. Nous définissons ce qu'est une entité et comment créer, lire, mettre à jour et supprimer un. Nous présentons également les gestionnaires d'entités et comment ils sont obtenus et utilisés. Ensuite, nous examinons rapidement les requêtes et expliquons comment spécifier et exécutez une requête à l'aide des objets EntityManager et Query. Le chapitre conclut en montrant une application de travail simple qui s'exécute dans un environnement Java SE standard et montre tout l'exemple de code en action.

Aperçu de l'entité

Une entité n'est pas une nouveauté dans la gestion des données. En fait, les entités existent plus long que de nombreux langages de programmation et certainement plus long que Java.

En général, une entité est une représentation Java de la table de base de données qui a des caractéristiques telles que la persistance, l'identité, la transaction et la granularité.

Ils ont été présentés par Peter Chen dans son article fondateur sur les relations d'entité la modélisation.[1] Ildécritentités commechoses qui ontattributs etrelations. le l'attente était que les attributs et les relations seraient persistants dans une relation

base de données.

Même maintenant, la définition est vraie. Une entité est essentiellement un nom ou un groupement de états associés ensemble en une seule unité. Il peut participer à des relations avec tout nombre d'autres entités de différentes manières. Dans le paradigme orienté objet,

[1] Peter P. Chen, «Le modèle de relation entité-vers une vue unifiée des données», ACM Transactions sur Database Systems 1, no. 1 (1976): 9–36.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_2

Piste 46

nous y ajoutons un comportement et l'appelons un objet. Dans JPA, tout objet défini par l'application peut être une entité, la question importante pourrait donc être celle-ci: quelles sont les caractéristiques d'un objet qui a été transformé en une entité?

Persistabilité

La première caractéristique et la plus fondamentale des entités est qu'elles sont persistantes. Cette signifie généralement simplement qu'ils peuvent être rendus persistants. Plus précisément, cela signifie que leur état peut être représenté dans un magasin de données et peut être consulté ultérieurement, peut-être bien après la fin du processus qui l'a créé.

Vous pouvez les appeler des objets persistants, comme beaucoup de gens le font, mais ce n'est pas techniquement correct. À proprement parler, un objet persistant devient persistant au moment où il instancié en mémoire. Si un objet persistant existe, alors par définition il l'est déjà persistant.

Une entité est persistante car elle peut être enregistrée dans un magasin persistant. La différence est qu'il n'est pas automatiquement persistant, et que pour qu'il ait un représentation, l'application doit invoquer activement une méthode API pour lancer la processus. Il s'agit d'une distinction importante car elle laisse fermement le contrôle sur la persistance entre les mains de l'application. L'application a la flexibilité de manipuler les données et exécutez la logique métier sur l'entité, la rendant persistante uniquement lorsque l'application décide que c'est le bon moment. La leçon est que les entités peuvent être manipulées sans forcément persistantes, et c'est l'application qui décide si elles le sont.

Identité

Comme tout autre objet Java, une entité a une identité d'objet, mais lorsqu'elle existe dans le base de données, il a également une identité persistante. L'identité d'objet est simplement la différenciation entre des objets qui occupent la mémoire. Une identité persistante, ou un identifiant, est la clé qui identifie de manière unique une instance d'entité et la distingue de toutes les autres instances du même type d'entité. Une entité a une identité persistante lorsqu'il existe un représentation de celui-ci dans le magasin de données; c'est-à-dire une ligne dans une table de base de données. Si ce n'est pas dans le base de données, alors même si l'entité en mémoire peut avoir son identité définie dans un champ, il n'a pas d'identité persistante. L'identifiant d'entité équivaut donc au clé primaire dans la table de base de données qui stocke l'état de l'entité.

Piste 47

Transactionnalité

Les entités peuvent être qualifiées de quasi-transactionnelles. Bien qu'ils puissent être créés, mis à jour, et supprimées dans n'importe quel contexte, ces opérations sont normalement effectuées dans le cadre de une transaction[2] car une transaction est requise pour que les modifications soient validées dans la base de données. Les modifications apportées à la base de données réussissent ou échouent de manière atomique. la vision persistante d'une entité doit en effet être transactionnelle.

En mémoire, c'est une histoire légèrement différente dans le sens où les entités peuvent être changées sans que les changements ne soient jamais persistés. Même lorsqu'ils sont enrôlés dans une transaction, ils peut être laissé dans un état indéfini ou incohérent en cas d'annulation ou de transaction échec. Les entités en mémoire sont de simples objets Java qui obéissent à toutes les règles et contraintes appliquées par la machine virtuelle Java (JVM) à d'autres objets Java.

Granularité

Enfin, un bon moyen de montrer ce que sont les entités est de décrire ce qu'elles ne sont pas. Elles sont pas des primitives, des wrappers primitifs ou des objets intégrés avec un état unidimensionnel. Ce ne sont que des scalaires et n'ont aucune signification sémantique inhérente à un application. Une chaîne, par exemple, est un objet trop fin pour être une entité car elle n'a pas de connotation spécifique au domaine. Plutôt, une corde est bien adaptée et très souvent utilisé comme type pour un attribut d'entité et donné une signification selon l'entité attribut qu'il tape.

Les entités sont censées être des objets à granularité fine qui ont un ensemble de états normalement stockés dans un seul endroit, comme une ligne dans une table, et ont généralement relations avec d'autres entités. Au sens le plus général, ils sont du domaine commercial objets qui ont une signification spécifique pour l'application qui y accède.

S'il est certainement vrai que les entités peuvent être définies de manière exagérée pour être aussi fin que le stockage d'une seule corde ou assez gros pour contenir 500 colonnes de données, les entités JPA étaient définitivement destinées à être plus petites du spectre de granularité. Idéalement, les entités devraient être conçues et définies comme équitablement objets légers d'une taille comparable à celle de l'objet Java moyen.

[2] Dans la plupart des cas, il s'agit d'une exigence, mais dans certaines configurations, la transaction peut ne pas commencé jusqu'après l'opération.

Piste 48

Métadonnées d'entité

En plus de son état persistant, chaque entité JPA a des métadonnées associées (même si une très petite quantité) qui le décrit. Ces métadonnées peuvent exister dans le cadre de la sauvegarde class ou il peut être stocké à l'extérieur de la classe, mais il n'est pas conservé dans la base de données. Il permet à la couche de persistance de reconnaître, d'interpréter et de gérer correctement l'entité depuis le moment où il est chargé jusqu'à son appel à l'exécution.

Les métadonnées réellement requises pour chaque entité sont minimales, rendant les entités facile à définir et à utiliser. Cependant, comme toute technologie sophistiquée avec sa part de commutateurs, leviers et boutons, il y a aussi la possibilité de spécifier beaucoup, beaucoup plus métadonnées que nécessaire. Il peut s'agir de montants importants, selon l'application exigences, et peut être utilisé pour personnaliser chaque détail de la configuration de l'entité ou mappages d'état.

Les métadonnées d'entité peuvent être spécifiées de deux manières: annotations ou XML. Chacun est également valide, mais celui que vous utiliserez dépendra de vos préférences ou processus de développement.

Annotations

Les métadonnées d'annotation sont une fonctionnalité de langage introduite dans Java SE 5 qui permet
et des métadonnées typées à joindre au code source. Bien que les annotations soient
non requis par JPA, ils constituent un moyen pratique d'apprendre et d'utiliser l'API. Parce que
les annotations co-localisent les métadonnées avec les artefacts du programme, il n'est pas nécessaire de
échapper à un fichier supplémentaire et à un langage spécial (XML) juste pour spécifier les métadonnées.

Les annotations sont utilisées tout au long des exemples et des explications qui les accompagnent
dans ce livre. Toutes les annotations JPA affichées et décrites (sauf au chapitre3,
qui parle des annotations Java EE) sont définies dans le package javax.persistence.
Les exemples d'extraits de code peuvent être supposés avoir une importation implicite de l'importation de formulaire
javax.persistence. * ;.

Comme nous l'avons dit au chapitre 1, l'une des nouvelles fonctionnalités de JPA 2.2 consiste à créer une annotation
@ Répétable. Il s'agit en fait d'un changement de Java 8 et il a été suivi dans le numéro 115 de la spécification JPA.

Cette nouvelle fonctionnalité JPA 2.2 permettra aux développeurs d'utiliser le même multiple d'annotation
fois pour une certaine classe ou attribut sans utiliser d'annotation de conteneur. Cette fonctionnalité
vous aidera en rendant votre code Java beaucoup plus facile à lire.

En pratique, vous pourrez annoter n'importe laquelle de vos classes d'entités avec plusieurs
Annotations @NamedQuery sans avoir besoin de les envelopper dans une annotation @NamedQueries.

Piste 49

Voici les annotations qui peuvent être répétées lors de l'utilisation de JPA 2.2:

- AssociationOverride
- AttributeOverride
- Convertir
- JoinColumn
- MapKeyJoinColumn
- NamedEntityGraph
- NamedNativeQuery
- Requête nommée
- NamedStoredProcedureQuery
- PersistenceContext
- PersistenceUnit
- PrimaryKeyJoinColumn
- Table secondaire
- SqlResultSetMapping

Prenons un de la liste, par exemple l'annotation AssociationOverride, qui
est utilisé pour remplacer un mappage pour une relation d'entité.

Voici comment fonctionne l'annotation AssociationOverride. Remarquez le
@Repeatable (AssociationOverrides.class):

```
@Target ({TYPE, METHOD, FIELD}) @Retention (RUNTIME)
@Repeatable (AssociationOverrides.class)
public @interface AssociationOverride {
Nom de chaîne ();
JoinColumn [] joinColumns () default {};
ForeignKey ForeignKey () default @ForeignKey (PROVIDER_DEFAULT);
JoinTable joinTable () default @JoinTable;
}
```

Piste 50

XML

Pour ceux qui préfèrent utiliser XML traditionnel, cette option est toujours disponible. Cela devrait être assez simple pour passer à l'utilisation des descripteurs XML après avoir appris et compris les annotations, car le XML a principalement été modelé après le annotations. Chapitre12 décrit comment utiliser XML pour spécifier ou remplacer le mappage d'entités métadonnées.

Configuration par exception

La notion de configuration par exception signifie que le moteur de persistance définit les valeurs par défaut qui s'appliquent à la majorité des applications et dont les utilisateurs doivent fournir des valeurs uniquement lorsqu'ils souhaitent remplacer la valeur par défaut. En d'autres termes, devoir fournir un La valeur de configuration est une exception à la règle, pas une exigence.

La configuration par exception est ancrée dans JPA et contribue fortement à sa utilisabilité. La plupart des valeurs de configuration ont des valeurs par défaut, ce qui rend les métadonnées qui doivent être précisés de manière plus pertinente et concise.

L'utilisation étendue des valeurs par défaut et la facilité d'utilisation qu'elle apporte à la configuration viennent à un prix, cependant. Lorsque les valeurs par défaut sont intégrées à l'API et n'ont pas à être spécifiés, alors ils ne sont ni visibles ni évidents pour les utilisateurs. Cela peut permettre aux utilisateurs ne pas être conscient de la complexité du développement d'applications de persistance, ce qui rend la tâche plus difficile pour déboguer ou pour changer le comportement quand cela devient nécessaire.

Les valeurs par défaut ne visent pas à protéger les utilisateurs des problèmes souvent complexes persistance. Ils sont destinés à permettre à un développeur de démarrer facilement et rapidement avec quelque chose qui fonctionnera, puis améliorera et implémentera de manière itérative fonctionnalité à mesure que la complexité de leur application augmente. Même si le les valeurs par défaut peuvent être ce que vous voulez voir arriver la plupart du temps, c'est toujours important pour les développeurs doivent se familiariser avec les valeurs par défaut qui sont appliquées. Par exemple, si un nom de table par défaut est supposé, il est important de savoir de quelle table est le runtime attente, ou si la génération de schéma est utilisée, quelle table sera générée.

Pour chacune des annotations, nous discutons également de la valeur par défaut afin que ce soit clair sera appliquée si l'annotation n'est pas spécifiée. Nous vous recommandons de vous souvenir ces valeurs par défaut au fur et à mesure que vous les apprenez. Après tout, une valeur par défaut fait toujours partie de la configuration de l'application; c'est vraiment simple à configurer!

Piste 51

Créer une entité

Les classes Java standard sont facilement transformées en entités simplement en les annotant. Dans En fait, en ajoutant quelques annotations, presque toutes les classes avec un constructeur sans argument peuvent

devenir une entité.

L'artefact de programmation principal d'une entité est la classe d'entité.

Voici les exigences qu'une classe d'entité doit suivre:

- Doit être annoté avec l'annotation javax.persistence.Entity
- Doit avoir un constructeur sans argument public ou protégé (il peut avoir d'autres constructeurs)
- Ne doit pas être déclaré définitif (par conséquent, aucune méthode ou persistance les variables d'instance peuvent être déclarées finales)
- Doit implémenter l'interface sérialisable (au cas où l'entité instance est passée par valeur en tant qu'objet détaché via un bean session interface professionnelle à distance)
- Peut étendre les classes d'entité et de non-entité, et les classes de non-entité peut étendre les classes d'entités
- Les variables d'instance persistantes doivent être déclarées privées, protégées ou package-private (ils ne sont accessibles directement que par la classe d'entité méthodes)

Commençons par créer une classe Java standard pour un employé. Le listing 2-1 montre un simple Classe d'employé.

Liste 2-1. Classe d'employé

```
Employé de classe publique {
    id int privé;
    nom de chaîne privé;
    long salaire privé;

    employé public () {}
    employé public (int id) {this.id = id; }

    public int getId () {identifiant de retour; }
    public void setId (int id) {this.id = id; }
```



```
    public String getName () {nom de retour; }
    public void setName (String name) {this.name = nom; }
    public long getSalary () {salaire de retour; }
    public void setSalary (salaire long) {this.salary = salaire; }
}
```

Vous remarquerez peut-être que cette classe ressemble à une classe de style JavaBean avec trois propriétés: identifiant, nom et salaire. Chacune de ces propriétés est représentée par une paire d'accesseurs méthodes pour obtenir et définir la propriété et est soutenu par un champ membre. Propriétés ou Les champs membres sont les unités d'état au sein de l'entité qui peuvent être persistantes.

Pour transformer Employee en une entité, nous annotons d'abord la classe avec @Entity. C'est principalement juste une annotation de marqueur pour indiquer au moteur de persistance que la classe est une entité.

La deuxième annotation que nous devons ajouter est @Id. Cela annote le particulier champ ou propriété qui contient l'identité persistante de l'entité (la clé primaire) et est nécessaire pour que le fournisseur sache quel champ ou quelle propriété utiliser comme identification unique clé dans le tableau.

En ajoutant ces deux annotations à la classe Employee, nous nous retrouvons à peu près même classe que nous avions avant, sauf que maintenant c'est une entité. Référencement2-2 montre l'entité classe.

Liste 2-2. Entité des employés

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;

    employé public () {}
    employé public (int id) {this.id = id; }

    public int getId () {identifiant de retour; }
    public void setId (int id) {this.id = id; }
    public String getName () {nom de retour; }
    public void setName (String name) {this.name = nom; }
    public long getSalary () {salaire de retour; }
    public void setSalary (salaire long) {this.salary = salaire; }
}
```


Piste 53

Quand on dit que l'annotation @Id est placée sur le champ ou la propriété, on entend que l'utilisateur peut choisir d'annoter soit le champ déclaré, soit la méthode getter[3] d'un Propriété de style JavaBean. La stratégie de terrain ou de propriété est autorisée, selon le besoins et goûts du développeur de l'entité. Nous avons choisi dans cet exemple d'annoter le champ parce que c'est plus simple; en général, ce sera l'approche la plus simple et la plus directe. Nous discutons des détails de l'annotation de l'état persistant à l'aide de l'accès aux champs ou aux propriétés dans chapitres suivants.

Les champs de l'entité sont automatiquement rendus persistants du fait de leur existence dans l'entité. Le mappage par défaut et les valeurs de configuration de chargement s'appliquent à ces champs et permettre leur persistance lorsque l'objet est persistant. Compte tenu des questions qui étaient évoquée dans le dernier chapitre, on pourrait être amené à se demander: «Comment les champs ont-ils été cartographiés, et vers où persistent-ils? »

Pour trouver la réponse, il faut d'abord faire un petit détour pour creuser à l'intérieur de @Entity annotation et regardez un élément appelé nom qui identifie de manière unique l'entité. le Le nom d'entité peut être spécifié explicitement pour toute entité en utilisant cet élément de nom dans le annotation, comme dans @Entity (name = "Emp"). En pratique, cela est rarement spécifié car il obtient par défaut le nom non qualifié de la classe d'entité. C'est presque toujours les deux raisonnable et adéquat.

Nous pouvons maintenant revenir à la question de savoir où les données sont stockées. Il s'avère que le nom par défaut de la table utilisé pour stocker une entité donnée d'un type d'entité particulier est le nom de l'entité. Si nous avons spécifié le nom de l'entité, ce sera la valeur par défaut nom de la table; sinon, la valeur par défaut du nom d'entité sera utilisée. Nous venons de dire que le nom d'entité par défaut était le nom non qualifié de la classe d'entité, donc c'est effectivement la réponse à la question de savoir quelle table est utilisée. Dans l'exemple Employee, l'entité nom sera par défaut "Employé" et toutes les entités de type Employé seront stockées dans un table appelée EMPLOYÉ.

Chacun des champs ou propriétés a un état individuel et doit être orienté à une colonne particulière du tableau. On sait aller à la table EMPLOYEE, mais qui doit être utilisée pour un champ ou une propriété donné? Lorsqu'aucune colonne n'est explicitement spécifié, la colonne par défaut est utilisée pour un champ ou une propriété, qui est juste le nom du champ ou propriété elle-même. Ainsi, l'ID d'employé sera stocké dans la colonne ID, le nom dans la colonne NOM et le salaire dans la colonne SALARY de la table EMPLOYEE.

3 Les annotations sur les méthodes de setter seront simplement ignorées.

Piste 54

Bien entendu, ces valeurs peuvent toutes être remplacées pour correspondre à un schéma existant. nous discuter de la façon de les remplacer lorsque nous arrivons au chapitre 4 et discuter de la cartographie en plus détail.

Gestionnaire d'entités

Dans la section "Présentation de l'entité", il a été indiqué qu'un appel d'API spécifique doit être appelé avant qu'une entité ne soit réellement conservée dans la base de données. En fait, les appels d'API séparés sont nécessaire pour effectuer de nombreuses opérations sur les entités. Cette API est implémentée par le gestionnaire d'entités et encapsulé presque entièrement dans une seule interface appelée javax. persistence.EntityManager. En fin de compte, c'est à un gestionnaire d'entité qui le vrai travail de persévérance est délégué. Jusqu'à ce qu'un gestionnaire d'entités soit habitué à créer, lire ou écrire une entité, l'entité n'est rien de plus qu'un régulier (non persistant) Objet Java.

Lorsqu'un gestionnaire d'entités obtient une référence à une entité, soit en la faisant explicitement passé en argument à un appel de méthode ou parce qu'il a été lu à partir de la base de données, que l'objet est dit géré par le gestionnaire d'entités. L'ensemble des instances d'entités gérées dans un gestionnaire d'entités à un moment donné est appelé son contexte de persistance. Seulement un Une instance Java avec la même identité persistante peut exister dans un contexte de persistance à tout temps. Par exemple, si un employé avec une identité persistante (ou ID) de 158 existe dans le contexte de persistance, alors aucun autre objet Employé avec son ID défini sur 158 ne peut exister dans ce même contexte de persistance.

Les gestionnaires d'entités sont configurés pour pouvoir conserver ou gérer des types spécifiques d'objets, lire et écrire dans une base de données donnée, et être implémenté par un fournisseur de persistance (ou fournisseur pour faire court). C'est le fournisseur qui fournit le support moteur d'implémentation pour l'ensemble de l'API Java Persistence, depuis EntityManager jusqu'à l'implémentation des classes de requêtes et de la génération SQL.

Tous les gestionnaires d'entités proviennent d'usines de type javax.persistence. EntityManagerFactory. La configuration d'un gestionnaire d'entités est basée sur le modèle usine de gestionnaire d'entités qui l'a créé, mais il est défini séparément en tant qu'unité de persistance. UNE l'unité de persistance dicte implicitement ou explicitement les paramètres et les classes d'entités utilisés par tous les gestionnaires d'entités obtenus à partir de l'instance unique EntityManagerFactory liée à cette unité de persistance. Il existe donc une correspondance biunivoque entre un unité de persistance et son instance EntityManagerFactory concrète.

Piste 55

Les unités de persistance sont nommées pour permettre la différenciation d'une fabrique de gestionnaires d'entités D'un autre. Cela donne à l'application le contrôle sur la configuration ou la persistance l'unité doit être utilisée pour opérer sur une entité particulière.

Figure 2-1 montre que pour chaque unité de persistance il existe une fabrique de gestionnaire d'entités et que de nombreux gestionnaires d'entités peuvent être créés à partir d'une seule usine de gestionnaires d'entités. le Une partie qui peut surprendre est que de nombreux responsables d'entités peuvent pointer vers le même contexte de persistance. Nous n'avons parlé que d'un gestionnaire d'entité et de sa persistance contexte, mais plus tard vous verrez qu'il peut en fait y avoir plusieurs références à différents

les gestionnaires d'entités, tous pointant vers le même groupe d'entités gérées. Cela permettra le flux de contrôle pour traverser les composants du conteneur mais continuer à accéder au même contexte de persistance.

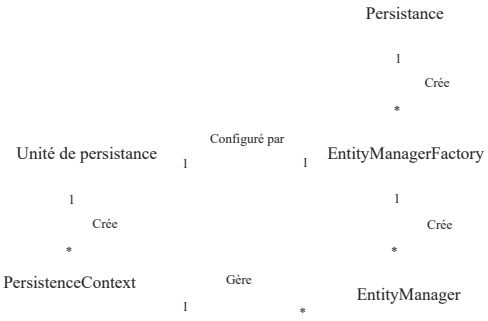

Figure 2-1. *Relations entre les concepts JPA*

Le tableau 2-1 résume les concepts et objets API mentionnés précédemment ou discuté. Notez que si certains sont de véritables objets API, d'autres ne sont que des concepts abstraits qui aident à expliquer le fonctionnement de l'API.

Tableau 2-1. *Résumé des objets et concepts API*

Objet	Objet API	La description
persistance	persistance	Classe Bootstrap utilisée pour obtenir une entité directeur de l'usine
Entité Manager Factory	entityManagerFactory	Objet d'usine configuré utilisé pour obtenir gestionnaires d'entités
Unité de persistance	-	configuration nommée déclarant l'entité informations sur les classes et le magasin de données
Gestionnaire d'entité	entityManager	Objet API principal utilisé pour effectuer des opérations et requêtes sur les entités
Contexte de persistance	-	Ensemble de toutes les instances d'entité gérées par un gestionnaire d'entité spécifique

Obtention d'un gestionnaire d'entités

Un gestionnaire d'entités est toujours obtenu à partir d'une EntityManagerFactory. le l'usine à partir de laquelle il a été obtenu détermine les paramètres de configuration qui régir son fonctionnement. Bien qu'il existe des raccourcis qui masquent l'usine de l'utilisateur vue lors de l'exécution dans un environnement de serveur d'applications Java EE, dans Java SE environnement, nous pouvons utiliser une classe d'amorçage simple appelée Persistence. Le statique La méthode createEntityManagerFactory () de la classe Persistence renvoie le EntityManagerFactory pour le nom d'unité de persistance spécifié. L'exemple suivant

illustre la création d'un EntityManagerFactory pour l'unité de persistance nommée EmployéService:

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory ("EmployeeService");
```

Le nom de l'unité de persistance spécifiée EmployeeService transmis au
La méthode createEntityManagerFactory () identifie l'unité de persistance donnée
configuration qui détermine des éléments tels que les paramètres de connexion de cette entité
les gestionnaires générés à partir de cette fabrique utiliseront lors de la connexion à la base de données.

Psaumes 57

Maintenant que nous avons une usine, nous pouvons facilement en obtenir un gestionnaire d'entité. le
l'exemple suivant illustre la création d'un gestionnaire d'entités à partir de l'usine acquise dans
l'exemple précédent:

```
EntityManager em = emf.createEntityManager ();
```

Avec ce gestionnaire d'entités, nous sommes en mesure de commencer à travailler avec des entités persistantes.

Persistance d'une entité

La persistance d'une entité est l'opération consistant à prendre une entité transitoire, ou une entité qui n'a pas encore
avoir une représentation persistante dans la base de données et stocker son état afin qu'il puisse être
récupéré plus tard. C'est vraiment la base de la persistance - créer un état qui peut survivre
le processus qui l'a créé. Nous commençons par utiliser le gestionnaire d'entités pour conserver une instance de
Employé. Voici un exemple de code qui fait exactement cela:

```
Employé emp = nouvel employé (158);
em.persist (emp);
```

La première ligne de ce segment de code crée simplement une instance Employee qui
nous voulons persister. Si nous ignorons le triste fait d'employer un individu sans nom et
ne rien lui payer (nous définissons uniquement l'ID, pas le nom ou le salaire), l'instancié
L'employé n'est qu'un objet Java standard.

La ligne suivante utilise le gestionnaire d'entités pour conserver l'entité. L'appel de persist () est
tout ce qui est nécessaire pour l'initier étant conservé dans la base de données. Si le responsable de l'entité
rencontre un problème en faisant cela, il lèvera une PersistenceException non cochée.
Lorsque l'appel persist () sera terminé, emp deviendra une entité gérée au sein du
contexte de persistance du gestionnaire d'entités.

Référencement 2-3 montre comment incorporer cela dans une méthode simple qui crée un nouveau
employé et le persiste dans la base de données.

Liste 2-3. Méthode de création d'un employé

```
public Employee createEmployee (int id, nom de chaîne, salaire long) {
    Employé emp = nouvel employé (id);
    emp.setName (nom);
    emp.setSalary (salaire);
```

```
        em.persist (emp);
        return emp;
}
```

Cette méthode suppose l'existence d'un gestionnaire d'entités dans le champ em du
instance et l'utilise pour conserver l'employé. Notez que nous n'avons pas à nous soucier de
le cas d'échec dans cet exemple. Il en résultera une exception PersistenceException d'exécution
jeté, qui sera propagé jusqu'à l'appelant.

Trouver une entité

Une fois qu'une entité est dans la base de données, la prochaine chose que l'on veut généralement faire est de la retrouver.
Dans cette section, vous découvrez comment une entité peut être trouvée à l'aide du gestionnaire d'entités. Là
est vraiment une seule ligne qui est importante:

```
Employé emp = em.find (Employee.class, 158);
```

Nous passons dans la classe de l'entité recherchée (dans cet exemple, nous
recherchent une instance de Employee) et l'ID ou la clé primaire qui identifie le
entité particulière (dans ce cas, nous voulons trouver l'entité que nous venons de créer). C'est
toutes les informations nécessaires au gestionnaire d'entités pour retrouver l'instance dans la base de données,
et une fois l'appel terminé, l'employé renvoyé sera un
entité, ce qui signifie qu'elle existera dans le contexte de persistance actuel associé au
gestionnaire d'entité. Passer la classe comme paramètre permet également à la méthode find d'être
paramétré et pour renvoyer un objet du même type qui a été passé, en sauvegardant l'appelant
un casting supplémentaire.

Que se passe-t-il si l'objet a été supprimé ou si vous fournissez le mauvais ID
accident? Dans le cas où l'objet n'a pas été trouvé, l'appel find () retourne simplement
nul. Vous devez vous assurer qu'une vérification nulle est effectuée avant la prochaine fois
La variable emp est utilisée.

Le code d'une méthode qui recherche et renvoie l'employé avec un ID donné est maintenant
trivial et est montré dans la liste 2-4.

Liste 2-4. Méthode de recherche d'un employé

```
public Employee findEmployee (int id) {
        return em.find (Employee.class, id);
}
```


Dans le cas où aucun employé n'existe pour l'ID qui est transmis, la méthode
retourne null car c'est ce que find () retournera.

Suppression d'une entité

La suppression d'une entité de la base de données n'est pas aussi courante qu'on pourrait le penser. Beaucoup
les applications ne suppriment jamais les objets, ou si elles le font, elles signalent simplement que les données sont obsolètes
ou n'est plus valide et gardez-le simplement hors de vue des clients. Nous ne parlons pas de
ce type de suppression logique au niveau de l'application où les données ne sont même pas supprimées
la base de données. Nous parlons de quelque chose qui entraîne une instruction DELETE
fait sur une ou plusieurs tables.

Afin de supprimer une entité, l'entité elle-même doit être gérée, c'est-à-dire qu'elle est

présent dans le contexte de persistance. Cela signifie que l'application appelante doit avoir déjà chargé ou accédé à l'entité et émet maintenant une commande pour la supprimer. Cette n'est normalement pas un problème étant donné que le plus souvent l'application l'a amené à devenir géré dans le cadre du processus visant à déterminer que c'était l'objet qu'il voulait supprimer.

Un exemple simple de suppression d'un employé est le suivant:

```
Employé emp = em.find (Employee.class, 158);
em.remove (emp);
```

Dans cet exemple, nous recherchons d'abord l'entité à l'aide de l'appel find (), qui renvoie une instance gérée de Employee, puis suppression de l'entité à l'aide de l'appel remove () sur le gestionnaire d'entités. Bien sûr, vous avez appris dans la section précédente que si l'entité n'a pas été trouvé, la méthode find () retournera null. Vous obtiendrez un java.lang. IllegalArgumentException s'il s'est avéré que vous avez passé null dans l'appel remove () car vous avez oublié d'inclure une vérification nulle avant d'appeler remove ().

Dans la méthode d'application pour supprimer un employé, le problème peut être résolu par vérifier l'existence de l'employé avant d'émettre l'appel remove (), comme indiqué dans Référencement 2-5.

Liste 2-5. Méthode de suppression d'un employé

```
public void removeEmployee (int id) {
    Employé emp = em.find (Employee.class, id);
    if (emp! = null) {
```


Piste 60


```
        em.remove (emp);
    }
}
```

Cette méthode garantira que l'employé avec l'ID donné, à condition que l'ID ne soit pas null, est supprimé de la base de données. Il reviendra avec succès si l'employé existe ou pas.

Mettre à jour une entité

Il existe plusieurs façons de mettre à jour une entité, mais pour l'instant nous allons illustrer cas le plus simple et le plus courant. C'est là que nous avons une entité gérée et que nous voulons apporter des modifications. Si nous n'avons pas de référence à l'entité gérée, nous devons commencez par en obtenir un en utilisant find (), puis effectuez nos opérations de modification sur le entité. Le code suivant ajoute 1000 $ au salaire de l'employé avec un ID de 158:

```
Employé emp = em.find (Employee.class, 158);
emp.setSalary (emp.getSalary () + 1000);
```

Notez la différence entre cette opération et les autres. Dans ce cas, nous ne sommes pas appeler le gestionnaire d'entités pour modifier l'objet, mais appeler directement l'objet lui-même. Pour cette raison, il est important que l'entité soit une instance gérée; sinon, le le fournisseur de persistance n'aura aucun moyen de détecter le changement, et aucun changement ne sera fait à la représentation persistante de l'employé.

La méthode pour augmenter le salaire d'un employé donné prendra l'ID et le montant de l'augmentation, trouvez l'employé et changez le salaire pour le salaire ajusté. Référencement2-6 démontre cette approche.

Liste 2-6. Méthode de mise à jour d'un employé

```
public Employee riseEmployeeSalary (int id, longue augmentation) {
    Employé emp = em.find (Employee.class, id);
    if (emp! = null) {
        emp.setSalary (emp.getSalary () + augmenter);
    }
    return emp;
}
```


Piste 61

Si nous ne pouvons pas trouver l'employé, nous retournons null pour que l'appelant sache que non des changements pourraient être apportés. Nous indiquons le succès en renvoyant l'employé mis à jour.

Transactions

Vous pouvez penser que le code jusqu'à présent semble incompatible avec ce que nous avons dit plus tôt à propos de transactionnel lorsque vous travaillez avec des entités. Il n'y a eu aucune transaction dans aucun des les exemples précédents, même si nous avons dit que des modifications des entités doivent être apportées persistant à l'aide d'une transaction.

Dans tous les exemples à l'exception de celui qui n'a appelé que find (), nous supposons qu'un transaction inclus chaque méthode. L'appel find () n'est pas une opération de mutation, il peut donc être appelé à tout moment, avec ou sans transaction.

Encore une fois, la clé est l'environnement dans lequel le code est exécuté. le situation typique lors de l'exécution dans l'environnement de conteneur Java EE est que le Java Transaction API (JTA) standard est utilisé. Le modèle de transaction lors de l'exécution dans le conteneur doit supposer que l'application s'assurera qu'un contexte transactionnel est présent quand on en a besoin. Si une transaction n'est pas présente, alors soit l'opération de modification lèvera une exception ou la modification ne sera tout simplement jamais conservée dans le magasin de données. Nous revenons à discuter des transactions dans l'environnement Java EE plus en détail dans Chapitre 3.

Dans l'exemple de ce chapitre, cependant, nous ne fonctionnons pas en Java EE. C'était dans un L'environnement Java SE et le service de transaction à utiliser dans Java SE est le javax.persistence.EntityTransaction service. Lors de l'exécution dans Java SE, nous besoin de commencer et de valider la transaction dans les méthodes opérationnelles, ou nous devons commencer et valider la transaction avant et après l'appel d'une méthode opérationnelle. Dans dans les deux cas, une transaction est lancée en appelant getTransaction () sur le gestionnaire d'entités pour obtenir EntityTransaction, puis en invoquant begin () dessus. De même, pour engager le transaction, l'appel commit () est invoqué sur l'objet EntityTransaction obtenu du gestionnaire d'entité. Par exemple, démarrer et valider avant et après le La méthode produirait du code qui crée un employé comme cela est fait dans le Listing 2-7 .

Liste 2-7. Début et validation d'une transaction d'entité

```
em.getTransaction (). begin ();
createEmployee (158, "John Doe", 45000);
em.getTransaction (). commit ();
```


Piste 62

Plus de détails sur les transactions au niveau des ressources et l'API EntityTransaction sont contenus dans le chapitre 6 .

Requêtes

En général, étant donné que la plupart des développeurs ont utilisé une base de données relationnelle à un moment donné ou un autre dans leur vie, la plupart d'entre nous savent à peu près ce qu'est une requête de base de données. Dans JPA, une requête est similaire à une requête de base de données, sauf qu'au lieu d'utiliser une requête structurée Langage (SQL) pour spécifier les critères de requête, nous interrogeons les entités et utilisons un langage appelé Java Persistence Query Language (JP QL).

Une requête est implémentée dans le code en tant qu'objet Query ou TypedQuery <X>. Il est construit en utilisant EntityManager en tant que fabrique. L'interface EntityManager comprend une variété de Appels d'API qui renvoient un nouvel objet Query ou TypedQuery <X>. En tant qu'objet de première classe, une requête peut à son tour être personnalisé en fonction des besoins de l'application.

Notez que JPA version 2.2 introduit pour les interfaces JPA Query et TypedQuery a nouvelle méthode appelée getResultStream (), qui retournera un flux Java 8 de la requête résultat. Cette méthode, par défaut, déléguera à getResultList (). Stream (). Cette fournit un meilleur moyen de parcourir l'ensemble de résultats de la requête.

Ce changement a été suivi dans le numéro 99 de la spécification JPA, où il a été remarqué que lorsque la méthode de la liste est JPA 2.1 a été utilisée pour lire de grands ensembles de données, le résultat a été d'ajouter le «jeu de résultats» entier en mémoire avant de pouvoir être utilisé dans l'application.

Une requête peut être définie de manière statique ou dynamique. Une requête statique est généralement défini dans une annotation ou des métadonnées XML, et il doit inclure les critères de requête comme ainsi qu'un nom attribué par l'utilisateur. Ce type de requête est également appelé une requête nommée, et c'est recherché plus tard par son nom au moment de son exécution.

Une requête dynamique peut être émise lors de l'exécution en fournissant les critères de requête JP QL ou un objet de critères. Ils peuvent être un peu plus chers à exécuter car le le fournisseur de persistance ne peut effectuer aucune préparation de requête au préalable, mais les requêtes JP QL sont néanmoins très simple à utiliser et peut être émis en réponse à la logique du programme ou même logique utilisateur.

Ainsi, lorsque vous utilisez JPA 2.2, vous appelez la méthode getResultStream () au lieu de la méthode getResultList (). Le reste de l'API n'a pas changé, vous pouvez donc toujours créer le requête comme vous l'avez fait dans JPA 2.1.

L'exemple suivant montre comment créer une requête dynamique, puis l'exécuter pour obtenir tous les employés de la base de données. Bien sûr, ce n'est peut-être pas une très bonne requête pour s'exécute si la base de données est volumineuse et contient des centaines de milliers d'employés, mais est néanmoins un exemple légitime. La requête simple est la suivante:

Exemple utilisant JPA 2.1:

```
TypedQuery <Employee> query = em.createQuery ("SELECT e FROM Employee e",
                                                Classe.employé);
List <Employee> emps = query.getResultList ();
```

Exemple utilisant JPA 2.2 si nous voulons avoir un flux Java 8 de la requête résultat:

```
Stream <Employee> employee = em.createQuery ("SELECT a FROM Employee e",
Employee.class) .getResultStream ();
```

Nous créons un objet TypedQuery <Employee> en émettant l'appel createQuery () sur EntityManager et en passant la chaîne JP QL qui spécifie les critères de requête, comme ainsi que la classe sur laquelle la requête doit être paramétrée. La chaîne JP QL ne fait pas référence à une table de base de données EMPLOYEE mais à l'entité Employee, donc cette requête sélectionne tout Objets employés sans les filtrer davantage. Vous plongerez dans les requêtes dans Chapitre 7, JP QL dans les chapitres 7 et 8 , et requêtes de critères dans le chapitre 9 . Vous verrez que vous pouvez être beaucoup plus discrétionnaire quant aux objets que vous souhaitez retourner.

Comme lors de l'utilisation de JPA 2.2, la nouvelle méthode appelée getResultStream () renvoie un Java 8 flux du résultat de la requête. Donc, dans ce cas, il retournera le flux de la requête Employé résultat.

Pour utiliser la méthode getResultList à la place pour exécuter la requête, nous appelons simplement getResultList () dessus. Cela renvoie une liste <Employee> contenant les objets Employee correspondant aux critères de la requête. Notez que la liste est paramétrée par l'employé depuis le type paramétré est propagé à partir de l'argument de classe initial passé dans le méthode createQuery (). Nous pouvons facilement créer une méthode qui retourne tous les employés, comme indiqué dans la liste 2-8.

Piste 64

Liste 2-8. Méthode d'émission d'une requête

```
Liste publique <Employee> findAllEmployees () {
    TypedQuery <Employee> query = em.createQuery ("SELECT e FROM Employee e",
                                                  Classe.employé);
    return query.getResultList ();
}
```

Cet exemple montre à quel point il est simple de créer, d'exécuter et de traiter des requêtes, mais ne montre pas la puissance des requêtes. Au chapitre7, vous voyez beaucoup d'autres extrêmement des moyens utiles et intéressants de définir et d'utiliser des requêtes dans une application.

Mettre tous ensemble

Nous pouvons maintenant prendre toutes les méthodes que nous avons créées et les combiner dans une classe. le class agit comme une classe de service, que nous appelons EmployeeService, et nous permet d'exécuter opérations sur les employés. Le code devrait être assez familier maintenant. Liste 2-9 spectacles la mise en œuvre complète.

Liste 2-9. Classe de service pour l'exploitation des entités d'employés

```
import javax.persistence. *;
import java.util.List;

public class EmployeeService {
    protected EntityManager em;

    public EmployeeService (EntityManager em) {
        this.em = em;
    }

    public Employee createEmployee (int id, nom de chaîne, salaire long) {
        Employé emp = nouvel employé (id);
        emp.setName (nom);
```

```
        emp.setSalary (salaire);
        em.persist (emp);
        return emp;
    }
```


Piste 65


```
    public void removeEmployee (int id) {
        Employé emp = findEmployee (id);
        if (emp! = null) {
            em.remove (emp);
        }
    }

    public Employee riseEmployeeSalary (int id, longue augmentation) {
        Employé emp = em.find (Employee.class, id);
        if (emp! = null) {
            emp.setSalary (emp.getSalary () + augmenter);
        }
        return emp;
    }

    public Employee findEmployee (int id) {
        return em.find (Employee.class, id);
    }

    Liste publique <Employee> findAllEmployees () {
        TypedQuery <Employee> query = em.createQuery (
                «SELECT e FROM Employee e», Employee.class);
        return query.getResultList ();
    }
}
```

Il s'agit d'une classe simple mais entièrement fonctionnelle qui peut être utilisée pour lancer la création typique, opérations de lecture, de mise à jour et de suppression (CRUD) sur les entités Employé. Cette classe nécessite qu'un gestionnaire d'entités soit créé et transmis par l'appelant et aussi que tout les transactions requises sont commencées et validées par l'appelant. Cela peut sembler étrange à d'abord, mais le découplage de la logique de transaction de la logique d'opération rend cette classe plus portable dans l'environnement Java EE. Nous revisitons cet exemple dans le chapitre suivant, qui se concentre sur les applications Java EE.

Un programme principal simple qui utilise ce service et exécute toutes les entités requises la création du gestionnaire et la gestion des transactions sont affichées dans la liste 2-10 .

Piste 66

Liste 2-10. Utiliser EmployeeService

```
import javax.persistence. *;
```

```
import java.util.List;
Public class EmployeeTest {

    public static void main (String [] args) {
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory ("EmployeeService");
        EntityManager em = emf.createEntityManager ();
        EmployeeService service = nouveau EmployeeService (em);

        // créer et conserver un employé
        em.getTransaction (). begin ();
        Employé emp = service.createEmployee (158, "John Doe", 45000);
        em.getTransaction (). commit ();
        System.out.println ("Persisté" + emp);

        // trouver un employé spécifique
        emp = service.findEmployee (158);
        System.out.println ("Trouvé" + emp);

        // trouver tous les employés
        List <Employee> emps = service.findAllEmployees ();
        pour (Employé e: emps)
            System.out.println ("Employé trouvé:" + e);

        // mettre à jour l'employé
        em.getTransaction (). begin ();
        emp = service.raiseEmployeeSalary (158, 1000);
        em.getTransaction (). commit ();
        System.out.println ("Mis à jour" + emp);

        // supprimer un employé
        em.getTransaction (). begin ();
        service.removeEmployee (158);
        em.getTransaction (). commit ();
        System.out.println ("Employé supprimé 158");
```



```
        // ferme l'EM et l'EMF une fois terminé
        em.close ();
        emf.close ();
    }
}
```

Notez qu'à la fin du programme, nous utilisons les méthodes close () pour nettoyer le gestionnaire d'entités et l'usine que nous avons utilisée pour le créer. Cela garantit que tous les les ressources qu'ils auraient pu allouer sont correctement libérées.

Emballer

Maintenant que vous connaissez les éléments de base de JPA, vous êtes prêt à organiser les pièces dans une application qui s'exécute dans Java SE. La seule chose qui reste à discuter est de savoir comment le mettre ensemble pour qu'il fonctionne.

Unité de persistance

La configuration qui décrit l'unité de persistance est définie dans un fichier XML appelé

persistence.xml. Chaque unité de persistance est nommée, donc lorsqu'une application référençante veut spécifier la configuration d'une entité, il suffit de référencer le nom de l'unité de persistance qui définit cette configuration. Un seul fichier persistence.xml peut contiennent une ou plusieurs configurations d'unités de persistance nommées, mais chaque unité de persistance est séparé et distinct des autres, et ils peuvent être logiquement considérés comme fichiers persistence.xml séparés.

De nombreux éléments d'unité de persistance dans le fichier persistence.xml s'appliquent à unités de persistance déployées dans le conteneur Java EE. Les seuls que nous doivent spécifier pour cet exemple le nom, le type de transaction, la classe et les propriétés. Il existe un certain nombre d'autres éléments qui peuvent être spécifiés dans l'unité de persistance configuration dans le fichier persistence.xml, mais ils sont décrits plus en détail dans Chapitre 13. Le listing 2-11 montre les parties pertinentes du fichier persistence.xml pour cela exemple.

Piste 68

Liste 2-11. Éléments du fichier persistence.xml

```
<persistance>
    <persistence-unit name = "EmployeeService"
                                transaction-type = "RESOURCE_LOCAL">
        <classe> examples.model.Employee </class>
        <propriétés>
            <nom de propriété = "javax.persistence.jdbc.driver"
                                value = "org.apache.derby.jdbc.ClientDriver" />
            <nom de la propriété = "javax.persistence.jdbc.url"
                    value = "jdbc: derby: // localhost: 1527 / EmpServDB; create = true" />
            <property name = "javax.persistence.jdbc.user" value = "APP" />
            <property name = "javax.persistence.jdbc.password" value = "APP" />
        </properties>
    </persistence-unit>
</persistence>
```

L'attribut name de l'élément persistence-unit indique le nom de l'unité de persistance et est la chaîne que nous spécifions lorsque nous créons le EntityManagerFactory. Nous avons utilisé EmployeeService comme nom. Le type de transaction L'attribut indique que l'unité de persistance utilise EntityTransaction au niveau des ressources au lieu des transactions JTA. L'élément de classe répertorie l'entité qui fait partie du unité de persistance. Plusieurs éléments de classe peuvent être spécifiés lorsqu'il y a plus de une entité. Ils ne seraient normalement pas nécessaires lors du déploiement dans un conteneur Java EE car le conteneur recherchera automatiquement les classes d'entités annotées avec @Entity dans le cadre du processus de déploiement, mais ils sont nécessaires pour une exécution portable lorsque fonctionnant sous Java SE. Nous n'avons qu'une seule entité Employé.

La dernière section est juste une liste de propriétés qui peuvent être standard ou spécifiques au fournisseur. Les paramètres de connexion à la base de données JDBC doivent être spécifiés lors de l'exécution dans un Java SE environnement pour indiquer au fournisseur à quelle ressource se connecter. Autres propriétés du fournisseur, telles que les options de journalisation, sont spécifiques au fournisseur et peuvent également être utiles.

Archive de persistance

Les artefacts de persistance sont empaquetés dans ce que nous appellerons vaguement une archive de persistance. Il s'agit en réalité d'un fichier au format JAR contenant le fichier persistence.xml dans le META-

Répertoire INF et normalement les fichiers de classe d'entité.
48

Piste 69

Parce que l'application s'exécute comme une simple application Java SE, tout ce que nous avons à est de mettre l'archive de persistance, les classes d'application qui utilisent les entités et le JAR du fournisseur de persistance sur le chemin de classe lorsque le programme est exécuté.

Résumé

Ce chapitre a abordé juste assez des bases de l'API Java Persistence pour développer et exécutez une application simple dans un environnement d'exécution Java SE.

Nous avons commencé par discuter de l'entité, comment en définir une et comment transformer une entité existante Classe Java en une seule. Nous avons discuté des gestionnaires d'entités et de la manière dont ils sont obtenus et construit dans l'environnement Java SE.

L'étape suivante consistait à instancier une instance d'entité et à utiliser le gestionnaire d'entités pour le conserver dans la base de données. Après avoir inséré une nouvelle entité, nous pourrions la récupérer à nouveau et puis retirez-le. Nous avons également fait quelques mises à jour et nous nous sommes assurés que les changements étaient écrits retour à la base de données.

Nous avons parlé de l'API de transaction locale aux ressources et de son utilisation. Nous sommes ensuite allés sur certains des différents types de requêtes et comment les définir et les exécuter. Finalement, nous avons agrégé toutes ces techniques et les avons combinées dans une application simple qui nous pouvons exécuter indépendamment d'un environnement d'entreprise.

Dans le chapitre suivant, nous examinons l'impact de l'environnement Java EE lorsque développer des applications d'entreprise à l'aide de l'API Java Persistence.

Piste 70

CHAPITRE 3

Applications de l'entreprise

Aucune technologie n'existe dans le vide, et JPA n'est pas différent à cet égard. Bien que la graisse
le style d'application client démontré dans le chapitre précédent est une utilisation viable de JPA, le
la majorité des applications Java d'entreprise sont déployées sur un serveur d'applications, généralement
en utilisant les technologies Web Java EE, et éventuellement d'autres technologies. Il est donc
essentiel pour comprendre les composants qui composent une application déployée et le
rôle de JPA dans cet environnement.

Ce livre traite de JPA dans Java EE 8, dont les principales nouvelles fonctionnalités prennent en charge HTML5,
Intégration de la norme HTTP / 2 et du bean, et infrastructure améliorée pour les applications
s'exécutant dans le cloud.

Nous commençons par un aperçu des principales technologies Java EE pertinentes pour la persistance.
Dans le cadre de cet aperçu, nous décrivons le modèle de composant EJB, démontrant les
syntaxe pour certains des différents types d'EJB.

Nous allons ensuite couvrir brièvement le mécanisme standard d'injection de dépendances (DI),
principalement en utilisant l'approche de contextes Java EE et d'injection de dépendances (CDI). Cette
Le chapitre n'est pas destiné à être une exploration complète ou détaillée de Java EE ou d'un composant
frameworks, et nous ne pouvons pas entrer dans tous les frameworks DI de la sphère DI, ou
même les installations offertes par CDI. Cependant, les exemples CDI et EJB sont assez typiques
de DI en général et devrait donner aux lecteurs une idée générale de la façon dont JPA peut être utilisé avec
Composants compatibles DI, qu'ils soient de la variété Java EE ou d'un autre conteneur DI
composant, tel que Spring ou Guice.

Nous examinons ensuite les transactions, une autre technologie de serveur d'application qui a
impact majeur sur les applications utilisant JPA. Les transactions sont un élément fondamental de toute
application d'entreprise qui doit garantir l'intégrité des données.

Enfin, nous expliquons comment utiliser les technologies décrites dans ce chapitre dans
le contexte de la façon dont la persistance s'intègre dans chaque technologie de composant. Nous avons aussi
revoir l'application Java SE du chapitre précédent et la recibler vers l'entreprise
Plate-forme.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_3

Piste 71

Voici une liste des modifications les plus importantes de Java EE 8:

- Plateforme Java EE 8
- JSON-B 1.0
- JSON-P 1.1
- JAX-RS 2.1
- MVC 1.0
- Java Servlet 4.0
- JSF 2.3
- JMS 2.1
- CDI 2.0
- Java EE Security 1.0
- Java EE Management 2.0
- Utilitaires de concurrence
- Architecture de connecteur
- WebSocket
- JPA
- EJB

- JTA
- JCache
- JavaMail

Figure 3-1 montre les principaux composants de Java EE 8.

Psaumes 72

Figure 3-1. *Composants Java EE 8*

Figure 3-2 montre la pile Java EE 8.

Figure 3-2. *Pile Java EE 8*

Piste 73

Modèles de composants d'application

Le mot «composant» a pris de nombreuses significations dans le développement de logiciels, alors commencez par une définition. Un composant est une unité logicielle autonome et réutilisable qui peut être intégré dans une application. Les clients interagissent avec les composants via un Contrat. En Java, la forme la plus simple de composant logiciel est le JavaBean, généralement appelé juste un haricot. Les haricots sont des composants implémentés en termes d'une seule classe dont le contrat est défini par les patrons de dénomination des méthodes sur le bean. le Les modèles de dénomination JavaBean sont si courants maintenant qu'il est facile d'oublier qu'ils étaient initialement destiné à donner aux créateurs d'interfaces utilisateur une manière standard de gérer composants tiers.

Dans l'espace entreprise, les composants se concentrent davantage sur la mise en œuvre de services métier, avec le contrat du composant défini en termes d'opérations commerciales pouvant être effectuée par ce composant. Le modèle de composant traditionnel pour Java EE a toujours été le modèle EJB, qui définit les moyens d'empaqueter, de déployer et d'interagir avec services aux entreprises autonomes. Puis CDI est venu et a apporté un plus puissant et modèle flexible du composant de bean géré, les beans CDI étant soit des EJB ou classes Java non-EJB.

Le choix d'utiliser ou non un modèle de composant dans votre application est en grande partie personnel préférence, mais est généralement un bon choix de conception. L'utilisation de composants nécessite une organisation l'application en couches, les services métier vivant dans le modèle de composant et services de présentation superposés.

Historiquement, l'un des défis liés à l'adoption de composants dans Java EE était le complexité de leur mise en œuvre. Avec ce problème en grande partie résolu, nous nous retrouvons avec le les avantages suivants qu'un ensemble bien défini de services métier apporte à une application:

- *Couplage lâche* : l'utilisation de composants pour mettre en œuvre des services encourage couplage lâche entre les couches d'une application. La mise en oeuvre d'un composant peut changer sans aucun impact sur les clients ou composants qui en dépendent.
- *Gestion des dépendances* : les dépendances d'un composant peuvent être déclaré dans les métadonnées et résolu automatiquement par le conteneur.
- *Gestion du* cycle de vie: le cycle de vie des composants est bien défini et géré par le serveur d'application. Implémentations de composants peut participer aux opérations du cycle de vie pour acquérir et publier ressources, ou effectuez un autre comportement d'initialisation et d'arrêt.

Piste 74

- *Services de conteneur déclaratifs* : les méthodes commerciales pour les composants sont intercepté par le serveur d'application afin d'appliquer des services tels comme accès concurrentiel, gestion des transactions, sécurité et accès à distance.
- *Portabilité* : composants conformes aux normes Java EE et qui sont déployés sur des serveurs normalisés peuvent être portés plus facilement d'un serveur conforme à un autre.
- *Évolutivité et fiabilité* : les serveurs d'applications sont conçus pour garantir que les composants sont gérés efficacement dans un souci d'évolutivité. En fonction du type de composant et de la configuration du serveur, les opérations commerciales mises en œuvre à l'aide de composants peuvent échouer à nouveau

les appels de méthode ou même le basculement vers un autre serveur dans un cluster.

En lisant ce livre, vous remarquerez que dans certains cas, un exemple utilisera un composant pour héberger la logique métier et appeler l'API Java Persistence. Dans de nombreux cas ce sera un bean session, dans d'autres cas ce sera un bean CDI non-EJB, et encore d'autres ce sera un bean session géré par CDI (bean session avec une portée). Beans de session sont les composants préférés car ils sont les plus simples à écrire et à configurer et un ajustement naturel pour interagir avec JPA. Le type de composant que nous utilisons est moins important (le type de composant est largement substituable tant qu'il est géré par un conteneur qui prend en charge JPA et les transactions) que d'illustrer comment les composants s'intègrent dans le JPA et peut l'invoquer.

Figure 3-3 montre le modèle des composants d'application Java.

Figure 3-3. *Modèle de composants d'application Java*

Session Beans

Les Session Beans sont une technologie de composant conçue pour encapsuler des services métier. Les opérations accessibles au client prises en charge par le service peuvent être définies à l'aide d'un Java interface, ou en l'absence d'interface, juste l'ensemble des méthodes publiques sur le bean

classe d'implémentation. La classe bean n'est guère plus qu'une classe Java classique, et pourtant, par
du fait de faire partie du modèle de composant EJB, le bean a accès à un large éventail de
services de conteneurs. La signification du nom «session bean» est liée à la manière dont
dans lequel les clients y accèdent et interagissent avec eux. Une fois qu'un client acquiert une référence à un
session bean du serveur, il démarre une session avec ce bean et peut appeler business
opérations sur elle.

Il existe trois types de beans session: sans état, avec état et singleton. Interaction
avec un bean session *sans état* commence au début d'un appel de méthode métier et se termine
lorsque l'appel de méthode se termine. Il n'ya pas d'État qui soit reporté d'une entreprise

Psaumes 76

opération à l'autre. Une interaction avec des beans session avec *état* devient plus
une conversation qui commence à partir du moment où le client acquiert une référence au
session et se termine lorsque le client le libère explicitement sur le serveur. Entreprise
les opérations sur un bean session avec état peuvent maintenir l'état de l'instance du bean
appels. Nous fournissons plus de détails sur les considérations de mise en œuvre de cette différence dans
style d'interaction comme nous décrivons chaque type de session bean.

Les beans session *singleton* peuvent être considérés comme un hybride de session sans état et avec état
des haricots. Tous les clients partagent la même instance de bean singleton, il devient donc possible de partager
état à travers les appels de méthode, mais les beans session singleton n'ont pas la conversation
contrat et mobilité des beans session avec état. Etat sur un bean session singleton également
soulève des problèmes de concurrence qui doivent être pris en compte au moment de décider
s'il faut utiliser ce style de session bean.

Comme avec la plupart des conteneurs de composants, les clients d'un conteneur EJB ne
interagissez directement avec une instance de bean session. Le client référence et appelle un
implémentation de l'interface métier ou de la classe de bean fournie par le serveur. Cette
La classe d'implémentation agit comme un proxy pour l'implémentation du bean sous-jacent. Cette
le découplage du client du bean permet au serveur d'intercepter les appels de méthode afin de
fournir les services requis par le bean, comme la gestion des transactions. Ça aussi
permet au serveur d'optimiser et de réutiliser les instances de la classe du bean session si nécessaire.

Dans les sections suivantes, nous abordons les beans session utilisant une activité synchrone
invocations de méthode. Les méthodes commerciales asynchrones offrent une autre invocation
modèle impliquant des futurs, mais sortent du cadre de ce livre.

Beans session sans état

Comme mentionné, un bean session sans état se propose de terminer une opération dans le
durée de vie d'une seule méthode. Les beans apatrides peuvent mettre en œuvre de nombreuses opérations commerciales,
mais chaque méthode ne peut pas supposer qu'une autre a été invoquée avant elle.

Cela peut sembler être une limitation du haricot sans état, mais c'est de loin le plus
forme courante de mise en œuvre de services commerciaux. Contrairement aux beans session avec état, qui
sont bons pour accumuler l'état au cours d'une conversation (comme le panier de
une application de vente au détail), les beans session sans état sont conçus pour effectuer des
opérations très efficacement. Les beans session sans état peuvent s'adapter à un grand nombre de clients
avec un impact minimal sur les ressources globales du serveur.

Piste 77

Définition d'un bean session sans état

Un bean session est défini en deux parties:

- Zéro ou plus d'interfaces métiers qui définissent quelles méthodes un client peut invoquer sur le haricot. Lorsqu'aucune interface n'est définie, l'ensemble des les méthodes publiques sur la classe d'implémentation du bean forment un interface client.
- Une classe qui implémente ces interfaces, appelée la classe bean, qui est marqué de l'annotation @Stateless.

Que vous souhaitiez mettre en avant votre session bean avec une interface réelle ou non, c'est question de préférence. Nous montrons des exemples des deux, mais n'utilisons généralement pas l'interface dans exemples ultérieurs.

Regardons d'abord une version interfacée d'un bean session sans état. Liste 3-1 montre l'interface métier prise en charge par ce bean session. Dans cet exemple, le service se compose d'une seule méthode, sayHello (), qui accepte un argument String correspondant au nom d'une personne et renvoie une réponse String. Il n'y a pas d'annotation ou interface parent pour indiquer qu'il s'agit d'une interface métier. Lorsqu'il est mis en œuvre par le session bean, il sera automatiquement traité comme une interface métier locale, ce qui signifie qu'il est accessible uniquement aux clients du même serveur d'applications. Un deuxième type d'entreprise L'interface pour les clients distants est également possible mais rarement utilisée.

Liste 3-1. L'interface métier d'un bean session

```
interface publique HelloService {
    public String sayHello (nom de la chaîne);
}
```

Considérons maintenant l'implémentation, qui est montrée dans l'extrait 3-2 . C'est un classe Java standard qui implémente l'interface métier HelloService. La seule chose qui est unique à propos de cette classe est l'annotation [1] @Stateless qui la marque comme un état sans état session bean. La méthode métier est mise en œuvre sans contrainte particulière ni exigences. Il s'agit d'une classe régulière qui se trouve être un EJB.

[1] Toutes les annotations utilisées dans ce chapitre sont définies dans le fichier javax.ejb, javax.inject, javax. packages enterprise.inject ou javax.annotation.

Liste 3-2. La classe Bean implémentant l'interface HelloService

```
@Apatride
public class HelloServiceBean implémente HelloService {
    public String sayHello (String name) {
        retourne "Bonjour," + nom;
    }
}
```

En termes de conception d'API, l'utilisation d'une interface est probablement le meilleur moyen d'exposer un les opérations du bean session, car il sépare l'interface de l'implémentation. Cependant, la norme actuelle est d'implémenter les composants sous forme de classes simples contenant la logique métier sans interface. En utilisant cette approche, le bean session être simplement comme indiqué dans le Listing 3-3 .

Liste 3-3. Un Session Bean sans interface

```
@Apatride
public class HelloService {
        public String sayHello (String name) {
                retourne «Bonjour», + nom;
        }
}
```

L'interface logique du bean session est constituée de ses méthodes publiques; dans ce cas, la méthode sayHello (). Les clients utilisent la classe HelloServiceBean comme s'il s'agissait d'une interface et doit ignorer les méthodes non publiques ou les détails de la mise en œuvre. En dessous de les couvertures, le client interagira avec un proxy qui étend la classe de bean et remplace les méthodes commerciales pour fournir les services de conteneur standard.

Rappels de cycle de vie

Contrairement à une classe Java standard utilisée dans le code d'application, le serveur gère le cycle de vie de un bean session sans état. Le serveur décide quand créer et supprimer des instances de bean et doit initialiser les services pour une instance de bean après sa construction, mais avant la logique métier du bean est invoquée. De même, le haricot devra peut-être acquérir un ressource, telle qu'une source de données JDBC, avant que les méthodes métier puissent être utilisées. cependant, pour que le bean acquière une ressource, le serveur doit d'abord avoir terminé

Piste 79

initialisation de ses services pour le bean. Cela limite l'utilité du constructeur pour le class car le bean n'aura accès à aucune ressource tant que l'initialisation du serveur n'aura pas terminé.

Pour permettre au serveur et au bean de répondre à leurs exigences d'initialisation, Les EJB prennent en charge les méthodes de rappel du cycle de vie qui sont appelées par le serveur à différents moments dans le cycle de vie du haricot. Pour les beans session sans état, il existe deux rappels de cycle de vie: PostConstruct et PreDestroy. Le serveur invoquera le rappel PostConstruct comme dès qu'il a terminé l'initialisation de tous les services de conteneur pour le bean. En effet, ce remplace le constructeur comme emplacement de la logique d'initialisation car ce n'est qu'ici que les services de conteneurs sont garantis disponibles. Le serveur appelle le PreDestroy rappel immédiatement avant que le serveur ne libère l'instance de bean comme étant des ordures collectées. Toutes les ressources acquises pendant PostConstruct qui nécessitent un arrêt explicite devrait être libéré pendant PreDestroy.

Le Listing 3-4 montre un bean session sans état qui acquiert une référence à un java. util.logging.Logger lors du rappel PostConstruct, identifié par le Annotation de marqueur @PostConstruct. De même, un rappel PreDestroy est identifié par l'annotation @PreDestroy.

Liste 3-4. Utilisation du rappel PostConstruct pour acquérir un enregistreur

```
@Apatride
public class LoggerBean {
        enregistreur privé;

        @PostConstruct
        void init () {
                logger = Logger.getLogger ("notification");
        }

        public void logMessage (String message) {
                logger.info (message);
        }
```

}

Piste 80

Beans session avec état

Dans l'introduction aux beans session, nous avons décrit la différence entre stateless
et les beans avec état comme étant basés sur le style d'interaction entre le client et le serveur.
Dans le cas des beans session sans état, cette interaction a commencé et s'est terminée par un seul
appel de méthode. Parfois, les clients doivent émettre plusieurs demandes et recevoir chaque demande
être en mesure d'accéder ou de prendre en compte les résultats des demandes précédentes. Les beans session avec état sont
conçu pour gérer ce scénario en fournissant un service dédié à un client qui démarre
lorsque le client obtient une référence au bean et ne se termine que lorsque le client choisit
pour mettre fin à la conversation.

L'exemple par excellence du bean session avec état est le panier d'un
Application de commerce électronique. Le client obtient une référence au panier, en commençant le
conversation. Pendant la durée de la session utilisateur, le client ajoute ou supprime des éléments de
le panier, qui conserve un état propre au client. Ensuite, lorsque la session est
terminé, le client finalise l'achat, entraînant la suppression du panier.

Ce n'est pas sans rappeler l'utilisation d'un objet Java non géré dans le code d'application. Tu crées
une instance, invoquez des opérations sur l'objet qui accumulent l'état, puis supprimez
l'objet lorsque vous n'en avez plus besoin. La seule différence avec le bean session avec état
est que le serveur gère l'instance d'objet réelle et le client interagit avec cela
instance indirectement via un objet proxy.

Les beans session avec état offrent un sur-ensemble des fonctionnalités disponibles en état sans état
haricots de session. Les fonctionnalités que nous avons couvertes pour les beans session sans état s'appliquent également à
beans session avec état.

Définition d'un bean session avec état

Maintenant que nous avons établi le cas d'utilisation d'un bean session avec état, voyons comment
définir un. Similaire au bean session sans état, un bean session avec état peut ou non
avoir une interface implémentée par une seule classe de bean. Référencement3-5 montre la classe de haricots
pour le bean session avec état ShoppingCart. La classe de haricots a été marquée du
@Stateful annotation pour indiquer au serveur que la classe est un bean session avec état.

Liste 3-5. Implémentation d'un panier à l'aide d'un bean session avec état

```
@Stateful
ShoppingCart de classe publique {
    private HashMap <String, Integer> items = new HashMap <String, Integer> ();
```


Piste 81


```
    public void addItem (String item, quantité int) {
        Nombre entier orderQuantity = items.get (item);
```

```
            if (orderQuantity == null) {
                orderQuantity = 0;
            }
            orderQuantity + = quantité;
            items.put (item, orderQuantity);
        }

        public void removeItem (String item, quantité int) {
            // ...
        }

        Public Map <String, Integer> getItems () {
            // ...
        }

        // ...

        @Retirer
        paiement annulé public (int paymentId) {
            // stocker les éléments dans la base de données
            // ...
        }

        @Retirer
        public void cancel () {
        }
}
```

Il y a deux choses différentes dans ce bean par rapport aux beans session sans état nous avons traité jusqu'à présent.

La première différence est que la classe de bean a des champs d'état qui sont modifiés par le méthodes commerciales du haricot. Ceci est autorisé car le client qui utilise le bean a effectivement accès à une instance privée du bean session sur laquelle apporter des modifications.

La deuxième différence est qu'il existe des méthodes marquées avec @Remove annotation. Ce sont les méthodes que le client utilisera pour terminer la conversation avec le haricot. Une fois l'une de ces méthodes appelée, le serveur détruira le bean

instance, et la référence client lèvera une exception si une autre tentative est faite pour invoquer des méthodes commerciales. Chaque bean session avec état doit définir au moins une méthode marqué avec l'annotation @Remove, même si la méthode ne fait rien d'autre que servir de fin à la conversation. Dans la liste3-5, la méthode checkout () est appelée si le l'utilisateur termine la transaction d'achat, bien que cancel () soit appelé si l'utilisateur décide de ne pas continuer. Le bean session est supprimé dans les deux cas.

Rappels de cycle de vie

Comme le bean session sans état, le bean session avec état prend également en charge les rappels de cycle de vie afin de faciliter l'initialisation et le nettoyage du bean. Il prend également en charge deux des rappels pour permettre au bean de gérer correctement la passivation et l'activation du bean exemple. *La passivation* est le processus par lequel le serveur sérialise l'instance du bean afin qu'il peut être stocké hors ligne pour libérer des ressources ou répliqué sur un autre serveur dans Un groupe. *L'activation* est le processus de désérialisation d'une instance de bean session passivée et le rendre à nouveau actif dans le serveur. Parce que les beans session avec état conservent l'état au nom d'un client et ne sont supprimés que lorsque le client appelle l'un des méthodes sur le bean, le serveur ne peut pas détruire une instance de bean pour libérer des ressources.

La passivation permet au serveur de récupérer temporairement des ressources tout en préservant la session Etat.

Avant qu'un bean ne soit passivé, le serveur invoquera le callback PrePassivate. le bean utilise ce rappel pour préparer le bean à la sérialisation, généralement en fermant tout live connexions à d'autres ressources du serveur. La méthode PrePassivate est identifiée par @ Annotation de marqueur PrePassivate. Une fois qu'un bean a été activé, le serveur invoquera le rappel PostActivate. Une fois l'instance sérialisée restaurée, le bean doit alors réacquérir toutes les connexions à d'autres ressources que les méthodes métier du bean pourrait dépendre de. La méthode PostActivate est identifiée par @PostActivate annotation de marqueur.

Référencement 3-6 montre un bean session qui utilise les rappels de cycle de vie pour maintenir un JDBC lien. Notez que seule la connexion JDBC est gérée explicitement. En tant que ressource fabrique de connexions, le serveur enregistre et restaure automatiquement la source de données pendant passivation et activation.

Piste 83

Liste 3-6. Utilisation des rappels de cycle de vie sur un bean session avec état

```
@Stateful
public class OrderBrowser {
    DataSource ds;
    Connexion conn;

    @PostConstruct
    void init () {
        // acquérir la source de données
        // ...

        AcquérirConnexion ();
    }

    @PrePassivate
    void passivate () {releaseConnection (); }

    @PostActivate
    void activate () {AcquérirConnexion (); }

    @PreDestroy
    void shutdown () {releaseConnection (); }

    private void acquiertConnection () {
        essayez {
            conn = ds.getConnection ();
        } catch (SQLException e) {
            lancer une nouvelle EJBException (e);
        }
    }

    private void releaseConnection () {
        essayez {
            conn.close ();
        } catch (SQLException e) {
        } enfin {
            conn = null;
```

```
        }
    }
```


Psaumes 84


```
    Collection publique <Order> listOrders () {
        // ...
    }

    @Retirer
    public void remove () {}
}
```

Haricots de session singleton

Deux des critiques les plus courantes du bean session apatride ont été
la surcharge perçue du regroupement de haricots et l'incapacité de partager l'état via des champs statiques. le
Le bean session singleton tente de fournir une solution aux deux problèmes en fournissant
une seule instance de bean partagé qui peut être accédée simultanément et utilisée comme
mécanisme pour l'état partagé. Les beans session singleton partagent les mêmes rappels de cycle de vie
en tant que bean session sans état et ressources gérées par le serveur telles que les contextes de persistance
se comportent de la même manière que s'ils faisaient partie d'un bean session sans état. Mais les similitudes s'arrêtent
là parce que les beans session singleton ont un cycle de vie global différent de celui sans état
session beans et ont la complexité supplémentaire du verrouillage contrôlé par le développeur pour
synchronisation.

Contrairement aux autres beans session, le singleton peut être déclaré créé avec empressement
pendant l'initialisation de l'application et exister jusqu'à ce que l'application s'arrête. Une fois que
créé, il continuera d'exister jusqu'à ce que le conteneur le supprime, indépendamment de tout
exceptions qui se produisent pendant l'exécution de la méthode métier. C'est une différence clé par rapport à
d'autres types de bean session car l'instance de bean ne sera jamais recréée dans l'événement
d'une exception système.

La longue durée de vie et l'instance partagée du bean session singleton en font l'idéal
endroit pour stocker l'état d'application commun, qu'il soit en lecture seule ou en lecture-écriture. Pour sauvegarder
accès à cet état, le bean session singleton fournit un certain nombre de concurrence
options en fonction des besoins du développeur de l'application. Les méthodes peuvent être
complètement non synchronisé pour les performances, ou automatiquement verrouillé et géré par
le conteneur.

Piste 85

Définition d'un bean de session singleton

Suivant le modèle des beans session sans état et avec état, les beans session singleton
sont définis à l'aide de l'annotation @Singleton. Les beans session singleton peuvent inclure un
interface ou utilisez une vue sans interface. Le Listing 3-7 montre un simple bean session singleton
avec une vue sans interface pour suivre le nombre de visites sur un site Web.

Liste 3-7. Implémentation d'un bean de session singleton

```
@Singleton
public class HitCounter {
      nombre int;

      public void increment () {++ count; }

      public void getCount () {nombre de retours; }

      public void reset () {count = 0; }
}
```

Si vous comparez le bean HitCounter dans la liste 3-7 avec les apatrides et avec état
session beans définis précédemment, vous pouvez voir deux différences immédiates. Contrairement au
bean session sans état, il existe un état sous la forme d'un champ de comptage utilisé pour capturer la visite
compter. Mais contrairement au bean session avec état, il n'y a pas d'annotation @Remove pour identifier le
méthode commerciale qui terminera la session.

Par défaut, le conteneur gérera la synchronisation des méthodes métiers
pour garantir que les données ne sont pas corrompues. Dans cet exemple, cela signifie tous les accès
au bean est sérialisé de sorte qu'un seul client appelle une méthode métier sur le
instance à tout moment.

Le cycle de vie du bean session singleton est lié au cycle de vie de l'ensemble
application. Le conteneur détermine le moment où l'instance singleton obtient
créé à moins que la classe de bean ne soit annotée avec l'annotation @Startup pour forcer
initialisation au démarrage de l'application. Le conteneur peut créer des singletons qui ne
spécifiez l'initialisation hâtive paresseusement, mais cela est spécifique au fournisseur et ne peut pas être supposé.

Psaumes 86

Rappels de cycle de vie

Les rappels de cycle de vie pour les beans session singleton sont les mêmes que pour les sessions sans état
beans: PostConstruct et PreDestroy. Le conteneur appellera le PostConstruct
callback après l'initialisation du serveur de l'instance du bean et invoquez également le
Rappel de PreDestroy avant de supprimer l'instance de bean. La principale différence ici
en ce qui concerne les beans session sans état, PreDestroy n'est appelé que lorsque le
l'application s'arrête dans son ensemble. Il ne sera donc appelé qu'une seule fois, alors que le
les rappels de cycle de vie des beans session sans état sont fréquemment appelés comme les instances de bean
créé et détruit.

Servlets

Les servlets sont une technologie de composants conçue pour répondre aux besoins des développeurs Web qui
besoin de répondre aux requêtes HTTP et de générer du contenu dynamique en retour. Les servlets sont
la technologie la plus ancienne et la plus populaire introduite dans le cadre de la plate-forme Java EE. Ils
sont la base de technologies telles que JavaServer Pages (JSP) et l'épine dorsale de
infrastructures Web telles que JavaServer Faces (JSF).

Bien que vous ayez une certaine expérience avec les servlets, il vaut la peine de décrire
l'impact que les modèles d'applications Web ont eu sur le développement d'applications d'entreprise.
En raison de sa dépendance au protocole HTTP, le Web est par nature un support sans état.

Tout comme les beans session sans état décrits précédemment, un client fait une requête, le
le serveur déclenche la méthode de service appropriée dans le servlet et le contenu est généré
et renvoyé au client. Chaque demande est entièrement indépendante de la dernière.

Cela présente un défi car de nombreuses applications Web impliquent une sorte de
conversation entre le client et le serveur dans laquelle les actions précédentes de l'utilisateur
influencer les résultats renvoyés sur les pages suivantes. Pour maintenir cette conversation
état, de nombreuses applications antérieures ont tenté d'incorporer dynamiquement des informations de contexte dans
URL. Malheureusement, non seulement cette technique ne s'adapte pas très bien, mais elle nécessite également
un élément dynamique à toute génération de contenu qui rend difficile pour les non-développeurs de
rédiger du contenu pour une application Web.

Les servlets résolvent le problème de l'état conversationnel avec la session. Ne pas être
confondue avec le bean session, la session HTTP est une carte de données associée à un
ID de session. Lorsque l'application demande la création d'une session, le serveur génère
un nouvel ID et renvoie un objet HTTPSession que l'application peut utiliser pour stocker la clé /

paires de valeurs de données. Il utilise ensuite des techniques telles que les cookies du navigateur pour lier la session
ID avec le client, liant les deux ensemble dans une conversation. Pour les applications Web, le
le client ignore en grande partie l'état de conversation suivi par le serveur.

L'utilisation efficace de la session HTTP est un élément important du développement de servlet.
Référencement 3-8 montre les étapes requises pour demander une session et stocker
données dedans. Dans cet exemple, en supposant que l'utilisateur s'est connecté, le servlet stocke le
ID utilisateur dans la session, le rendant disponible pour une utilisation dans toutes les demandes ultérieures par le même
client. L'appel getSession () sur l'objet HttpServletRequest renverra soit le
session active ou en créer une nouvelle s'il n'en existe pas. Une fois obtenue, la session
agit comme une carte, avec des paires clé / valeur définies et récupérées avec setAttribute () et
getAttribute (), respectivement. Comme vous le verrez plus loin dans ce chapitre, le servlet
session, qui stocke des données non structurées, est parfois associée à un bean session avec état
pour gérer les informations de session avec le bénéfice d'une interface métier bien définie.

Liste 3-8. Maintien de l'état conversationnel avec un servlet

```
La classe publique LoginServlet étend HttpServlet {

    protected void doPost (requête HttpServletRequest, HttpServletResponse
    réponse)
                lance ServletException, IOException {
        String userId = request.getParameter ("utilisateur");
        Session HttpSession = request.getSession ();
        session.setAttribute ("utilisateur", userId);
        // ...
    }
}
```

La montée en puissance des frameworks applicatifs ciblés sur le Web a également changé la manière
dans lequel nous développons des applications Web. Le code d'application écrit dans les servlets est rapidement
en cours de remplacement par un code d'application extrait du modèle de base en utilisant
frameworks tels que JSF. Lorsque vous travaillez dans un environnement comme celui-ci, l'application de base
problèmes de persistance, tels que où acquérir et stocker le gestionnaire d'entités et comment
utiliser efficacement les transactions rapidement, devenir pertinentes.

Bien que nous explorions certains de ces problèmes, la persistance dans le contexte d'un cadre
comme JSF est au-delà de la portée de ce livre. Comme solution générale, nous recommandons
adopter un modèle de composant dans lequel concentrer les opérations de persistance. Session beans,

Psaumes 88

par exemple, sont facilement accessibles de n'importe où dans une application Java EE, parfait terrain neutre pour les services aux entreprises. La possibilité d'échanger des entités à l'intérieur et en dehors du modèle de bean session signifie que les résultats des opérations de persistance sera directement utilisable dans les frameworks Web sans avoir à coupler étroitement votre code de présentation à l'API de persistance.

Gestion des dépendances et CDI

La logique métier d'un composant Java EE n'est généralement pas complètement autonome. Plus souvent qu'autrement, la mise en œuvre dépend d'autres ressources. Cette peut inclure des ressources de serveur, telles qu'une source de données JDBC, ou définies par l'application ressources, telles qu'un autre composant ou gestionnaire d'entités pour une persistance spécifique unité. La plate-forme Java EE principale contient une prise en charge assez limitée pour l'injection dépendances dans un nombre limité de ressources serveur prédéfinies, telles que des données sources, transactions gérées et autres. Cependant, la norme CDI va bien au-delà injection de dépendances simple (DI) et fournit un cadre complet pour prendre en charge une gamme d'exigences, du trivial à l'exotique. Nous commençons par décrire la base les concepts et le support contenus dans la plate-forme avant le CDI, puis passer à autre chose au CDI et à son modèle DI contextuel.

Les composants Java EE prennent en charge la notion de références aux ressources. Une référence est un lien nommé vers une ressource qui peut être résolu dynamiquement à l'exécution depuis l'intérieur code d'application ou résolu automatiquement par le conteneur lorsque le composant l'instance est créée. Nous couvrons chacun de ces scénarios brièvement.

Une référence se compose de deux parties: un nom et une cible. Le nom est utilisé par code d'application pour résoudre la référence de manière dynamique, alors que le serveur utilise la cible informations pour trouver la ressource recherchée par l'application. Le type de ressource à être localisé détermine le type d'informations requises pour correspondre à la cible. Chaque ressource référence requiert un ensemble différent d'informations spécifiques au type de ressource auquel se réfère.

Une référence est déclarée à l'aide de l'une des annotations de référence de ressource: @Resource, @EJB, @PersistenceContext ou @PersistenceUnit. Ces annotations peuvent être placées sur une méthode de classe, de champ ou de setter. Le choix de l'emplacement détermine le nom par défaut du référence, et si le serveur résout ou non la référence automatiquement.

Psaumes 89

Recherche de dépendances

La première stratégie de résolution des dépendances dans le code d'application que nous discutons est appelée la *recherche de dépendance* . Il s'agit de la forme traditionnelle de gestion des dépendances en Java EE, dans lequel le code d'application est responsable de la recherche d'une référence nommée à l'aide de l'interface de nommage et d'annuaire Java (JNDI).

Toutes les annotations de ressources prennent en charge un attribut appelé nom qui définit le JNDI nom de la référence. Lorsque l'annotation de ressource est placée sur la définition de classe, cet attribut est obligatoire. Si l'annotation de ressource est placée sur un champ ou un setter

méthode, le serveur générera un nom par défaut. Lors de l'utilisation de la recherche de dépendances, les annotations sont généralement placées au niveau de la classe et le nom est explicitement spécifié. Placer une référence de ressource sur un champ ou une méthode de définition a d'autres effets que générer un nom par défaut dont nous parlerons dans la section suivante.

Le rôle du nom est de fournir un moyen pour le client de résoudre la référence dynamiquement. Chaque serveur d'applications Java EE prend en charge JNDI, même s'il est moins fréquemment utilisé par les applications depuis l'avènement de l'injection de dépendances, et chaque Java Le composant EE a son propre contexte de dénomination JNDI à portée locale appelé environnement contexte de dénomination. Le nom de la référence est lié à la dénomination de l'environnement contexte, et lorsqu'il est recherché à l'aide de l'API JNDI, le serveur résout la référence et renvoie la cible de la référence.

Considérez le bean de session DeptService illustré dans le Listing 3-9. Il a déclaré une dépendance sur un bean session utilisant l'annotation @EJB et lui donnant le nom deptAudit. L'élément beanInterface de l'annotation @EJB fait référence à la session classe de haricots. Dans le rappel PostConstruct, le bean d'audit est recherché et stocké dans le domaine d'audit. Les interfaces Context et InitialContext sont toutes deux définies par le JNDI API. La méthode lookup () de l'interface contextuelle est la manière traditionnelle de récupérer objets d'un contexte JNDI. Pour trouver la référence nommée deptAudit, l'application recherche le nom java: comp / env / deptAudit et convertit le résultat en AuditService. le préfixe java: comp / env / qui a été ajouté au nom de référence indique au serveur que le contexte de dénomination de l'environnement doit être recherché pour trouver la référence. Si le nom est incorrectement spécifié, la recherche échouera.

Piste 90

Liste 3-9. Recherche d'une dépendance EJB

```
@Apatride
@EJB (nom = "deptAudit", beanInterface = AuditService.class)
public class DeptService {
    audit privé AuditService;

    @PostConstruct
    void init () {
        essayez {
            Contexte ctx = new InitialContext ();
            audit = (AuditService) ctx.lookup ("java: comp / env / deptAudit");
        } catch (NamingException e) {
            lancer une nouvelle EJBException (e);
        }
    }

    // ...
}
```

Utilisation de l'API JNDI pour rechercher des références de ressources à partir de la dénomination de l'environnement context est pris en charge par tous les composants Java EE. C'est, cependant, un peu encombrant méthode de recherche d'une ressource en raison des exigences de gestion des exceptions de JNDI. Les EJB prennent également en charge une syntaxe alternative utilisant la méthode lookup () de l'interface EJBContext. L'interface EJBContext (et les sous-interfaces telles que SessionContext) est disponible pour n'importe quel EJB et permet au bean d'accéder au runtime services tels que le service de minuterie. Référencement3-10 montre le même exemple que le Listing 3-9

en utilisant la méthode lookup (). L'instance SessionContext dans cet exemple est fournie via une méthode setter appelée par le conteneur. Nous revisitons cet exemple plus loin dans le section intitulée «Référencement des ressources du serveur» pour voir comment il est appelé.

Liste 3-10. Utilisation de la méthode lookup () EJBContext

```
@Apatride
@EJB (nom = "deptAudit", beanInterface = AuditService.class)
public class DeptService {
        Contexte SessionContext;
        AuditService audit;
```



```
        public void setSessionContext (contexte SessionContext) {
                this.context = contexte;
        }

        @PostConstruct
        public void init () {
                audit = (AuditService) context.lookup ("deptAudit");
        }

        // ...
}
```

La méthode lookup () EJBContext présente deux avantages par rapport à l'API JNDI. La première est que l'argument de la méthode est le nom exactement tel qu'il a été spécifié dans la ressource référence. La seconde est que seules les exceptions d'exécution sont levées à partir de la recherche () afin d'éviter la gestion des exceptions vérifiées de l'API JNDI. Derrière la scènes, la même séquence d'appels d'API JNDI du Listing 3-9 est effectuée, mais le Les exceptions JNDI sont gérées automatiquement.

Injection de dépendance

Lorsqu'une annotation de ressource est placée sur un champ ou une méthode de définition, deux choses se produisent. Premier, une référence de ressource est déclarée comme si elle avait été placée sur la classe du bean (similaire à la façon dont l'annotation @EJB fonctionnait dans l'exemple du Listing 3-9) et le nom de cette ressource sera liée au contexte de dénomination de l'environnement lorsque le composant est créé. Deuxièmement, le serveur effectue la recherche automatiquement en votre nom et définit le result dans la classe instanciée.

Le processus de recherche automatique d'une ressource et de sa définition dans la classe est appelé *injection de dépendances* car on dit que le serveur injecte la dépendance résolue dans la classe. Cette technique, l'une des nombreuses communément appelées inversion de contrôle, supprime le fardeau de la recherche manuelle des ressources du JNDI contexte environnemental.

L'injection de dépendances est considérée comme une meilleure pratique pour le développement d'applications, non seulement parce que cela réduit le besoin de recherches JNDI mais aussi parce que cela simplifie essai. Sans aucun code d'API JNDI dans la classe qui a des dépendances sur l'application environnement d'exécution du serveur, la classe bean peut être instanciée directement dans un test unitaire.

Psaumes 92

Le développeur peut ensuite fournir manuellement les dépendances requises et tester le
fonctionnalité de la classe en question au lieu de se soucier de la façon de contourner le
Recherche JNDI.

Injection de champ

La première forme d'injection de dépendances est appelée *injection de champ.* Injecter une dépendance
dans un champ signifie qu'après que le serveur recherche la dépendance dans l'environnement
contexte de nommage, il affecte le résultat directement dans le champ annoté de la classe.
Référencement 3-11 revisite l'exemple du Listing 3-9 et démontre une utilisation plus simple de
l'annotation @EJB, cette fois en injectant le résultat dans le champ d'audit. Le répertoire
le code d'interface utilisé auparavant est parti, et les méthodes métier du bean peuvent supposer
que le champ d'audit contient une référence au bean AuditService.

Liste 3-11. Utilisation de l'injection de champ

```
@Apatride
public class DeptService {
      Audit @EJB AuditService;

      // ...
}
```

L'injection de champ est certainement la plus simple à mettre en œuvre, et les exemples de ce livre
choisissez toujours d'utiliser ce formulaire plutôt que le formulaire de recherche dynamique. La seule chose à
considérer avec l'injection sur le terrain est que si vous prévoyez des tests unitaires, vous devez soit
pour ajouter une méthode setter ou rendre le champ accessible à vos tests unitaires pour satisfaire manuellement
la dépendance. Les champs privés, bien que légaux, nécessitent des hacks désagréables s'il n'y a pas
moyen accessible de définir leur valeur. Considérez la portée du package pour l'injection de champ si vous le souhaitez
test unitaire sans avoir à ajouter de passeur.

Nous avons mentionné dans la section précédente qu'un nom est automatiquement généré pour
la référence lorsqu'une annotation de ressource est placée sur un champ ou une méthode de définition. Pour
exhaustivité, nous décrivons le format de ce nom, mais il est peu probable que vous trouviez
de nombreuses possibilités de l'utiliser. Le nom généré est le nom complet de la classe,
suivi d'une barre oblique puis du nom du champ ou de la propriété. Cela signifie que si
le bean DeptService est situé dans le package persistence.session, l'EJB injecté
référencé dans le Listing 3-9 serait accessible dans le contexte de dénomination de l'environnement sous

Piste 93

le nom persistence.session.DeptService / audit. Spécification de l'élément de nom pour
l'annotation de ressource remplacera cette valeur par défaut.

Injection de poseur

La deuxième forme d'injection de dépendances est appelée *injection de setter* et implique
annoter une méthode setter au lieu d'un champ de classe. Lorsque le serveur résout le
référence, il invoquera la méthode de définition annotée avec le résultat de la recherche.
Référencement 3-12 revisite le Listing 3-9 une fois de plus pour démontrer l'utilisation de l'injection de setter.

Liste 3-12. Utilisation de l'injection de poseur

```
@Apatride
public class DeptService {
```

```
    audit privé AuditService;

    @EJB
    public void setAuditService (audit AuditService) {
        this.audit = audit;
    }

    // ...
}
```

Ce style d'injection permet des champs privés, mais fonctionne également bien avec les tests unitaires. Chaque test peut simplement instancier la classe du bean et effectuer manuellement la dépendance injection en invoquant la méthode setter, généralement en fournissant une implémentation du ressource requise adaptée au test.

Déclaration de dépendances

Les sections suivantes décrivent certaines des annotations de ressources décrites dans Java Spécification EE. Chaque annotation a un attribut de nom pour spécifier éventuellement le nom de référence de la dépendance. D'autres attributs sur les annotations sont spécifiques à le type de ressource à acquérir.

Épisode 94

Référencer un contexte de persistance

Dans le chapitre précédent, nous avons montré comment créer un gestionnaire d'entités pour un contexte de persistance utilisant un EntityManagerFactory retourné par la Persistence classe. Dans l'environnement Java EE, l'annotation @PersistenceContext peut être utilisée déclarer une dépendance sur un contexte de persistance et avoir le gestionnaire d'entités pour cela contexte de persistance acquis automatiquement.

Référencement 3-13 illustre l'utilisation de l'annotation @PersistenceContext pour acquérir un gestionnaire d'entités via l'injection de dépendances dans un bean session sans état. le L'élément unitName spécifie le nom de l'unité de persistance sur laquelle la persistance le contexte sera basé.

> Conseil Si l'élément unitName est omis, la manière dont le nom d'unité car le contexte de persistance est déterminé. Certains fournisseurs peuvent fournir une valeur par défaut value s'il n'y a qu'une seule unité de persistance pour une application, alors que d'autres peuvent exigent que le nom de l'unité soit spécifié dans un fichier de configuration spécifique au fournisseur.

Liste 3-13. Injection d'une instance EntityManager

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    // ...
}
```

Vous vous demandez peut-être pourquoi un champ d'état existe dans un bean session sans état; après tous, les responsables d'entités doivent conserver leur propre état pour pouvoir gérer un contexte de persistance. La bonne nouvelle est que la spécification a été conçue avec un conteneur

intégration à l'esprit, donc ce qui est réellement injecté dans la liste 3-13 n'est pas une entité
instance de gestionnaire comme celles que nous avons utilisées dans le chapitre précédent. La valeur injectée dans
le bean est un proxy géré par conteneur qui acquiert et libère des contextes de persistance
au nom du code d'application. Il s'agit d'une fonctionnalité puissante de l'API Java Persistence dans
Java EE et est couvert en détail dans le chapitre 6 .

Psaumes 95

Pour l'instant, il est prudent de supposer que la valeur injectée «fera ce qu'il faut». Cela fait
ne doit pas être éliminé et fonctionne automatiquement avec la gestion des transactions
du serveur d'applications. D'autres conteneurs prenant en charge JPA, tels que Spring, offriront
fonctionnalité similaire mais ils nécessiteront généralement une configuration supplémentaire pour cela
travailler.

Référencer une unité de persistance

EntityManagerFactory pour une unité de persistance peut être référencé à l'aide de @
Annotation PersistenceUnit. Comme l'annotation @PersistenceContext, le nom d'unité
element identifie l'unité de persistance pour l'instance EntityManagerFactory que nous voulons
accéder. Si le nom d'unité persistante n'est pas spécifié dans l'annotation, il s'agit du fournisseur-
précise comment le nom est déterminé.

Référencement 3-14 montre l'injection d'une instance EntityManagerFactory dans
un bean session avec état. Le bean crée ensuite une instance EntityManager à partir du
factory pendant le rappel du cycle de vie PostConstruct. Un EntityManagerFactory injecté
l'instance peut être stockée en toute sécurité sur n'importe quelle instance de composant. Il est thread-safe et ne
doivent être supprimés lorsque l'instance de bean est supprimée.

Liste 3-14. Injection d'une instance EntityManagerFactory

```
@Stateful
public class EmployeeService {
    @PersistenceUnit (unitName = "EmployeeService")
    private EntityManagerFactory emf;
    privé EntityManager em;

    @PostConstruct
    public void init () {
        em = emf.createEntityManager ();
    }

    // ...
}
```

EntityManagerFactory pour une unité de persistance n'est pas utilisé aussi souvent dans Java
Environnement EE car les gestionnaires d'entités injectées sont plus faciles à acquérir et à utiliser. Comme
vous verrez au chapitre 6, il existe des différences importantes entre les responsables d'entités

Psaumes 96

renvoyés de l'usine et ceux fournis par le serveur en réponse à @
Annotation PersistenceContext.

Référencer les ressources du serveur

L'annotation @Resource est la référence fourre-tout pour les types de ressources Java EE qui ne ont des annotations dédiées. Il est utilisé pour définir des références aux usines de ressources, aux données sources et autres ressources du serveur. L'annotation @Resource est également la plus simple à définir car le seul élément supplémentaire est resourceType, qui vous permet de spécifier le type de ressource si le serveur ne peut pas le déterminer automatiquement. Par exemple, si le champ vous injectez est de type Object, alors il n'y a aucun moyen pour le serveur de savoir que vous vouliez plutôt une source de données. L'élément resourceType peut être défini sur javax.sql. DataSource pour rendre le besoin explicite.

L'une des caractéristiques de l'annotation @Resource est qu'elle est utilisée pour acquérir des ressources spécifiques au type de composant. Cela inclut les implémentations EJBContext ainsi que des services tels que le service de minuterie EJB. Sans le définir comme tel, nous utilisé l'injection de setter pour acquérir l'instance EJBContext dans le Listing 3-10 . Faire cet exemple est terminé, l'annotation @Resource aurait pu être placée sur le méthode setSessionContext (). Référencement3-15 revisite l'exemple de la liste3-10 , cette fois, démonstration de l'injection de champ avec @Resource pour acquérir un SessionContext exemple.

Liste 3-15. Injection d'une instance SessionContext

```
@Apatride
@EJB (nom = "audit", beanInterface = AuditService.class)
public class DeptService {
    @Ressource
    Contexte SessionContext;
    AuditService audit;

    @PostConstruct
    public void init () {
        audit = (AuditService) context.lookup ("audit");
    }

    // ...
}
```


Épisode 97

CDI et injection contextuelle

Bien que les installations d'injection de base de la plate-forme soient utiles, elles sont clairement limitées les termes de ce qui peut être injecté et du niveau de contrôle pouvant être exercé sur l'injection processus. CDI fournit un standard d'injection plus puissant qui étend d'abord la notion d'un bean géré et d'une injection de ressources de plate-forme, puis définit un ensemble de services d'injection supplémentaires disponibles pour les beans gérés par CDI. Bien sûr, le la caractéristique clé de l'injection contextuelle est la possibilité d'injecter une instance d'objet donnée selon le contexte actuellement actif.

Les capacités de CDI sont vastes et étendues, et bien au-delà portée d'un livre sur JPA. Pour les besoins de ce livre, nous ne rayons que la surface et montre comment créer et utiliser des beans CDI simples avec des qualificatifs. Nous suggérons que intéressé les lecteurs se réfèrent à certains des nombreux livres écrits sur CDI pour en savoir plus sur intercepteurs, décorateurs, événements et les nombreuses autres fonctionnalités disponibles dans un CDI récipient.

Haricots CDI

L'un des avantages des EJB est qu'ils fournissent tous les services dont on pourrait avoir besoin, de la sécurité à la gestion automatique des transactions et au contrôle de la concurrence. cependant, le modèle de service complet peut être considéré comme un inconvénient si vous n'utilisez pas ou ne voulez pas services, puisque la perception est qu'il y a au moins un certain coût associé à leur. Les beans gérés et les extensions CDI qui leur sont associés fournissent davantage modèle you-go. Vous n'obtenez que les services que vous spécifiez. Mais ne vous laissez pas berner par pensant que les beans CDI sont moins volumineux qu'un EJB moderne. Quand il s'agit de implémentation, les deux types d'objets sont mandatés par le conteneur dans à peu près de la même manière et les hooks de service seront ajoutés et déclenchés si nécessaire.

Que sont les beans CDI, de toute façon? Un *bean CDI* est une classe qui se qualifie pour le CDI services d'injection, dont la première exigence est simplement qu'il s'agisse d'un classe. [2] Même les beans session peuvent être des beans CDI et donc se qualifier pour les services d'injection CDI, bien qu'il y ait quelques mises en garde concernant leurs contextes de cycle de vie.

[2] Classes internes non statiques exclues.

Psaumes 98

Injection et résolution

Un bean peut avoir ses champs ou propriétés être la cible de l'injection s'ils sont annotés par @ javax.inject.Inject. CDI définit un algorithme sophistiqué pour résoudre les type d'objet à injecter, mais dans le cas général, si vous avez un champ déclaré de type X, alors une instance de X y sera injectée. Nous pouvons réécrire l'exemple dans la liste3-10 à utiliser Beans CDI gérés simples au lieu d'EJB. Le Listing 3-16 montre l'utilisation de l'injection annotation sur un champ. L'instance AuditService sera injectée après le DeptService l'instance est instanciée.

Liste 3-16. Bean CDI avec injection de champ

```
public class DeptService {
    Audit @Inject AuditService;

    // ...
}
```

L'annotation pourrait également être placée sur une méthode de propriété pour employer un setter injection. Encore une autre façon d'obtenir le même résultat consiste à utiliser un troisième type d'injection appelée *injection de constructeur* . Comme son nom l'indique, l'injection de constructeur implique le container appelant la méthode constructeur et injectant les arguments. Référencement3-17 montre comment l'injection de constructeur peut être configurée. L'injection constructeur est particulièrement bon pour tester en dehors du conteneur car un framework de test peut facilement faire le injection nécessaire en appelant simplement le constructeur sans avoir à proxy l'objet.

Liste 3-17. Bean CDI avec injection de constructeur

```
public class DeptService {
    audit privé AuditService;

    @Inject DeptService (audit AuditService) {
        this.audit = audit;
    }
    // ...
}
```

Psaumes 99

Portées et contextes

Une portée définit une paire de points de démarcation temporelle: un début et une fin. Pour
Par exemple, une portée de requête commencerait lorsque la requête démarre et se terminerait lorsque la réponse
a été renvoyé. De même, d'autres étendues définissent des durées en fonction des actions du client
et conditions. Il existe cinq étendues prédéfinies, dont trois (requête, session et
application) sont définis par la spécification du servlet, une autre (conversation) qui était
ajouté par CDI, et un autre qui a été ajouté par la spécification JTA:

- *Requête* : délimitée par une requête de méthode client spécifique.
- *Session* : démarre au lancement depuis un client HTTP et se termine le
  fin de la session HTTP. Partagé par toutes les demandes dans le même
  Session HTTP.
- *Application* : globale à l'ensemble de l'application tant qu'elle est active.
- *Conversation* : couvre une série de requêtes JSF séquentielles.
- *Transaction* : correspond à la durée de vie d'une transaction JTA active.

Un type de bean est associé à une portée en annotant la classe de bean avec le
l'annotation de portée prévue. Les instances gérées de ce bean auront un cycle de vie similaire
à la portée déclarée.

Chaque étendue active sera associée à un contexte actuel. Par exemple,
lorsqu'une demande arrive, un contexte de demande sera créé pour cette étendue de demande. Chaque
la requête aura son propre contexte de requête actuel, mais il n'y aura qu'une seule session
contexte de portée pour toutes les demandes provenant de la même session HTTP. Le contexte est
uniquement l'endroit où résident les instances étendues pendant la durée de l'étendue. Il peut
être une seule instance de chaque type de bean dans chaque contexte.

Le bean AuditService de portée d'application qui serait injecté dans le
Bean DeptService dans les listes 3-16 ou 3-17 ressemblerait à ceci:

```
@ApplicationScoped
Public class AuditService {...}
```

Comme il s'agit d'une application, une seule instance serait créée par CDI et
résident dans le contexte de portée de l'application. Notez que ceci est similaire à un EJB singleton dans
qu'une seule instance serait créée et utilisée par l'ensemble de l'application.

Piste 100

Une portée supplémentaire, *dépendante* , existe également, mais c'est vraiment l'absence de portée. Sinon
portée est spécifiée, alors la portée dépendante est supposée, ce qui signifie qu'aucune instance de
le bean est placé dans n'importe quel contexte et une nouvelle instance est créée chaque fois qu'une injection de
ce type de haricot se produit. L'annotation @Dependent peut être utilisée pour marquer explicitement un bean

comme dépendant. Les annotations de portée sont définies dans le javax.enterprise.context paquet.

Astuce Un descripteur beans.xml était requis dans CDI 1.0. Dans CDI 1.1, les beans peuvent à la place être annoté avec une annotation définissant le bean (c'est-à-dire, une annotation de portée) pour empêchez le descripteur beans.xml d'être requis. Dans les exemples qui utilisent CDI, nous utilisons des annotations de portée pour éviter d'avoir à spécifier un fichier beans.xml.

Injection qualifiée

Un qualificatif est une annotation utilisée pour contraindre ou distinguer un type de bean de d'autres types de bean qui ont le même type d'interface hérité ou implémenté. Un qualificatif peut aider le conteneur à déterminer le type de bean à injecter.

La classe d'annotation de qualificatif est généralement définie par l'application, puis utilisée pour annoter une ou plusieurs classes de bean. La définition d'annotation de qualificatif doit être annoté avec la méta-annotation @Qualifier, définie dans le package javax.inject.
Un exemple de définition d'une annotation de qualificatif se trouve dans le Listing 3-18.

Liste 3-18. Définition d'annotation de qualificatif

```
@Qualificatif
@Target ({METHODE, FIELD, PARAMETER, TYPE})
@Retention (RUNTIME)
public @interface Secure {}
```

Cette annotation peut désormais être utilisée sur une classe de bean, telle que la nouvelle Bean SecureDeptService affiché dans la liste 3-19. Le qualificatif indique que le bean est un une variété sécurisée de DeptService, par opposition à un standard (non sécurisé?).

Épisode 101

Liste 3-19. Classe de haricots qualifiés

```
@Sécurise
classe publique SecureDeptService étend DeptService {...}
```

Lorsqu'un DeptService doit être injecté dans un autre composant, le qualificatif peut être placé au site d'injection et le CDI activera son algorithme de résolution pour déterminer quelle instance créer. Il correspondra aux qualificatifs pour utiliser SecureDeptService au lieu de DeptService. Le listing 3-20 montre comment un servlet, par exemple, peut injecter un Secure DeptService sans même avoir à connaître le nom de la sous-classe.

Liste 3-20. Injection qualifiée

```
La classe publique LoginServlet étend HttpServlet {

    @Inject @Secure DeptService deptService;
    // ...
}
```

Méthodes et champs du producteur

Lorsque le conteneur a besoin d'injecter une instance de type bean, il cherchera d'abord dans le

contextes actuels pour voir s'il en existe déjà un. S'il n'en trouve pas, il doit en obtenir ou
créer une nouvelle instance. CDI permet à l'application de contrôler l'instance qui
obtient "créé" pour un type donné en utilisant une méthode ou un champ producteur.

Une méthode de producteur est une méthode que le conteneur CDI appellera pour obtenir une nouvelle instance de bean. L'instance peut être instanciée par la méthode du producteur ou le producteur peut l'obtenir par d'autres moyens; c'est entièrement à la implémentation de la méthode du producteur comment elle produit l'instance. Producteur les méthodes peuvent même décider en fonction des conditions d'exécution de renvoyer un bean différent sous-classe.

Une méthode de producteur peut être annotée avec un qualificatif. Puis quand un site d'injection est qualifié de la même manière, le conteneur appellera ce producteur qualifié correspondant méthode pour obtenir l'instance.

Dans la liste 3-21 , nous montrons une méthode de producteur qui renvoie de nouvelles instances sécurisées de DeptService. Cela pourrait être utile, par exemple, si nous ne pouvions pas modifier le SecureDeptService pour l'annoter avec l'annotation @Secure comme nous l'avons fait dans Référencement 3-19. Nous pourrions plutôt déclarer une méthode de producteur qui retournait des instances de

Épisode 102

SecureDeptService et ils seraient injectés dans des champs qualifiés avec @Secure (comme comme celui montré dans la liste 3-20). Notez qu'une méthode de producteur peut être sur n'importe quel haricot. Il peut s'agir d'une méthode statique ou d'instance, mais elle doit être annotée avec @Produces. Nous avons créé une classe de bean nommée ProducerMethods pour contenir la méthode du producteur et séparez-le du reste de la logique d'application.

Liste 3-21. Méthode du producteur

```
public class ProducerMethods {

    @Produit @Secure
    DeptService secureDeptServiceProducer () {
        retourne le nouveau SecureDeptService ();
    }
}
```

Les champs de producteur fonctionnent de la même manière que les méthodes de producteur sauf que le conteneur accède au champ pour obtenir l'instance au lieu d'appeler une méthode. C'est à la application pour s'assurer que le champ contient une instance lorsque le conteneur en a besoin. Tu verra un exemple d'utilisation d'un champ producteur dans la section suivante.

Utilisation des méthodes Producer avec les ressources JPA

Maintenant que vous connaissez certaines des bases de CDI, il est temps d'apprendre comment CDI peut être utilisé pour aider à gérer les unités et les contextes de persistance JPA. Vous pouvez utiliser une combinaison de Java EE injection de ressources avec des champs de producteurs CDI et des qualificatifs pour injecter et maintenir votre contextes de persistance.

Supposons que nous ayons deux unités de persistance, l'une nommée Employee et l'autre nommé Audit. Nous voulons utiliser les beans CDI et l'injection contextuelle. Nous créons une classe nommé EmProducers et utilisez les champs de producteur et l'injection de ressources Java EE pour obtenir les responsables d'entités. Le code du producteur et les définitions d'annotation de qualificatif sont dans Annonce 3-22.

Épisode 103

Liste 3-22. Définitions des annotations de classe et de qualificatif de producteur

```
@RequestScoped
classe publique EmProducers {

    @Produces @EmployeeEM
    @PersistenceContext (unitName = "Employé")
    privé EntityManager em1;

    @Produces @AuditEM
    @PersistenceContext (unitName = "Audit")
    privé EntityManager em2;

}

@Qualificatif
@Target ({METHODE, FIELD, PARAMETER, TYPE})
@Retention (RUNTIME)
public @interface EmployeeEM {}

@Qualificatif
@Target ({METHODE, FIELD, PARAMETER, TYPE})
@Retention (RUNTIME)
public @interface AuditEM {}
```

L'annotation Java EE @PersistenceContext entraînera le conteneur Java EE
injecter les champs producteurs avec le gestionnaire d'entités pour l'unité de persistance donnée. le
les champs producteurs seront ensuite utilisés par le CDI pour obtenir le gestionnaire d'entités avec les
qualificatif du site d'injection. Étant donné que la classe de producteur a une portée de demande, nouvelle entité
des gestionnaires seront utilisés pour chaque demande. La portée appropriée dépendra évidemment
sur l'architecture de l'application. La seule chose à faire est d'avoir un haricot injecté
avec ces responsables d'entités. Un bean DeptService correspondant qui fait référence au
les gestionnaires d'entités sont présentés dans le Listing 3-23 . Une même approche pourrait tout aussi bien être utilisée pour
injecter une fabrique de gestionnaire d'entités à l'aide de l'annotation de ressource @PersistenceUnit.

Épisode 104

Liste 3-23. Bean DeptService avec champs EntityManager injectés

```
public class DeptService {

    @Inject @EmployeeEM
    privé EntityManager empEM;

    @Inject @AuditEM
    auditEM privé EntityManager;
```

```
    // ...
}
```

Dans la majorité de nos exemples, nous utilisons simplement l'annotation @PersistenceContext car il n'implique aucun code producteur supplémentaire. Il est également pris en charge par Spring et d'autres types de conteneurs JPA, le bean sera donc plus générique.

Gestion des transactions

Plus que tout autre type d'application d'entreprise, des applications qui utilisent la persistance nécessitent une attention particulière aux problèmes de gestion des transactions. Lorsque les transactions début, quand ils se terminent et comment le gestionnaire d'entités participe à la gestion des conteneurs les transactions sont tous des sujets essentiels pour les développeurs utilisant JPA. Les sections suivantes jeter les bases des transactions dans Java EE; nous revisitons ce sujet en détail à nouveau au chapitre 6, lorsque nous examinons le gestionnaire d'entité et comment il participe aux transactions. Les sujets avancés sur les transactions sortent du cadre de ce livre. Nous recommandons *Maîtriser le développement d'applications Java EE 8* [3] pour une discussion approfondie sur le développement Applications dans Java EE 8.

Examen des transactions

Une *transaction* est une abstraction utilisée pour regrouper une série d'opérations. Une fois que groupées, l'ensemble des opérations est traité comme une seule unité et toutes les opérations doivent réussir ou aucun d'entre eux ne peut réussir. La conséquence de certaines opérations seulement réussir, c'est produire une vision incohérente des données qui seront nuisibles

[3] Kapila Bogahapitiya, Sandeep Nair. *Maîtriser le développement d'applications Java EE 8* . Broché: 2018

Épisode 105

ou indésirable pour l'application. Le terme utilisé pour décrire si les opérations réussir ensemble ou pas du tout s'appelle l' *atomicité* , et c'est sans doute le plus important des quatre propriétés de base utilisées pour caractériser le comportement des transactions. La compréhension de ces quatre propriétés est fondamentale pour comprendre les transactions. le La liste suivante résume ces propriétés:

- *Atomicité* : soit toutes les opérations d'une transaction sont réussies, soit aucun d'entre eux ne l'est. Le succès de chaque opération individuelle est lié à le succès de tout le groupe.
- *Cohérence* : l'état résultant à la fin de la transaction respecte à un ensemble de règles qui définissent l'acceptabilité des données. Les données dans le tout le système est légal ou valide par rapport au reste des données du système.
- *Isolation* : les modifications apportées au sein d'une transaction ne sont visibles que transaction qui apporte les changements. Une fois qu'une transaction est validée les changements, ils sont visibles de manière atomique pour les autres transactions.
- *Durabilité* : les modifications apportées au cours d'une transaction perdurent au-delà l'achèvement de la transaction.

Une transaction qui répond à toutes ces exigences est considérée comme une transaction ACID (le terme ACID familier étant obtenu en combinant la première lettre de chacun des quatre Propriétés).

Toutes les transactions ne sont pas des transactions ACID, et celles qui offrent souvent flexibilité dans la réalisation des propriétés ACID. Par exemple, le niveau d'isolement est

un paramètre commun qui peut être configuré pour fournir des degrés plus lâches ou plus serrés
isolement que ce qui a été décrit précédemment. Ils sont généralement effectués pour des raisons soit
performances accrues ou, de l'autre côté du spectre, si une application a plus
exigences strictes en matière de cohérence des données. Les transactions dont nous discutons dans le contexte
de Java EE sont normalement de la variété ACID.

Transactions d'entreprise en Java

Les transactions existent en fait à différents niveaux au sein du serveur d'applications d'entreprise.
La transaction la plus basse et la plus élémentaire se situe au niveau de la ressource, qui dans notre
La discussion est supposée être une base de données relationnelle frontée par une interface DataSource. Cette
est appelée une transaction locale de ressource et équivaut à une transaction de base de données. Celles-ci

Épisode 106

les types de transactions sont manipulés en interagissant directement avec la source de données JDBC
obtenu à partir du serveur d'applications. Les transactions locales aux ressources sont moins utilisées
fréquemment dans les serveurs que les transactions de conteneur.

La transaction de conteneur plus large utilise l'API de transaction Java (JTA) qui est
disponible sur tous les serveurs d'applications Java EE compatibles et sur la plupart des sites Web Java
les serveurs. Il s'agit de la transaction typique utilisée pour les applications d'entreprise et peut
impliquer ou mobiliser un certain nombre de ressources, y compris des sources de données ainsi que d'autres types
des ressources transactionnelles. Ressources définies à l'aide de Java Connector Architecture (JCA)
les composants peuvent également être inscrits dans la transaction de conteneur.

Les conteneurs ajoutent généralement leur propre couche au-dessus de la source de données JDBC pour effectuer
des fonctions telles que la gestion des connexions et la mise en commun qui permettent une utilisation plus efficace
des ressources et fournir une intégration transparente avec la gestion des transactions
système. Ceci est également nécessaire car il est de la responsabilité du conteneur d'effectuer
l'opération de validation ou de restauration sur la source de données lorsque la transaction de conteneur
se termine.

Parce que les transactions de conteneur utilisent JTA et parce qu'elles peuvent s'étendre sur plusieurs
ressources, elles sont également appelées transactions JTA ou transactions globales. Le conteneur
la transaction est un aspect central de la programmation au sein des serveurs Java.

Démarcation de transaction

Chaque transaction a un début et une fin. Commencer une transaction permettra
opérations ultérieures pour faire partie de la même transaction jusqu'à la transaction
a completé. Les transactions peuvent être effectuées de deux manières. Ils peuvent être
validée, ce qui entraîne la persistance de toutes les modifications dans le magasin de données ou leur annulation,
indiquant que les modifications doivent être rejetées. Le fait de faire en sorte qu'une transaction
soit commencer soit terminer est appelé démarcation de transaction. C'est une partie critique de
écrire des applications d'entreprise, car une démarcation incorrecte des transactions est
l'une des sources les plus courantes de dégradation des performances.

Les transactions locales aux ressources sont toujours délimitées explicitement par l'application,
tandis que les transactions de conteneurs peuvent être soit délimitées automatiquement par le conteneur
ou en utilisant une interface JTA qui prend en charge la démarcation contrôlée par l'application. La première
cas, lorsque le conteneur prend en charge la responsabilité de la démarcation de la transaction, est
appelée gestion des transactions gérées par conteneur (CMT), mais lorsque l'application est
responsable de la démarcation, c'est ce qu'on appelle la gestion des transactions gérées par le bean (BMT).

Épisode 107

Les EJB peuvent utiliser des transactions gérées par conteneur ou des transactions gérées par bean. Les servlets sont limités à la transaction gérée par bean, quelque peu mal nommée. le le style de gestion des transactions par défaut pour un composant EJB est géré par conteneur. Pour configurer explicitement un EJB pour que ses transactions soient délimitées dans un sens ou autre, l'annotation @TransactionManagement doit être spécifiée sur la classe EJB. Le type énuméré TransactionManagementType définit BEAN pour le bean-managed transactions et CONTAINER pour les transactions gérées par conteneurs. Annonce 3-24 montre comment activer les transactions gérées par bean à l'aide de cette approche.

Liste 3-24. Modification du type de gestion des transactions d'un EJB

```
@Apatride
@TransactionManagement (TransactionManagementType.BEAN)
public class ProjectService {
    // les méthodes de cette classe contrôlent manuellement la démarcation des transactions
}
```

Remarque La gestion des transactions par défaut d'un EJB étant le conteneur géré, l'annotation @TransactionManagement doit être spécifiée uniquement si des transactions gérées par bean sont souhaitées.

Transactions gérées par conteneur

Le moyen le plus courant de délimiter les transactions consiste à utiliser des CMT, qui épargnent application l'effort et le code pour commencer et valider les transactions explicitement.

Les exigences de transaction sont déterminées par les métadonnées et sont configurables au granularité de la classe ou même d'une méthode. Par exemple, un bean session peut déclarer que chaque fois qu'une méthode spécifique sur ce bean est appelée, le conteneur doit s'assurer que une transaction est lancée avant le début de la méthode. Le conteneur est également responsable de la validation de la transaction après l'achèvement de la méthode.

Il est assez courant pour un haricot d'invoquer un autre haricot à partir d'un ou plusieurs de ses méthodes. Dans ce cas, une transaction lancée par la méthode appelante n'aura pas été validée car la méthode appelante ne sera pas terminée avant son appel à la seconde bean est terminé. C'est pourquoi nous avons besoin de paramètres pour définir comment le conteneur doit se comportent lorsqu'une méthode est appelée dans un contexte transactionnel spécifique.

Épisode 108

Par exemple, si une transaction est déjà en cours lorsqu'une méthode est appelée, le on peut s'attendre à ce que le conteneur utilise uniquement cette transaction, alors qu'il peut être dirigé vers commencer une nouvelle si aucune transaction n'est active. Ces paramètres sont appelés attributs de transaction, et ils déterminent le comportement transactionnel géré par le conteneur.

Les choix d'attributs de transaction définis sont les suivants:

- OBLIGATOIRE: si cet attribut est spécifié pour une méthode, une transaction est censé avoir déjà été démarré et être actif lorsque le méthode est appelée. Si aucune transaction n'est active, une exception est levée. Cet attribut est rarement utilisé, mais peut être un outil de développement pour attraper des erreurs de démarcation de transaction lorsqu'il est prévu qu'une transaction aurait déjà dû être démarré.

• REQUIS: cet attribut est le cas le plus courant dans lequel une méthode
devrait être dans une transaction. Le conteneur offre une garantie
qu'une transaction est active pour la méthode. Si l'un est déjà actif,
c'est utilisé; s'il n'en existe pas, une nouvelle transaction est créée pour le
exécution de la méthode.

• REQUIRES_NEW: cet attribut est utilisé lorsque la méthode a toujours besoin
être dans sa propre transaction; autrement dit, la méthode doit être validée
ou annulé indépendamment des méthodes plus haut dans la pile d'appels.
Il doit être utilisé avec prudence car il peut entraîner une
frais généraux de transaction.

• SUPPORTS: les méthodes marquées avec des supports ne dépendent pas d'un
transaction, mais tolérera l'exécution à l'intérieur d'un si elle existe. C'est un
indique qu'aucune ressource transactionnelle n'est accessible dans la méthode.

• NOT_SUPPORTED: une méthode marquée comme ne prenant pas en charge les transactions entraînera
le conteneur pour suspendre la transaction en cours si l'une est active lorsque le
méthode est appelée. Cela implique que la méthode n'effectue pas de transaction
opérations, mais pourrait échouer d'une autre manière qui pourrait affecter
résultat d'une transaction. Ce n'est pas un attribut couramment utilisé.

• JAMAIS: une méthode marquée comme ne prenant jamais en charge les transactions entraînera
le conteneur pour lever une exception si une transaction est active lorsque
la méthode est appelée. Cet attribut est très rarement utilisé, mais peut être un
outil de développement pour détecter les erreurs de démarcation des transactions
s'attendait à ce que les transactions aient déjà été conclues.

Épisode 109

Chaque fois que le conteneur démarre une transaction pour une méthode, le conteneur est supposé
essayez également de valider la transaction à la fin de la méthode. Chaque fois que le courant
la transaction doit être suspendue, le conteneur est responsable de la reprise de la suspension
transaction à l'issue de la méthode.

Il existe en fait deux manières différentes de spécifier des transactions gérées par des conteneurs,
un pour les EJB et un pour les beans CDI, les servlets, les classes de ressources JAX-RS et tous les autres
types de composants gérés par Java EE. Les EJB ont été le premier type de composant à offrir une
fonction de transaction gérée par conteneur et des métadonnées spécifiques définies à cet effet.
Beans CDI et autres types de composants Java EE, tels que les servlets ou les ressources JAX-RS
classes, utilisez un intercepteur transactionnel.

Transactions gérées par des conteneurs EJB

L'attribut de transaction d'un EJB peut être indiqué en annotant la classe EJB,
ou l'une de ses méthodes qui fait partie de l'interface métier optionnelle, avec le @
Annotation TransactionAttribute. Cette annotation nécessite un seul argument du
type énuméré TransactionAttributeType, dont les valeurs sont définies dans le
liste précédente. L'annotation de la classe de bean entraînera l'application de l'attribut de transaction
à toutes les méthodes métier de la classe, tandis que l'annotation d'une méthode applique le
attribuer uniquement à la méthode. Si des annotations au niveau de la classe et au niveau de la méthode existent, le
l'annotation au niveau de la méthode a la priorité. En l'absence de niveau classe ou méthode
@TransactionAttribute annotations, l'attribut par défaut REQUIRED sera appliqué.

Référencement 3-25 montre comment la méthode addItem () du bean panier
Référencement 3-5 peut utiliser un attribut de transaction. Aucun paramètre de gestion des transactions n'était
fourni, les transactions gérées par le conteneur seront donc utilisées. Aucun attribut n'a été spécifié
sur la classe, donc le comportement par défaut de REQUIRED s'appliquera à toutes les méthodes du
classe. L'exception est que la méthode addItem () a déclaré un attribut de transaction
de SUPPORTS, qui remplace le paramètre REQUIRED. Chaque fois qu'un appel pour ajouter un élément est

effectuée, cet article sera ajouté au panier, mais si aucune transaction n'était active, aucune
doivent être démarrés.

Liste 3-25. Spécification d'un attribut de transaction EJB

```
@Stateful
ShoppingCart de classe publique {

    @TransactionAttribute (TransactionAttributeType.SUPPORTS)
    public void addItem (élément de chaîne, quantité entière) {
```


Épisode 110


```
            verifyItem (article, quantité);
            // ...
        }

        // ...
}
```

De plus, avant que la méthode addItem () ajoute l'article au panier, elle
validation dans une méthode privée appelée verifyItem () qui n'est pas affichée dans l'exemple.
Lorsque cette méthode est invoquée depuis addItem (), elle s'exécutera dans n'importe quelle transaction
context addItem () a été appelé.

Tout bean souhaitant faire annuler une transaction gérée par un conteneur peut le faire
donc en appelant la méthode setRollbackOnly () sur l'objet EJBContext. Bien que
cela n'entraînera pas l'annulation immédiate de la transaction, c'est une indication au
conteneur que la transaction doit être annulée une fois la transaction terminée.
Notez que les gestionnaires d'entités entraîneront également le lancement de la transaction en cours
retour lorsqu'une exception est levée lors d'un appel de gestionnaire d'entités ou lorsque le
la transaction est terminée.

Intercepteurs transactionnels

Lorsque les intercepteurs ont été introduits sur la plate-forme Java EE 6, ils ont ouvert le
possibilité future de découpler la fonction de transaction gérée par conteneur de l'EJB
et l'offrir à tout composant prenant en charge l'interception. Dans Java EE 7, le @javax.
Une annotation transactionnelle a été ajoutée pour indiquer qu'une
l'intercepteur transactionnel serait appliqué au composant cible. Le mécanisme
agit de manière similaire aux transactions gérées par conteneur EJB en ce sens qu'il est appliqué de manière déclarative
sur une classe ou une méthode et une sémantique de classe peut être remplacée par une méthode
un. La principale différence est que l'annotation @Transactional est utilisée à la place de
@TransactionAttribute, et le type de la valeur énumérée est une énumération imbriquée appelée
Transactional.TxType au lieu du TransactionAttributeType utilisé dans EJB. Celles-ci
les constantes enum sont nommées exactement de la même manière dans Transactional.TxType et ont le
même sémantique.

Peut-être la différence la plus pertinente entre les intercepteurs transactionnels
composants et EJB CMT est le fait que les composants qui utilisent transactionnel
les intercepteurs n'obtiennent pas automatiquement CMT mais doivent s'inscrire en utilisant le @Transactional
annotation. Les EJB obtiennent CMT par défaut et doivent se désinscrire s'ils préfèrent utiliser BMT.

Épisode 111

Nous pouvons maintenant réécrire la liste 3-25 pour utiliser un bean CDI au lieu d'un bean session avec état. Le haricot ShoppingCart dans la liste 3-26 a une portée de session afin que son état soit maintenu tout au long de la session. L'annotation @Transactional vide entraîne la valeur par défaut transactionnelle à définir sur REQUIRED pour toutes les méthodes de la classe, à l'exception de addItem () qui est explicitement substituée pour être SUPPORTS.

Liste 3-26. Intercepteur transactionnel dans un Bean CDI

```
@Transactionnel
@SessionScoped
ShoppingCart de classe publique {

    @Transactional (TxType.SUPPORTS)
    public void addItem (élément de chaîne, quantité entière) {
        verifyItem (article, quantité);
        // ...
    }

    // ...
}
```

Astuce @Transactional peut être utilisé sur des beans gérés ou des beans CDi mais ne pas être utilisé sur les eJB. Inversement, @TransactionAttribute ne peut être utilisé que sur eJB.

Transactions gérées par Bean

Une autre façon de délimiter les transactions consiste à utiliser les BMT. Cela signifie simplement que le l'application doit démarrer et arrêter les transactions explicitement en effectuant des appels d'API. Avec le à l'exception des EJB, tous les composants gérés utilisent par défaut des transactions gérées par bean et sont laissés à faire leur propre démarcation de transaction s'ils ne spécifient pas le @Transactional intercepteur. Seuls les EJB reçoivent des transactions gérées par conteneur par défaut.

Déclarer qu'un EJB utilise des transactions gérées par bean signifie que la classe de bean assume la responsabilité de commencer et de valider les transactions chaque fois qu'elle le juge il est nécessaire. Avec cette responsabilité, cependant, vient l'attente que le haricot

la classe fera les choses correctement. Les EJB qui utilisent BMT doivent s'assurer que chaque fois qu'une transaction été démarré, il doit également être terminé avant de revenir de la méthode qui l'a démarré. Si vous ne le faites pas, le conteneur annulera automatiquement la transaction et une exception est levée.

Une pénalité des transactions gérées par les EJB plutôt que par le conteneur est qu'elles ne sont pas propagées aux méthodes appelées sur un autre EJB BMT. Par exemple, si EJB A commence une transaction puis appelle EJB B, qui utilise la gestion bean transactions, la transaction ne sera pas propagée à la méthode dans l'EJB B. une transaction est active lorsqu'une méthode BMT EJB est appelée, la transaction active être suspendu jusqu'à ce que le contrôle revienne à la méthode appelante. Ce n'est pas vrai pour les non-EJB Composants.

BMT n'est généralement pas recommandé pour une utilisation dans les EJB car il ajoute de la complexité à la l'application et exige que l'application effectue le travail que le serveur peut déjà faire pour elle. Alors que d'autres types de composants peuvent utiliser des intercepteurs transactionnels s'ils le souhaitent, ils n'ont pas les mêmes restrictions BMT que les EJB, il est donc plus courant pour eux

adopter une approche BMT contrôlée par les applications. Ils ont également traditionnellement utilisé BMT car il n'y a jamais vraiment eu de choix dans le passé. Ce n'est que récemment, dans Java EE 7, que des intercepteurs de transaction ont été introduits.

UserTransaction

Pour qu'un composant puisse démarrer et valider manuellement des transactions de conteneur, l'application doit avoir une interface qui la prend en charge. Le fichier javax.transaction. L'interface UserTransaction est l'objet désigné dans le JTA que l'application les composants peuvent conserver et invoquer pour gérer les limites de transaction. Une instance de UserTransaction n'est pas réellement l'instance de transaction actuelle; c'est une sorte de proxy qui fournit l'API de transaction et représente la transaction actuelle. UNE L'instance UserTransaction peut être injectée dans des composants à l'aide de l'un des Java Annotations EE @Resource ou CDI @Inject. Une UserTransaction est également présente dans le contexte de nommage d'environnement sous le nom réservé java: comp / UserTransaction. L'interface UserTransaction est présentée dans le Listing 3-27.

Liste 3-27. L'interface UserTransaction

```
interface publique UserTransaction {
        public abstract void begin ();
        public abstract void commit ();
```

Épisode 113


```
        résumé public int getStatus ();
        public abstract void rollback ();
        public abstract void setRollbackOnly ();
        public abstract void setTransactionTimeout (int secondes);
}
```

Chaque transaction JTA est associée à un thread d'exécution, il s'ensuit donc que pas plus d'une transaction ne peut être active à un moment donné. Donc, si une transaction est actif, l'utilisateur ne peut pas en démarrer un autre dans le même thread tant que le premier n'a pas commis ou annulé. Alternativement, la transaction peut expirer, provoquant le transaction à annuler.

Nous avons discuté précédemment que dans certaines conditions CMT, le conteneur sera suspendu la transaction en cours. De l'API précédente, vous pouvez voir qu'il n'y a pas UserTransaction méthode pour suspendre une transaction. Seul le conteneur peut le faire à l'aide d'une API de gestion des transactions interne. De cette manière, plusieurs transactions peuvent être associé à un seul thread, même si un seul peut être actif à la fois.

Les restaurations peuvent se produire dans plusieurs scénarios différents. La méthode setRollbackOnly () indique que la transaction en cours ne peut pas être validée, laissant la restauration comme seule issue possible. La transaction peut être annulée immédiatement en appelant le méthode rollback (). Alternativement, une limite de temps pour la transaction peut être définie avec le setTransactionTimeout (), provoquant l'annulation de la transaction lorsque la limite est atteint. Le seul problème avec les délais d'expiration des transactions est que la limite de temps doit être définie avant le début de la transaction et il ne peut pas être modifié une fois la transaction en cours.

Dans JTA, chaque thread a un statut transactionnel accessible via le Appel à getStatus (). La valeur de retour de cette méthode est l'une des constantes définies sur l'interface javax.transaction.Status. Si aucune transaction n'est active, par exemple, alors la valeur retournée par getStatus () sera le STATUS_NO_TRANSACTION. De même, si setRollbackOnly () a été appelé sur la transaction en cours, alors le statut sera STATUS_MARKED_ROLLBACK jusqu'à ce que la transaction ait commencé à revenir en arrière.

Référencement 3-28 montre un fragment d'un servlet utilisant le bean ProjectService pour démontrer l'utilisation de UserTransaction pour appeler plusieurs méthodes EJB dans un même transaction. La méthode doPost () utilise l'instance UserTransaction injectée avec le

Annotation @Resource pour démarrer et valider une transaction. Notez le bloc try… catch requis autour des opérations de transaction pour s'assurer que la transaction est correctement nettoyé en cas de panne. Nous interceptons l'exception et la relançons ensuite dans un exception d'exécution après avoir effectué la restauration.

Psaumes 114

Liste 3-28. Utilisation de l'interface UserTransaction

```
La classe publique ProjectServlet étend HttpServlet {
    @Resource UserTransaction tx;
    @EJB ProjectService bean;

    protected void doPost (requête HttpServletRequest, HttpServletResponse
    réponse)
        lance ServletException, IOException {
        // ...

        essayez {
            tx.begin ();
            bean.assignEmployeeToProject (projectId, empId);
            bean.updateProjectStatistics ();
            tx.commit ();
        } catch (Exception e) {
            // Essayez de revenir en arrière (peut échouer si l'exception provient de begin () ou
            commit (), mais c'est ok)
            essayez {tx.rollback (); } catch (Exception e2) {}
            lancer une nouvelle MyRuntimeException (e);
        }

        // ...
    }
}
```

Mettre tous ensemble

Maintenant que nous avons discuté du modèle de composant d'application et des services disponibles en tant que partie d'un serveur d'applications Java EE, nous pouvons revoir l'exemple EmployeeService de le chapitre précédent et apportez-le à l'environnement Java EE. En cours de route, nous fournissons exemple de code pour montrer comment les composants s'emboîtent et comment ils se rapportent au Exemple Java SE.

Épisode 115

Définition du composant

Pour commencer, considérons la définition de la classe EmployeeService de Listing 2-9

au chapitre 2 . L'objectif de cette classe est de fournir des opérations commerciales liées à la maintenance des données des employés. Ce faisant, il encapsule toute la persistance opérations. Pour introduire cette classe dans l'environnement Java EE, nous devons d'abord décider comment il devrait être représenté. Le modèle de service présenté par la classe suggère un bean session ou composant similaire. Parce que les méthodes commerciales du haricot ont pas de dépendance les uns des autres, nous pouvons en outre décider que tout haricot apatride, tel qu'un bean session sans état, convient. En fait, ce haricot présente un design très typique motif appelé *façade de session,*[4] dans lequel un bean session sans état est utilisé pour protéger les clients de traiter une API de persistance particulière. Pour transformer la classe EmployeeService en un bean session sans état, il suffit de l'annoter avec @Stateless.

Dans l'exemple Java SE, la classe EmployeeService doit créer et gérer son propre instance de gestionnaire d'entités. Nous pouvons remplacer cette logique par l'injection de dépendances pour acquérir automatiquement le gestionnaire d'entités. Après avoir décidé d'un bean session apatride et injection de dépendances, le bean session sans état converti est illustré dans le Listing 3-29. À l'exception de la manière dont le gestionnaire d'entités est acquis, le reste de la classe est identique. Il s'agit d'une fonctionnalité importante de l'API Java Persistence car le même EntityManager L'interface peut être utilisée à la fois à l'intérieur et à l'extérieur du serveur d'applications.

Liste 3-29. Le bean de session EmployeeService

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    protected EntityManager em;

    EntityManager getEntityManager () {
        retournez-les;
    }

    public Employee createEmployee (int id, nom de chaîne, salaire long) {
        Employé emp = nouvel employé (id);
        emp.setName (nom);
```

[4] Alur et al., *Core J2EE Patterns* .

Épisode 116


```
        emp.setSalary (salaire);
        getEntityManager (). persist (emp);
        return emp;
    }

    public void removeEmployee (int id) {
        Employé emp = findEmployee (id);
        if (emp! = null) {
            getEntityManager (). remove (emp);
        }
    }

    public Employee changeEmployeeSalary (int id, long newSalary) {
        Employé emp = findEmployee (id);
        if (emp! = null) {
            emp.setSalary (newSalary);
        }
        return emp;
    }

    public Employee findEmployee (int id) {
```

```
        return getEntityManager (). find (Employee.class, id);
    }

    Liste publique <Employee> findAllEmployees () {
        TypedQuery query = getEntityManager (). CreateQuery ("SELECT e FROM
        Employee e ", Employee.class);
        return query.getResultList ();
    }
}
```

Définition de l'interface utilisateur

La prochaine question à considérer est de savoir comment accéder au bean. Une interface Web est un méthode de présentation commune pour les applications d'entreprise. Pour démontrer comment cela Un bean session sans état peut être utilisé par un servlet, consultez le Listing 3-30. La demande les paramètres sont interprétés pour déterminer l'action, qui est ensuite effectuée par

Épisode 117

appel de méthodes sur le bean EmployeeService injecté. Bien que seulement le premier est décrite, vous pouvez voir comment cela pourrait facilement être étendu pour gérer chacun des opérations définies sur EmployeeService.

Liste 3-30. Utilisation du bean de session EmployeeService à partir d'un servlet

```
Public class EmployeeServlet étend HttpServlet {
    @EJB EmployeeService bean;

    protected void doPost (requête HttpServletRequest,
                           Réponse HttpServletResponse) {
        String action = request.getParameter ("action");

        if (action.equals ("créer")) {
            String id = request.getParameter ("id");
            String name = request.getParameter ("name");
            String salaire = request.getParameter ("salaire");
            bean.createEmployee (Integer.parseInt (id), nom,
                                 Long.parseLong (salaire));
        }

        // ...
    }
}
```

Emballer

Dans l'environnement Java EE, de nombreuses propriétés requises dans le fichier persistence.xml pour Java SE peut être omis. Dans la liste3-31 , vous voyez le fichier persistence.xml de la liste2 à 11 converti pour le déploiement dans le cadre d'une application Java EE. Au lieu des propriétés JDBC pour créer une connexion, nous déclarons maintenant que le gestionnaire d'entités doit utiliser les données nom de la source jdbc / EmployeeDS. Si la source de données a été définie comme étant disponible dans le espace de noms d'application au lieu du contexte de dénomination du composant local, nous pourrions à la place, utilisez le nom de source de données java: app / jdbc / EmployeeDS. La transaction- L'attribut type a également été supprimé pour permettre à l'unité de persistance de passer par défaut à JTA. le le serveur d'application trouvera automatiquement les classes d'entités, de sorte que même la liste des classes a été supprimé. Cet exemple représente la configuration Java EE minimale idéale.

Épisode 118

Parce que la logique métier qui utilise cette unité de persistance est implémentée dans un bean session sans état, le fichier persistence.xml se trouverait généralement dans le Répertoire META-INF du JAR EJB correspondant, ou WEB-INF / classes / META-INF répertoire de la GUERRE. Nous décrivons complètement le fichier persistence.xml et son emplacement dans une application Java EE au chapitre 14.

Liste 3-31. Définition d'une unité de persistance dans Java EE

```
<persistance>
    <persistence-unit name = "EmployeeService">
        <jta-data-source> jdbc / EmployeeDS </jta-data-source>
    </persistence-unit>
</persistence>
```

Résumé

Il serait impossible de fournir des détails sur toutes les fonctionnalités de la plate-forme Java EE dans un chapitre unique. Cependant, nous ne pouvons pas mettre JPA en contexte sans expliquer l'application environnement de serveur dans lequel il sera utilisé. À cette fin, nous avons introduit les technologies qui sont les plus pertinentes pour le développeur utilisant la persistance en entreprise applications.

Nous avons commencé par une introduction aux composants logiciels d'entreprise et avons introduit le modèle de composant EJB. Nous avons fait valoir que l'utilisation de composants est plus importante que jamais et a identifié certains des avantages qui découlent de leur mise à profit. Nous avons présenté les principes fondamentaux des beans session sans état, avec état et singleton et a montré la syntaxe pour les déclarer ainsi que la différence de style d'interaction entre eux.

Nous avons ensuite examiné la gestion des dépendances dans les serveurs d'applications Java EE. nous discuté des types d'annotations de référence et comment les déclarer. Nous avons également regardé à la différence entre la recherche de dépendances et l'injection de dépendances. Dans le cas d'injection, nous avons examiné la différence entre l'injection de champ et l'injection de setter. Nous avons ensuite a exploré chacun des types de ressources, montrant comment acquérir et injecter des serveurs et Ressources JPA.

En nous appuyant sur l'injection de ressources Java EE, nous avons ensuite présenté le modèle CDI avec sa notion généralisée de bean géré. Nous avons répertorié les portées et a expliqué les contextes que CDI utilise pour mettre en cache les instances contextuelles qu'il

Épisode 119

injecte. Nous avons montré comment les qualificatifs peuvent apporter des contraintes supplémentaires à l'injection processus de résolution et comment les producteurs peuvent être définis pour renvoyer les instances au CDI conteneur utilise pour l'injection. Nous avons ensuite démontré une façon d'utiliser les producteurs pour injecter ressources de persistance qualifiées.

Dans la section sur la gestion des transactions, nous avons examiné JTA et son rôle dans créer des applications centrées sur les données. Nous avons ensuite examiné la différence entre le haricot transactions gérées et transactions gérées par conteneur pour les EJB et les non-EJB. nous

documenté les différents types d'attributs de transaction pour les beans CMT et montré comment pour contrôler manuellement les transactions gérées par le bean.

Nous avons conclu le chapitre en explorant comment utiliser les composants Java EE avec JPA en convertissant l'exemple d'application présenté dans le chapitre précédent à partir d'un application Java SE de ligne de commande vers une application Web exécutée sur une application serveur.

Maintenant que nous avons introduit JPA dans les environnements Java SE et Java EE, il il est temps de se plonger dans la spécification en détail. Dans le chapitre suivant, nous commençons ce voyage avec l'objectif central de JPA: le mapping objet-relationnel.

Épisode 120

CHAPITRE 4

Mappage objet-relationnel

La plus grande partie d'une API qui persiste des objets dans une base de données relationnelle finit par être le composant de mappage objet-relationnel (ORM). Le sujet de l'ORM comprend généralement tout, de la façon dont l'état de l'objet est mappé aux colonnes de la base de données à la façon de lancer requêtes sur les objets. Nous concentrons ce chapitre principalement sur la manière de définir et mapper l'état de l'entité à la base de données, en mettant l'accent sur la manière simple dont cela peut être fait.

Ce chapitre présente les bases du mappage des champs aux colonnes de la base de données, puis continue en montrant comment mapper et générer automatiquement des identifiants d'entité. Nous entrons dans quelques détails sur les différents types de relations et illustrent la façon dont elles sont cartographiées du modèle de domaine au modèle de données.

Les fonctionnalités ORM les plus importantes sont:

- *Persistance idiomatique* : en permettant d'écrire les classes de persistance en utilisant des classes orientées objet.
- *Haute performance* : en activant les techniques de récupération et de verrouillage.
- *Fiable* : en permettant la stabilité pour les programmeurs JPA.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_4

Épisode 121

Figure 4-1 montre l'architecture JPA ORM.

Figure 4-1. *Architecture JPA ORM*

Annotations de persistance

Nous avons montré dans les chapitres précédents comment les annotations ont été largement utilisées à la fois dans les spécifications EJB et JPA. Nous discutons de la persistance et du mappage des métadonnées dans détail significatif, et parce que nous utilisons des annotations pour expliquer les concepts, cela vaut la peine examiner quelques éléments sur les annotations avant de commencer.

Les annotations de persistance peuvent être appliquées à trois niveaux différents: classe, méthode et champ. Pour annoter l'un de ces niveaux, l'annotation doit être placée devant le code définition de l'artefact annoté. Dans certains cas, nous les mettons sur la même ligne juste avant la classe, la méthode ou le champ; dans d'autres cas, nous les mettons sur la ligne ci-dessus. le le choix est entièrement basé sur les préférences de la personne appliquant les annotations, et nous pensons qu'il est logique de faire une chose dans certains cas et une autre dans d'autres cas. Il dépend de la longueur de l'annotation et du format le plus lisible.

Les annotations JPA ont été conçues pour être lisibles, faciles à spécifier et flexibles assez pour permettre différentes combinaisons de métadonnées. La plupart des annotations sont spécifiées comme frères et sœurs au lieu d'être imbriqués les uns dans les autres, ce qui signifie que plusieurs annotations peut annoter la même classe, champ ou propriété au lieu d'avoir des annotations incorporées dans d'autres annotations. Comme pour tous les compromis, le joueur de cornemuse doit être payé, cependant, et le coût de la flexibilité est que de nombreuses permutations possibles de métadonnées de premier niveau seront syntaxiquement correcte mais sémantiquement invalide. Le compilateur ne sera d'aucune utilité, mais le Le moteur d'exécution du fournisseur effectue souvent une vérification de base pour les regroupements d'annotations incorrects.

Épisode 122

Cependant, la nature des annotations est que lorsqu'elles sont inattendues, elles
juste ne pas se faire remarquer du tout. Cela vaut la peine de s'en souvenir lorsque vous essayez de comprendre
comportement qui pourrait ne pas correspondre à ce que vous pensiez avoir spécifié dans les annotations. Il
peut être qu'une ou plusieurs annotations sont ignorées.

Les annotations de mappage peuvent être classées dans l'une des deux catégories suivantes:
annotations logiques et annotations physiques. Les annotations du groupe logique sont
ceux qui décrivent le modèle d'entité à partir d'une vue de modélisation d'objet. Ils sont étroitement
liés au modèle de domaine et sont le type de métadonnées que vous voudrez peut-être spécifier
en UML ou tout autre langage ou framework de modélisation d'objets. Les annotations physiques
se rapportent au modèle de données concret dans la base de données. Ils traitent des tableaux, des colonnes,
contraintes et autres artefacts au niveau de la base de données que le modèle objet pourrait ne jamais être
conscient du contraire.

Nous utilisons les deux types d'annotations tout au long des exemples et pour démontrer
les métadonnées de mappage. Comprendre et être capable de faire la distinction entre ces
deux niveaux de métadonnées vous aideront à décider où déclarer les métadonnées,
et où utiliser les annotations et XML. Comme vous le verrez au chapitre 13 , il existe du XML
équivalents à toutes les annotations de mappage décrites dans ce chapitre, vous donnant les
la liberté d'utiliser l'approche qui correspond le mieux à vos besoins de développement.

Accès à l'état de l'entité

L'état mappé d'une entité doit être accessible au fournisseur au moment de l'exécution, de sorte que
quand vient le temps d'écrire les données, elles peuvent être obtenues à partir de l'instance d'entité
et stocké dans la base de données. De même, lorsque l'état est chargé à partir de la base de données, le
Le moteur d'exécution du fournisseur doit pouvoir l'insérer dans une nouvelle instance d'entité. La façon dont l'état est
accédé dans l'entité est appelé le *mode d'accès.*

Au chapitre 2, vous avez appris qu'il existe deux manières différentes de spécifier une entité persistante
state: vous pouvez soit annoter les champs, soit annoter les propriétés de style JavaBean. le
le mécanisme que vous utilisez pour désigner l'état persistant est le même que le mode d'accès
que le fournisseur utilise pour accéder à cet état. Si vous annotez des champs, le fournisseur obtiendra
et définissez les champs de l'entité à l'aide de la réflexion. Si les annotations sont définies sur le getter
méthodes de propriétés, ces méthodes getter et setter seront appelées par le fournisseur pour
accéder et définir l'état.

Épisode 123

Accès sur le terrain

L'annotation des champs de l'entité obligera le fournisseur à utiliser l'accès aux champs pour obtenir et
définir l'état de l'entité. Les méthodes d'obtention et de définition peuvent ou non être présentes, mais
s'ils sont présents, ils sont ignorés par le fournisseur. Tous les champs doivent être déclarés comme
protégé, package ou privé. Les champs publics ne sont pas autorisés car ils s'ouvriraient
les champs d'état pour accéder par toute classe non protégée dans la machine virtuelle. Le faire n'est pas simplement un
manifestement mauvaise pratique mais pourrait également faire échouer l'implémentation du fournisseur. Bien sûr, le
d'autres qualificatifs n'empêchent pas les classes dans le même package ou hiérarchie de faire
la même chose, mais il y a un compromis évident entre ce qui devrait être contraint

et ce qui devrait être recommandé. Les autres classes doivent utiliser les méthodes d'une entité afin d'accéder à son état persistant, et même la classe d'entité elle-même ne devrait vraiment manipuler les champs directement lors de l'initialisation.

L'exemple du Listing 4-1 montre l'entité Employé mappée à l'aide du champ accès. L'annotation @Id indique non seulement que le champ id est l'identifiant persistant ou clé primaire pour l'entité, mais aussi que l'accès au champ doit être supposé. Le nom et les champs de salaire sont alors définis par défaut pour être persistants, et ils sont mappés aux colonnes de Le même nom.

Liste 4-1. Utilisation de Field Access

```
@Entité
Employé de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    long salaire privé;

    public long getId () {return id; }
    public void setId (long id) {this.id = id; }

    public String getName () {nom de retour; }
    public void setName (String name) {this.name = nom; }

    public long getSalary () {salaire de retour; }
    public void setSalary (salaire long) {this.salary = salaire; }
}
```


Épisode 124

Accès à la propriété

Lorsque le mode d'accès aux propriétés est utilisé, le même contrat que pour JavaBeans s'applique, et il doit y avoir des méthodes getter et setter pour les propriétés persistantes. Le type de property est déterminé par le type de retour de la méthode getter et doit être identique à le type du paramètre unique passé dans la méthode setter. Les deux méthodes doivent avoir visibilité publique ou protégée. Les annotations de mappage pour une propriété doivent être sur la méthode getter.

Dans la liste 4-2 , la classe Employee a une annotation @Id sur le getId () afin que le fournisseur utilise l'accès à la propriété pour obtenir et définir l'état de l'entité. Les propriétés de nom et de salaire seront rendues persistantes grâce au getter et les méthodes de définition qui existent pour eux et qui seront mappées aux colonnes NAME et SALARY, respectivement. Notez que la propriété de salaire est soutenue par le champ salaire, ce qui pas partager le même nom. Cela passe inaperçu par le fournisseur car en spécifiant accès à la propriété, nous demandons au fournisseur d'ignorer les champs d'entité et d'utiliser uniquement le méthodes getter et setter pour la dénomination.

Liste 4-2. Utilisation de l'accès à la propriété

```
@Entité
Employé de classe publique {
    identifiant long privé;
    nom de chaîne privé;
    salaire long privé;

    @Id public long getId () {return id; }
    public void setId (long id) {this.id = id; }
```

```
    public String getName () {nom de retour; }
    public void setName (String name) {this.name = nom; }

    public long getSalary () {salaire de retour; }
    public void setSalary (salaire long) {this.wage = salaire; }
}
```


Épisode 125

Accès mixte

Il est également possible de combiner l'accès au champ avec l'accès à la propriété au sein de la même entité hiérarchie, voire au sein de la même entité. Ce ne sera pas un événement très courant, mais peut être utile, par exemple, lorsqu'une sous-classe d'entité est ajoutée à une hiérarchie existante qui utilise un type d'accès différent. Ajout d'une annotation @Access avec un accès spécifié le mode sur l'entité de sous-classe entraînera le remplacement du type d'accès par défaut pour cela sous-classe d'entité.

L'annotation @Access est également utile lorsque vous devez effectuer un simple transformation en données lors de la lecture ou de l'écriture dans la base de données. Habituellement vous voudra accéder aux données via l'accès aux champs, mais dans ce cas, vous définirez un getter / paire de méthodes setter pour effectuer la transformation et utiliser l'accès aux propriétés pour celle-ci attribut. En général, il existe trois étapes essentielles pour ajouter un champ ou une propriété persistante à être accessible différemment du mode d'accès par défaut pour cette entité.

Considérez une entité Employee qui a un mode d'accès par défaut FIELD, mais le La colonne de base de données stocke l'indicatif régional dans le cadre du numéro de téléphone, et nous voulons seulement stockez l'indicatif régional dans le champ de l'entité phoneNum s'il ne s'agit pas d'un numéro local. Nous pouvons ajouter un propriété persistante qui le transforme en conséquence lors des lectures et des écritures.

La première chose à faire est de marquer explicitement le mode d'accès par défaut pour le classe en l'annotant avec l'annotation @Access et en indiquant le type d'accès. Sauf si ceci est fait, il ne sera pas défini si les champs et les propriétés sont annotés. Nous serions étiqueter notre entité Employé comme ayant un accès FIELD:

```
@Entité
@Access (AccessType.FIELD)
Employé de classe publique {...}
```

L'étape suivante consiste à annoter le champ ou la propriété supplémentaire avec @Access annotation, mais cette fois en spécifiant le type d'accès opposé à celui spécifié à le niveau de la classe. Il peut sembler un peu redondant, par exemple, de spécifier le type d'accès de AccessType.PROPERTY sur une propriété persistante car il est évident en le regardant que c'est une propriété, mais cela indique que ce que vous faites n'est pas un oubli, mais un exception consciente au cas par défaut.

```
@Access (AccessType.PROPERTY) @Column (name = "PHONE")
protected String getPhoneNumberForDb () {...}
```


Épisode 126

Le dernier point à retenir est que le champ ou la propriété correspondant à celui être rendu persistant doit être marqué comme transitoire afin que les règles d'accès par défaut ne provoque pas la persistance du même état deux fois. Par exemple, parce que nous ajoutons une propriété persistante à une entité pour laquelle le type d'accès par défaut est via des champs, le champ dans lequel l'état de la propriété persistante est stocké dans l'entité doit être annoté avec @Transient:

```
@Transient private String phoneNum;
```

Référencement 4-3 montre la classe d'entité Employee complète annotée pour utiliser la propriété accès pour une seule propriété.

Liste 4-3. Utilisation de l'accès combiné

```
@Entité
@Access (AccessType.FIELD)
Employé de classe publique {

    public static final String LOCAL_AREA_CODE = "613";

    @Id id long privé;
    @Transient private String phoneNum;
    ...
    public long getId () {return id; }
    public void setId (long id) {this.id = id; }

    public String getPhoneNumber () {return phoneNum; }
    public void setPhoneNumber (String num) {this.phoneNum = num; }

    @Access (AccessType.PROPERTY) @Column (name = "PHONE")
    protected String getPhoneNumberForDb () {
        if (phoneNum.length () == 10)
            return phoneNum;
        autre
            return LOCAL_AREA_CODE + phoneNum;
    }
    protected void setPhoneNumberForDb (String num) {
        si (num.startsWith (LOCAL_AREA_CODE))
            phoneNum = num.substring (3);
```


Épisode 127


```
        autre
            phoneNum = num;
    }
    ...
}
```

Mappage à une table

Vous avez vu au chapitre 2 que dans le cas le plus simple, mapper une entité à une table, ne besoin d'annotations de mappage. Seules les annotations @Entity et @Id doivent être spécifié pour créer et mapper une entité à une table de base de données.

Dans ces cas, le nom de la table par défaut, qui était simplement le nom non qualifié du la classe d'entité, était parfaitement adaptée. S'il arrive que le nom de la table par défaut ne soit pas le nom que vous aimez, ou si une table appropriée contenant l'état existe déjà dans votre

base de données avec un nom différent, vous devez spécifier le nom de la table. Vous faites cela en
annoter la classe d'entité avec l'annotation @Table et inclure le nom de la table
en utilisant l'élément name. De nombreuses bases de données ont des noms concis pour les tables. Référencement4-4 montre un
entité mappée à une table dont le nom est différent de son nom de classe.

Liste 4-4. Remplacement du nom de table par défaut

```
@Entité
@Table (nom = "EMP")
Employé de classe publique {...}
```

Conseil Les noms par défaut ne sont ni majuscules ni minuscules. Plus
les bases de données ne sont pas sensibles à la casse, de sorte que le fait qu'un fournisseur
utilise la casse du nom d'entité ou le convertit en majuscules. au chapitre 10 ,
nous expliquons comment délimiter les identifiants de base de données lorsque la base de données est définie pour être
sensible aux majuscules et minuscules.

Épisode 128

L'annotation @Table offre la possibilité non seulement de nommer la table que l'entité
state est stocké dans mais aussi pour nommer un schéma de base de données ou un catalogue. Le nom du schéma
est couramment utilisé pour différencier un ensemble de tableaux d'un autre et est indiqué par
en utilisant l'élément de schéma. Le Listing 4-5 montre une entité Employé qui est mappée au PGE
table dans le schéma HR.

Liste 4-5. Définition d'un schéma

```
@Entité
@Table (nom = "EMP", schéma = "HR")
Employé de classe publique {...}
```

Lorsqu'il est spécifié, le nom du schéma sera ajouté au nom de la table lorsque le
le fournisseur de persistance accède à la base de données pour accéder à la table. Dans ce cas, le schéma HR
sera ajouté au début de la table EMP à chaque accès à la table.

Conseil Certains fournisseurs peuvent autoriser l'inclusion du schéma dans le nom
élément de la table sans avoir à spécifier l'élément de schéma, comme dans
@Table (nom = "HR.EMP"). Prise en charge de l'inclusion du nom du schéma avec le
le nom de la table n'est pas standard.

Certaines bases de données supportent la notion de catalogue. Pour ces bases de données, le catalogue
l'élément de l'annotation @Table peut être spécifié. Le Listing 4-6 montre qu'un catalogue est
défini explicitement pour la table EMP.

Liste 4-6. Définition d'un catalogue

```
@Entité
@Table (nom = "EMP", catalogue = "HR")
Employé de classe publique {...}
```

Épisode 129

Mappage de types simples

Les types Java simples sont mappés dans le cadre de l'état immédiat d'une entité dans ses champs ou Propriétés. La liste des types persistants est assez longue et comprend à peu près tous type intégré que vous souhaitez conserver. Ils comprennent les éléments suivants:

- *Types Java primitifs* : byte, int, short, long, boolean, char, float, et double
- *Classes wrapper de types Java primitifs* : Byte, Integer, Short, Long, Booléen, caractère, flottant et double
- *Types de tableaux d'octets et de caractères* : byte [], Byte [], char [] et Personnage[]
- *Grands types numériques* : java.math.BigInteger et java.math. BigDecimal
- *Chaînes* : java.lang.String
- *Types temporels Java* : java.util.Date et java.util.Calendar
- *Types temporels JDBC* : java.sql.Date, java.sql.Time et java.sql. Horodatage
- *Types* énumérés: tout type énuméré système ou défini par l'utilisateur
- *Objets sérialisables* : tout système ou type sérialisable défini par l'utilisateur

Parfois, le type de la colonne de base de données mappée n'est pas exactement le même comme type Java. Dans presque tous les cas, le moteur d'exécution du fournisseur peut convertir le type renvoyé par JDBC dans le type Java correct de l'attribut. Si le type de la couche JDBC ne peut pas être converti au type Java du champ ou de la propriété, une exception sera normalement jeté, bien que ce ne soit pas garanti.

Conseil Lorsque le type persistant ne correspond pas au type jDbC, certains fournisseurs peut choisir d'entreprendre une action propriétaire ou de faire une meilleure estimation pour convertir entre les deux. dans d'autres cas, le pilote jDbC peut effectuer la conversion sur sa propre.

Épisode 130

Lors de la conservation d'un champ ou d'une propriété, le fournisseur examine le type et s'assure que c'est l'un des types persistants répertoriés précédemment. S'il est sur la liste, le fournisseur persistera

utilisez le type JDBC approprié et transmettez-le au pilote JDBC. À ce moment,
si le champ ou la propriété n'est pas sérialisable, le résultat n'est pas spécifié. Le fournisseur pourrait
choisissez de lever une exception ou essayez simplement de transmettre l'objet à JDBC. Tu verras
dans le chapitre 10 comment les convertisseurs peuvent être utilisés pour étendre la liste des types qui peuvent être persistants
dans JPA.

Une annotation @Basic facultative peut être placée sur un champ ou une propriété pour marquer explicitement
il est persistant. Cette annotation est principalement à des fins de documentation et n'est pas
requis pour que le champ ou la propriété soit persistant. Si ce n'est pas là, c'est implicitement
supposé en l'absence de toute autre annotation de mappage. En raison de l'annotation,
les mappages de types simples sont appelés mappages de base, que l'annotation @Basic soit
réellement présent ou est simplement supposé.

Notez maintenant que vous avez vu comment vous pouvez conserver les champs ou les propriétés et comment ils sont pratiquement équivalents en termes de persistance, nous les appellerons simplement les attributs. un attribut est un champ ou une propriété d'une classe, et nous utiliserons le terme attribut à partir de maintenant pour éviter d'avoir à se référer continuellement à des champs ou propriétés dans termes spécifiques.

Mappages de colonnes

L'annotation @Basic (ou le mappage de base supposé en son absence) peut être considérée comme
une indication logique qu'un attribut donné est persistant. L'annotation physique qui est
l'annotation associée au mappage de base est l'annotation @Column. En précisant
@Column sur l'attribut indique les caractéristiques spécifiques de la base de données physique
colonne dont le modèle objet se préoccupe moins. En fait, le modèle objet pourrait
jamais même besoin de savoir à quelle colonne il est mappé, et le nom de la colonne et
les métadonnées de mappage physique peuvent être situées dans un fichier XML distinct.

Un certain nombre d'éléments d'annotation peuvent être spécifiés dans le cadre de @Column, mais la plupart
ils s'appliquent uniquement à la génération de schéma et sont traités plus loin dans le livre. Le seul
qui a pour conséquence est l'élément name, qui est juste une chaîne qui spécifie le nom
de la colonne à laquelle l'attribut a été mappé. Ceci est utilisé lorsque la valeur par défaut
Le nom de la colonne n'est pas approprié ou ne s'applique pas au schéma utilisé. Vous pouvez

Épisode 131

considérez l'élément name de l'annotation @Column comme un moyen de remplacer la valeur par défaut
nom de colonne qui aurait autrement été appliqué. L'exemple du Listing 4-7
montre comment remplacer le nom de colonne par défaut d'un attribut.

Liste 4-7. Mappage d'attributs aux colonnes

```
@Entité
Employé de classe publique {
    @Id
    @Column (nom = "EMP_ID")
    identifiant long privé;
    nom de chaîne privé;
    @Column (nom = "SAL")
    long salaire privé;
    @Column (nom = "COMM")
    commentaires de chaîne privés;
    // ...
}
```

Pour mettre ces annotations en contexte, examinons le mappage complet de la table représenté par cette entité. La première chose à noter est qu'aucune annotation @Table n'existe sur la classe, donc le nom de table par défaut EMPLOYEE lui sera appliqué.

Ensuite, notez que @Column peut être utilisé avec les mappages @Id ainsi qu'avec les mappages. Le champ id est remplacé pour mapper à la colonne EMP_ID au lieu de la colonne ID par défaut. Le champ de nom n'est pas annoté avec @Column, donc la valeur par défaut nom de la colonne NAME serait utilisé pour stocker et récupérer le nom de l'employé. Le salaire et les champs de commentaires, cependant, sont annotés pour être mappés aux colonnes SAL et COMM, respectivement. L'entité Employé est donc mappée au tableau illustré à la Figure 4-2.

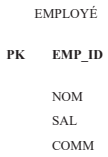

Figure 4-2. *Table d'entités EMPLOYEE*

Épisode 132

Récupération paresseuse

À l'occasion, on saura à l'avance que certaines parties d'une entité être rarement consulté. Dans ces situations, vous pouvez optimiser les performances lorsque récupérer l'entité en récupérant uniquement les données auxquelles vous vous attendez fréquemment accédées; le reste des données ne peut être récupéré que lorsque cela est nécessaire. Il y a beaucoup de noms pour ce type de fonctionnalité, y compris le chargement différé, le chargement différé, la récupération différée, récupération à la demande, lecture juste à temps, indirection et autres. Ils veulent tous dire jolis à peu près la même chose, c'est-à-dire que certaines données peuvent ne pas être chargées lorsque l'objet est initialement lu à partir de la base de données, mais ne sera extrait que lorsqu'il est référencé ou accédé.

Le type de récupération d'un mappage de base peut être configuré pour être chargé paresseusement ou avec empressement par en spécifiant l'élément fetch dans l'annotation @Basic correspondante. Le FetchType Le type énuméré définit les valeurs de cet élément, qui peut être EAGER ou LAZY. La définition du type de récupération d'un mappage de base sur LAZY signifie que le fournisseur peut différer chargement de l'état de cet attribut jusqu'à ce qu'il soit référencé. La valeur par défaut est de charger tous les mappages avec impatience. Le Listing 4-8 montre un exemple de remplacement d'un mappage de base pour être paresseusement chargé.

Liste 4-8. Chargement de champ paresseux

```
@Entité
Employé de classe publique {
    // ...
    @Basic (fetch = FetchType.LAZY)
    @Column (nom = "COMM")
    commentaires de chaîne privés;
    // ...
}
```

Nous supposons dans cet exemple que les applications accéderont rarement aux commentaires dans un enregistrement d'employé, nous le marquons donc comme étant paresseusement récupéré. Notez que dans ce cas l'annotation @Basic n'est pas seulement présente à des fins de documentation, mais aussi requis afin de spécifier le type d'extraction pour le champ. Configurer le champ de commentaires à récupérer paresseusement permettra à une instance Employee renvoyée par une requête d'avoir le champ de commentaires vide. L'application n'a rien à faire de spécial pour l'obtenir,

pourtant. En accédant simplement au champ de commentaires, il sera lu et rempli de manière transparente par le fournisseur s'il n'était pas déjà chargé.

Épisode 133

Avant d'utiliser cette fonctionnalité, vous devez être conscient de quelques points pertinents sur récupération d'attribut paresseux. Tout d'abord, la directive pour récupérer paresseusement un attribut est destiné uniquement à être un indice pour le fournisseur de persistance pour aider l'application à mieux fonctionner performance. Le fournisseur n'est pas tenu de respecter la demande car le comportement de l'entité n'est pas compromise si le fournisseur continue et charge l'attribut. le l'inverse n'est pas vrai, car en spécifiant qu'un attribut doit être récupéré avec impatience peut être critique pour pouvoir accéder à l'état de l'entité une fois que l'entité est détachée de le contexte de persistance. Nous discutons davantage du détachement au chapitre 6 et explorons les connexion entre le chargement paresseux et le détachement.

Deuxièmement, à première vue, il peut sembler que c'est une bonne idée pour certains attributs de une entité, mais en pratique, il n'est presque jamais une bonne idée de récupérer paresseusement des types simples. Là il y a peu à gagner à ne renvoyer qu'une partie d'une ligne de base de données, sauf si vous êtes certain que l'état ne sera plus accessible dans l'entité ultérieurement. Les seuls moments où le chargement paresseux de un mappage de base doit être considéré lorsqu'il y a plusieurs colonnes dans une table (pour par exemple, des dizaines ou des centaines) ou lorsque les colonnes sont grandes (par exemple, très grandes chaînes de caractères ou chaînes d'octets). Le chargement des données peut nécessiter des ressources importantes, et ne pas le charger pourrait économiser beaucoup d'efforts, de temps et de ressources. À moins que l'un ou l'autre de ces deux cas est vrai, dans la majorité des cas, la récupération paresseuse d'un sous-ensemble d'attributs d'objet finir par être plus cher que de les chercher avec impatience.

La récupération paresseuse est tout à fait pertinente lorsqu'il s'agit de mappages de relations, alors nous abordons ce sujet plus loin dans le chapitre.

Grands objets

Un terme de base de données courant pour un objet basé sur un caractère ou un octet qui peut être très volumineux (jusqu'à la plage du gigaoctet) est un gros objet, ou LOB en abrégé. Colonnes de base de données qui peuvent stocker ces types d'objets volumineux nécessitent des appels JDBC spéciaux pour être accessibles à partir de Java. À signaler au fournisseur qu'il doit utiliser les méthodes LOB lors du passage et de la récupération ces données vers et depuis le pilote JDBC, une annotation supplémentaire doit être ajoutée au cartographie de base. L'annotation @Lob fait office d'annotation de marqueur pour remplir cet objectif et peut apparaître en conjonction avec l'annotation @Basic, ou il peut apparaître lorsque @Basic est absent et implicitement supposé être sur le mappage.

Étant donné que l'annotation @Lob ne fait que qualifier le mappage de base, elle peut également être accompagné d'une annotation @Column lorsque le nom de la colonne LOB doit être remplacé par le nom par défaut supposé.

Épisode 134

Les LOB sont disponibles en deux versions dans la base de données: les grands objets de caractère, appelés CLOB, et les grands objets binaires, ou BLOB. Comme leur nom l'indique, une colonne CLOB contient un grand séquence de caractères et une colonne BLOB peut stocker une grande séquence d'octets. Le Java les types mappés aux colonnes BLOB sont les types byte [], Byte [] et Serializable, tandis que Les objets char [], Character [] et String sont mappés aux colonnes CLOB. Le fournisseur est

responsable de faire cette distinction en fonction du type de l'attribut mappé.

Un exemple de mappage d'une image à une colonne BLOB est présenté dans la liste 4-9 . Ici, la colonne PIC est supposée être une colonne BLOB pour stocker l'image de l'employé qui se trouve le champ d'image. Nous avons également marqué ce champ pour être chargé paresseusement, une pratique courante appliqué aux LOB qui ne sont pas souvent référencés.

Liste 4-9. Mappage d'une colonne BLOB

```
@Entité
Employé de classe publique {
    @Id
    identifiant long privé;
    @Basic (fetch = FetchType.LAZY)
    @Lob @Column (nom = "PIC")
    image privée d'octet [];
    // ...
}
```

Types énumérés

Un autre des types simples qui pourraient être traités spécialement est le type énuméré. le les valeurs d'un type énuméré sont des constantes qui peuvent être gérées différemment selon sur les besoins de l'application.

Comme pour les types énumérés dans d'autres langues, les valeurs d'un type énuméré dans Java a une affectation ordinale implicite qui est déterminée par l'ordre dans lequel ils ont été déclarés. Cet ordinal ne peut pas être modifié à l'exécution et peut être utilisé pour représenter et stocker les valeurs du type énuméré dans la base de données. Interpréter les valeurs comme ordinals est le moyen par défaut utilisé par les fournisseurs pour mapper les types énumérés à la base de données, et le fournisseur supposera que la colonne de base de données est de type entier.

Épisode 135

Considérez le type énuméré suivant:

```
public enum EmployeeType {
    EMPLOYÉ À PLEIN TEMPS,
    EMPLOYÉ À TEMPS PARTIEL,
    CONTRACT_EMPLOYEE
}
```

Les ordinaux affectés aux valeurs de ce type énuméré au moment de la compilation seraient être 0 pour FULL_TIME_EMPLOYEE, 1 pour PART_TIME_EMPLOYEE et 2 pour CONTRACT_EMPLOYEE. Dans la liste 4-10 , nous définissons un champ persistant de ce type.

Annonce 4-10. Mappage d'un type énuméré à l'aide d'ordinaux

```
@Entité
Employé de classe publique {
    @Id id long privé;
    type EmployeeType privé;
    // ...
}
```

Vous pouvez voir que le mappage EmployeeType est trivialement facile au point où vous n'ont rien à faire du tout. Les valeurs par défaut sont appliquées et tout sera travaille juste. Le champ de type sera mappé à une colonne TYPE d'entier, et tous à temps plein

les employés se verront attribuer un ordinal de 0. De même, les autres employés
ont leurs types stockés dans la colonne TYPE en conséquence.

Si un type énuméré change, cependant, nous avons un problème. Le persisté
les données ordinales de la base de données ne s'appliqueront plus à la valeur correcte. Dans cet exemple,
si la politique des avantages sociaux de l'entreprise a changé et que nous avons commencé à offrir des avantages supplémentaires à
employés à temps partiel qui travaillaient plus de 20 heures par semaine, nous voudrions
faire la distinction entre les deux types d'employés à temps partiel. En ajoutant un PART_TIME_
Valeur BENEFITS_EMPLOYEE après PART_TIME_EMPLOYEE, nous provoquerions un nouvel ordinal
affectation à se produire, où notre nouvelle valeur recevrait l'ordinal de 2 et
CONTRACT_EMPLOYEE obtiendrait 3. Cela aurait pour effet de provoquer tout le contrat
employés connus pour devenir soudainement des employés à temps partiel avec des avantages, manifestement pas
le résultat que nous espérions.

Épisode 136

Nous pourrions parcourir la base de données et ajuster toutes les entités Employés pour avoir leur
type correct, mais si le type d'employé est utilisé ailleurs, nous aurions besoin de
sûr qu'ils étaient tous réparés aussi. Ce n'est pas une bonne situation de maintenance.

Une meilleure solution serait de stocker le nom de la valeur sous forme de chaîne au lieu de stocker
l'ordinal. Cela nous isolerait de tout changement de déclaration et nous permettrait d'ajouter
de nouveaux types sans avoir à vous soucier des données existantes. Nous pouvons le faire en ajoutant un @
Annotation énumérée sur l'attribut et spécifiant une valeur de STRING.

L'annotation @Enumerated permet en fait de spécifier un EnumType, et le
EnumType est lui-même un type énuméré qui définit les valeurs de ORDINAL et STRING. Tandis que
il est quelque peu ironique qu'un type énuméré soit utilisé pour indiquer comment le fournisseur
devrait représenter des types énumérés, il est tout à fait approprié. Parce que la valeur par défaut de
@Enumerated est ORDINAL, spécifier @Enumerated (ORDINAL) n'est utile que lorsque vous le souhaitez
pour rendre ce mappage explicite.

Dans la liste 4-11 , nous stockons des chaînes pour les valeurs énumérées. Maintenant le TYPE
La colonne doit être de type chaîne et tous les employés à temps plein auront
string FULL_TIME_EMPLOYEE stocké dans leur colonne TYPE correspondante.

Annonce 4-11. Mappage d'un type énuméré à l'aide de chaînes

```
@Entité
Employé de classe publique {
    @Id
    identifiant long privé;
    @Enumerated (EnumType.STRING)
    type EmployeeType privé;
    // ...
}
```

Notez que l'utilisation de chaînes résoudra le problème de l'insertion de valeurs supplémentaires dans le
milieu du type énuméré, mais cela laissera les données vulnérables aux changements dans le
noms des valeurs. Par exemple, si nous voulions remplacer PART_TIME_EMPLOYEE par
PT_EMPLOYEE, alors nous serions en difficulté. C'est un problème moins probable, car
modifier les noms d'un type énuméré entraînerait tout le code qui utilise le
le type énuméré doit également changer. Ce serait un plus gros problème que de réaffecter
valeurs dans une colonne de base de données.

Épisode 137

En général, le stockage de l'ordinal est le moyen le meilleur et le plus efficace de stocker les types tant que la probabilité de valeurs supplémentaires insérées au milieu n'est pas haute. De nouvelles valeurs pourraient encore être ajoutées à la fin du type sans aucun négatif conséquences.

Une dernière remarque sur les types énumérés est qu'ils sont définis de manière assez flexible en Java. En fait, il est même possible d'avoir des valeurs contenant state. Il n'y a actuellement aucun support dans le JPA pour l'état de mappage contenu dans les valeurs énumérées. Il n'y a pas non plus prise en charge de la position de compromis entre STRING et ORDINAL du mappage explicite chaque valeur énumérée à une valeur numérique dédiée différente de son compilateur- valeur ordinale assignée. Un soutien énuméré plus étendu est envisagé pour versions futures.

Types temporels

Les types temporels sont l'ensemble des types basés sur le temps qui peuvent être utilisés dans un état persistant mappages. La liste des types temporels pris en charge comprend les trois types java.sql: java.sql.Date, java.sql.Time et java.sql.Timestamp - et les deux java.util types—java.util.Date et java.util.Calendar.

Les types java.sql sont totalement simples. Ils agissent comme n'importe quel autre simple type de mappage et ne nécessitent aucune considération particulière. Les deux types java.util besoin de métadonnées supplémentaires, cependant, pour indiquer lequel des types JDBC java.sql à à utiliser lors de la communication avec le pilote JDBC. Cela se fait en les annotant avec l'annotation @Temporal et en spécifiant le type JDBC comme valeur du TemporalType type énuméré. Il existe trois valeurs énumérées de DATE, TIME et TIMESTAMP à représentent chacun des types java.sql.

Référencement 4-12 montre comment java.util.Date et java.util.Calendar peuvent être mappés vers colonnes de date dans la base de données.

Annonce 4-12. Mappage des types temporels

```
@Entité
Employé de classe publique {
    @Id
    identifiant long privé;
    @Temporal (TemporalType.DATE)
    calendrier privé dob;
```


Épisode 138


```
    @Temporal (TemporalType.DATE)
    @Column (nom = "S_DATE")
    private Date startDate;
    // ...
}
```

Comme les autres variétés de mappages de base, l'annotation @Column peut être utilisée pour remplacer le nom de colonne par défaut.

État transitoire

Les attributs qui font partie d'une entité persistante mais qui ne sont pas destinés à être persistants peuvent non plus être modifié avec le modificateur transitoire en Java ou être annoté avec le @Transient annotation. Si l'un ou l'autre est spécifié, le moteur d'exécution du fournisseur n'appliquera pas son mappage par défaut règles à l'attribut sur lequel il a été spécifié.

Les champs transitoires sont utilisés pour diverses raisons. On pourrait être le cas plus tôt dans la chapitre lorsque nous avons mélangé le mode d'accès et que nous ne voulions pas conserver le même état deux fois. Un autre peut être lorsque vous souhaitez mettre en cache un état en mémoire que vous ne souhaitez pas doivent recalculer, redécouvrir ou réinitialiser. Par exemple, dans Listing4-13 nous utilisons un champ transitoire pour enregistrer le mot correct spécifique à la locale pour l'employé afin que nous l'imprimions correctement partout où il est affiché. Nous avons utilisé le modificateur transitoire à la place de l'annotation @Transient de sorte que si l'employé est sérialisé d'une VM à un autre, puis le nom traduit sera réinitialisé pour correspondre à la locale du nouvelle VM. Dans les cas où la valeur non persistante doit être conservée pendant la sérialisation, l'annotation doit être utilisée à la place du modificateur.

Annonce 4-13. Utilisation d'un champ transitoire

```
@Entité
Employé de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    long salaire privé;
    transient private String translateName;
    // ...
```


Épisode 139


```
    public String toString () {
        if (TranslatedName == null) {
            translateName =
                ResourceBundle.getBundle ("EmpResources").
                getString ("Employé");
        }
        renvoie nom traduit + ":" + id + "" + nom;
    }
}
```

Mappage de la clé primaire

Chaque entité mappée à une base de données relationnelle doit avoir un mappage vers un clé dans le tableau. Vous avez déjà appris les bases de la façon dont l'annotation @Id indique l'identifiant de l'entité. Dans cette section, vous explorez les identifiants simples et les clés un peu plus en profondeur et découvrez comment vous pouvez laisser le fournisseur de persistance générer valeurs d'identifiant uniques.

> Remarque Lorsqu'un identifiant d'entité est composé d'un seul attribut, il est appelé identifiant simple.

Remplacement de la colonne de clé primaire

Les mêmes règles par défaut s'appliquent aux mappages d'ID et aux mappages de base, c'est-à-dire que le

Le nom de la colonne est supposé être le même que le nom de l'attribut. Tout comme avec mappages de base, l'annotation @Column peut être utilisée pour remplacer le nom de colonne qui l'attribut ID est mappé sur.

Les clés primaires sont supposées être insérables, mais non nulles ou actualisables. Quand écrasant une colonne de clé primaire, les éléments Nullable et Modifiable ne doivent pas être remplacé. Uniquement dans le cas très spécifique du mappage de la même colonne à plusieurs champs / relations (comme décrit au chapitre 10) si l'élément insérable element soit défini sur false.

Épisode 140

Types de clé primaire

À l'exception de sa signification particulière dans la désignation du mappage vers la colonne de clé primaire, un mappage d'ID est presque le même que le mappage de base. L'autre différence principale est que les mappages d'ID sont généralement limités aux types suivants:

- *Types Java primitifs* : byte, int, short, long et char
- *Classes wrapper de types Java primitifs:* Byte, Integer, Short, Long, et caractère
- *Chaîne* : java.lang.String
- *Grand type numérique* : java.math.BigInteger
- *Types temporels* : java.util.Date et java.sql.Date

Les types à virgule flottante tels que float et double sont également autorisés, ainsi que le Classes de wrapper Float et Double et java.math.BigDecimal, mais ils sont déconseillés en raison de la nature de l'erreur d'arrondi et du manque de fiabilité des égaux () opérateur lorsqu'il leur est appliqué. L'utilisation de types flottants pour les clés primaires est une entreprise risquée et n'est certainement pas recommandé.

Génération d'identifiant

Parfois, les applications ne veulent pas se soucier d'essayer de définir et de garantir unicité dans certains aspects de leur modèle de domaine et se contentent de laisser l'identifiant les valeurs seront automatiquement générées pour eux. Ceci s'appelle la génération d'ID et est spécifié par l'annotation @GeneratedValue.

Lorsque la génération d'ID est activée, le fournisseur de persistance génère un identifiant valeur pour chaque instance de ce type d'entité. Une fois la valeur de l'identifiant obtenue, le le fournisseur l'insérera dans l'entité nouvellement persistante; cependant, selon la façon dont il est généré, il peut ne pas être présent dans l'objet tant que l'entité n'a pas été insérée dans la base de données. En d'autres termes, l'application ne peut pas compter sur la possibilité d'accéder au identifiant jusqu'à ce qu'un vidage ait eu lieu ou que la transaction soit terminée.

Les applications peuvent choisir l'une des quatre stratégies de génération d'identifiants différentes en spécifiant un stratégie dans l'élément de stratégie. La valeur peut être l'une des valeurs AUTO, TABLE, SEQUENCE ou IDENTITY valeurs énumérées du type énuméré GenerationType.

Épisode 141

Les générateurs de table et de séquence peuvent être spécifiquement définis puis réutilisés par plusieurs classes d'entités. Ces générateurs sont nommés et sont globalement accessibles à tous les entités dans l'unité de persistance.

Génération automatique d'ID

Si une application ne se soucie pas du type de génération utilisé par le fournisseur mais veut génération à se produire, il peut spécifier une stratégie de AUTO. Cela signifie que le fournisseur utilisez la stratégie de votre choix pour générer des identifiants. Le Listing 4-14 montre un exemple de en utilisant la génération automatique d'ID. Cela entraînera la création d'une valeur d'identifiant par le provider et inséré dans le champ id de chaque entité Employee qui est persistante.

> Astuce, il n'est pas explicitement requis que le champ d'identifiant d'entité soit de type intégral, mais c'est généralement le seul type créé par AUTO. Nous recommandons que longtemps utilisé pour accueillir toute l'étendue du domaine d'identifiant généré.

Liste 4-14. Utilisation de la génération automatique d'ID

```
@Entité
Employé de classe publique {
    @Id @GeneratedValue (stratégie = GenerationType.AUTO)
    identifiant long privé;
    // ...
}
```

Il y a cependant un problème à utiliser AUTO. Le fournisseur choisit sa propre stratégie pour stocker les identifiants, mais il doit avoir une sorte de ressource persistante pour faire cela. Par exemple, s'il choisit une stratégie basée sur une table, il doit créer une table; s'il choisit une stratégie basée sur une séquence, il doit créer une séquence. Le fournisseur ne peut pas toujours compter sur la connexion à la base de données qu'il obtient du serveur pour avoir autorisations pour créer une table dans la base de données. Il s'agit normalement d'une opération privilégiée qui est souvent limité au DBA. Il devra y avoir une sorte de phase de création ou la génération de schéma pour provoquer la création de la ressource avant que la stratégie AUTO ne puisse une fonction.

Épisode 142

Le mode AUTO est vraiment une stratégie de génération pour le développement ou le prototypage. Il fonctionne bien comme un moyen de vous mettre en marche plus rapidement lorsque la base de données le schéma est en cours de génération. Dans toute autre situation, il serait préférable d'utiliser l'un des autres stratégies de génération discutées dans les sections suivantes.

Génération d'ID à l'aide d'une table

La manière la plus flexible et la plus portable de générer des identifiants consiste à utiliser une table de base de données. Non seulement il portera vers différentes bases de données, mais il permet également de stocker plusieurs séquences d'identificateurs pour différentes entités dans la même table.

Une table de génération d'ID doit avoir deux colonnes. La première colonne est un type chaîne utilisé pour identifier la séquence particulière du générateur. C'est la clé primaire de tous les générateurs dans le tableau. La deuxième colonne est un type intégral qui stocke l'ID réel

séquence en cours de génération. La valeur stockée dans cette colonne est le dernier identifiant qui a été alloué dans la séquence. Chaque générateur défini représente une ligne dans le tableau.

La manière la plus simple d'utiliser une table pour générer des identifiants est de simplement spécifier le stratégie de génération pour être TABLE dans l'élément de stratégie:

```
@Id @GeneratedValue (stratégie = GenerationType.TABLE)
identifiant long privé;
```

Parce que la stratégie de génération est indiquée mais qu'aucun générateur n'a été spécifié, le Le fournisseur assumera une table de son choix. Si la génération de schéma est utilisée, elle sera créé; sinon, la table par défaut assumée par le fournisseur doit être connue et doit exister dans la base de données.

Une approche plus explicite consisterait à spécifier en fait la table à utiliser pour le stockage d'identité. Cela se fait en définissant un générateur de table qui, contrairement à son nom implique, ne génère pas réellement de tables. Il s'agit plutôt d'un générateur d'identifiant qui utilise une table pour stocker les valeurs d'identifiant. Nous pouvons en définir un en utilisant un @TableGenerator annotation, puis faites-y référence par son nom dans l'annotation @GeneratedValue:

```
@TableGenerator (nom = "Emp_Gen")
@Id @GeneratedValue (générateur = "Emp_Gen")
identifiant long privé;
```

Bien que nous montrions le @TableGenerator annotant l'attribut identificateur, il peut en fait être défini sur n'importe quel attribut ou classe. Peu importe où il est défini, il sera être disponible pour l'ensemble de l'unité de persistance. Une bonne pratique serait de le définir localement

---

Épisode 143

sur l'attribut ID si une seule classe l'utilise mais pour le définir en XML, comme décrit dans Chapitre 13, s'il sera utilisé pour plusieurs classes.

L'élément name nomme globalement le générateur, ce qui nous permet de le référencer dans le Élément générateur de l'annotation @GeneratedValue. C'est fonctionnellement équivalent à l'exemple précédent où nous avons simplement dit que nous voulions utiliser la génération de table mais n'a pas spécifié le générateur. Nous spécifions maintenant le nom du générateur mais ne fournissant aucun des détails du générateur, les laissant par défaut par le fournisseur.

Une autre approche de qualification consisterait à spécifier les détails du tableau, comme dans le Suivant:

```
@TableGenerator (nom = "Emp_Gen",
    table = "ID_GEN",
    pkColumnName = "GEN_NAME",
    valueColumnName = "GEN_VAL")
```

Nous avons inclus quelques éléments supplémentaires après le nom du générateur. Après le nom se trouvent trois éléments: table, pkColumnName et valueColumnName - qui définissent la table réelle qui stocke les identificateurs pour Emp_Gen.

L'élément table indique simplement le nom de la table. L'élément pkColumnName est le nom de la colonne de clé primaire de la table qui identifie de manière unique le générateur, et l'élément valueColumnName est le nom de la colonne qui stocke l'ID réel valeur de séquence en cours de génération. Dans ce cas, la table est nommée ID_GEN, le nom du la colonne de clé primaire (la colonne qui stocke les noms des générateurs) est nommée GEN_NAME, et la colonne qui stocke les valeurs de séquence d'ID est nommée GEN_VAL.

Le nom du générateur devient la valeur stockée dans la colonne pkColumnName pour cette ligne et est utilisé par le fournisseur pour rechercher le générateur pour obtenir son dernier alloué valeur.

Dans notre exemple, nous avons nommé notre générateur Emp_Gen afin que notre table ressemble au un dans la figure 4-3 .

ID_GEN

GEN_NAME	GEN_VAL
Emp_Gen	0

Figure 4-3. *Tableau de génération d'identifiant*

Épisode 144

Notez que le dernier identifiant d'employé alloué est 0, ce qui nous indique qu'aucun identifiant ont encore été générés. Un élément initialValue représentant le dernier alloué identificateur peut être spécifié dans le cadre de la définition du générateur, mais le paramètre par défaut de 0 suffira dans presque tous les cas. Ce paramètre est utilisé uniquement lors de la génération de schéma lorsque la table est créée. Lors des exécutions suivantes, le fournisseur lira le contenu de la colonne de valeur pour déterminer le prochain identifiant à donner.

Pour éviter de mettre à jour la ligne pour chaque identifiant unique demandé, un la taille d'allocation est utilisée. Cela obligera le fournisseur à préallouer un bloc d'identifiants puis donner des identifiants de la mémoire comme demandé jusqu'à ce que le bloc soit épuisé. Une fois ce bloc épuisé, la prochaine demande d'identifiant déclenche un autre bloc de identifiants à préallouer, et la valeur d'identifiant est incrémentée par l'allocation Taille. Par défaut, la taille d'allocation est définie sur 50. Cette valeur peut être remplacée pour être plus grande ou plus petit grâce à l'utilisation de l'élément allocationSize lors de la définition du générateur.

> Astuce, le fournisseur peut allouer des identifiants dans la même transaction que le entité persistante ou dans une transaction distincte. ce n'est pas spécifié, mais vous devrait vérifier la documentation de votre fournisseur pour voir comment il peut éviter le risque de blocage lorsque des threads simultanés créent des entités et verrouillent des ressources.

Référencement 4-15 montre un exemple de définition d'un deuxième générateur à utiliser pour l'adresse entités mais qui utilise la même table ID_GEN pour stocker la séquence d'identificateurs. Dans ce cas, nous dictons en fait explicitement la valeur que nous stockons dans la table des identificateurs colonne de clé primaire en spécifiant l'élément pkColumnvalue. Cet élément permet le nom du générateur soit différent de la valeur de la colonne, bien que cela soit rarement nécessaire. L'exemple montre un générateur d'ID d'adresse nommé Address_Gen mais définit ensuite la valeur stockée dans la table pour la génération d'ID d'adresse comme Addr_Gen. le Le générateur définit également la valeur initiale sur 10000 et la taille d'allocation sur 100.

Liste 4-15. Utilisation de la génération d'ID de table

```
@TableGenerator (nom = "Address_Gen",
        table = "ID_GEN",
        pkColumnName = "GEN_NAME",
        valueColumnName = "GEN_VAL",
        pkColumnValue = "Addr_Gen",
```


Épisode 145

initialValue = 10000,
allocationSize = 100)
@Id @GeneratedValue (générateur = "Address_Gen")
identifiant long privé;

Si les deux générateurs Emp_Gen et Address_Gen ont été définis, alors sur demande au démarrage, la table ID_GEN doit ressembler à la figure 4-4 . Au fur et à mesure que l'application alloue identificateurs, les valeurs stockées dans la colonne GEN_VAL augmenteront.

ID_GEN

GEN_NAME	GEN_VAL
Emp_Gen	0
Addr_Gen	10 000

Figure 4-4. *Tableau de génération des identifiants d'adresse et d'employé*

Si vous n'avez pas utilisé la fonction de génération automatique de schéma (décrite dans Chapitre 14), la table doit déjà exister ou être créée dans la base de données via d'autres moyens et être configuré pour être dans cet état lorsque l'application démarre pour le première fois. Le SQL suivant peut être appliqué pour créer et initialiser cette table:

```
CRÉER UNE TABLE id_gen (
    nom_gen VARCHAR (80),
    gen_val INTEGER,
    CONTRAINTE pk_id_gen
        CLÉ PRIMAIRE (nom_gén)
);
INSERT INTO id_gen (gen_name, gen_val) VALUES ('Emp_Gen', 0);
INSERT INTO id_gen (gen_name, gen_val) VALUES ('Addr_Gen', 10000);
```

Génération d'ID à l'aide d'une séquence de base de données

De nombreuses bases de données prennent en charge un mécanisme interne de génération d'ID appelé *séquences* . Une séquence de base de données peut être utilisée pour générer des identifiants lorsque la base de données sous-jacente les soutient.

Comme vous l'avez vu avec les générateurs de table, s'il est connu qu'une séquence de base de données doit être utilisé pour générer des identifiants, et vous ne craignez pas que ce soit séquence, la spécification du seul type de générateur devrait suffire:

```
@Id @GeneratedValue (stratégie = GenerationType.SEQUENCE)
identifiant long privé;
```

Dans ce cas, aucun générateur n'est nommé, donc le fournisseur utilisera une séquence par défaut objet de son choix. Notez que si plusieurs générateurs de séquence sont définis mais pas nommé, il n'est pas précisé s'ils utilisent la même séquence par défaut ou des séquences différentes. La seule différence entre l'utilisation d'une séquence pour plusieurs types d'entités et l'utilisation d'une seule pour chaque entité serait l'ordre des numéros de séquence et une éventuelle contention sur la séquence. La voie la plus sûre serait de définir un générateur de séquence nommé et faites-y référence dans l'annotation @GeneratedValue:

```
@SequenceGenerator (nom = "Emp_Gen", sequenceName = "Emp_Seq")
@Id @GeneratedValue (générateur = "Emp_Gen")
privé long getId;
```

À moins que la génération de schéma ne soit activée, il faudrait que la séquence soit définie et existent déjà. Le SQL pour créer une telle séquence serait le suivant:

```
CRÉER UNE SÉQUENCE Emp_Seq
    MINVALUE 1
    COMMENCER AVEC 1
    AUGMENTATION DE 50
```

La valeur initiale et la taille d'allocation peuvent également être utilisées dans les générateurs de séquence et devrait être reflété dans le SQL pour créer la séquence. Notez que la valeur par défaut la taille d'allocation est de 50, comme c'est le cas avec les générateurs de table. Si la génération de schéma n'est pas utilisé, et la séquence est créée manuellement, la clause INCREMENT BY aurait besoin à configurer pour correspondre à l'élément allocationSize ou à la taille d'allocation par défaut du l'annotation @SequenceGenerator correspondante.

Génération d'ID à l'aide de l'identité de base de données

Certaines bases de données prennent en charge une colonne d'identité de clé primaire, parfois appelée colonne de numérotation automatique. Chaque fois qu'une ligne est insérée dans la table, la colonne d'identité obtenir un identifiant unique qui lui est attribué. Il peut être utilisé pour générer les identifiants des objets,

Épisode 147

mais encore une fois n'est disponible que lorsque la base de données sous-jacente le prend en charge. L'identité est souvent utilisé lorsque les séquences de base de données ne sont pas prises en charge par la base de données ou parce qu'un Le schéma hérité a déjà défini la table pour utiliser les colonnes d'identité. Ils sont généralement moins efficace pour la génération d'identificateurs relationnels objet car ils ne peuvent pas être alloués en blocs et parce que l'identifiant n'est disponible qu'après l'heure de validation.

Pour indiquer que la génération d'IDENTITY doit avoir lieu, l'annotation @GeneratedValue devrait spécifier une stratégie de génération d'IDENTITY. Cela indiquera au fournisseur qu'il doit relire la ligne insérée à partir du tableau après une insertion. Cela lui permettra pour obtenir l'identifiant nouvellement généré à partir de la base de données et le mettre dans la mémoire entité qui vient de persister:

```
@Id @GeneratedValue (stratégie = GenerationType.IDENTITY)
identifiant long privé;
```

Il n'y a pas d'annotation de générateur pour IDENTITY car elle doit être définie dans le cadre de la définition du schéma de base de données pour la colonne de clé primaire de l'entité. Parce que chacun La colonne de clé primaire d'entité définit sa propre caractéristique d'identité, génération d'IDENTITY ne peut pas être partagé entre plusieurs types d'entités.

Une autre différence, évoquée plus tôt, entre l'utilisation d'IDENTITY et d'autres ID stratégies de génération est que l'identifiant ne sera accessible qu'après l'insertion s'est produit. Bien qu'aucune garantie ne soit donnée sur l'accessibilité de l'identifiant avant la fin de la transaction, il est au moins possible pour d'autres types de génération pour allouer avec empressement l'identifiant. Mais lors de l'utilisation de l'identité, c'est l'action d'insérer qui provoque la génération de l'identifiant. Il serait impossible que l'identifiant soit disponible avant l'insertion de l'entité dans la base de données et parce que l'insertion d'entités est le plus souvent différé jusqu'au moment de la validation, l'identifiant ne serait disponible qu'après la transaction a été validée.

Astuce si vous utilisez IDENTITY, assurez-vous que vous êtes conscient de ce que votre persistance le fournisseur fait et qu'il correspond à vos besoins. Certains fournisseurs avec impatience insérer (lorsque la méthode persist est appelée) des entités configurées pour utiliser IDENTITY iD generation, au lieu d'attendre l'heure de validation. cela permettra au iD soit disponible immédiatement, au détriment d'un verrouillage prématuré et réduit

concurrence. Certains fournisseurs ont même une option qui vous permet de configurer quelle approche est utilisée.

Épisode 148

Des relations

Si les entités ne contenaient qu'un état persistant simple, l'activité de relation objet la cartographie serait en effet triviale. La plupart des entités doivent pouvoir référencer, ou avoir des relations avec d'autres entités. C'est ce qui produit les graphes du modèle de domaine qui sont courantes dans les applications métier.

Dans les sections suivantes, nous explorons les différents types de relations qui peuvent exister et montrez comment les définir et les mapper à l'aide des métadonnées de mappage JPA.

Concepts relationnels

Avant de commencer à cartographier les relations, faisons un rapide tour d'horizon de certains des les concepts et la terminologie de base des relations. Avoir une bonne compréhension de ces concepts facilitera la compréhension du reste des sections de cartographie des relations.

Rôles

Il y a un vieil adage qui dit que chaque histoire a trois côtés: le vôtre, le mien et la vérité. Les relations sont un peu les mêmes en ce sens qu'il existe trois perspectives différentes. La première est la vue d'un côté de la relation, la seconde est de l'autre côté, et le le troisième est d'une perspective mondiale qui connaît les deux côtés. Les «côtés» sont appelés rôles. Dans chaque relation, il y a deux entités qui sont liées l'une à l'autre, et on dit que chaque entité joue un rôle dans la relation.

Les relations sont partout, donc les exemples ne sont pas difficiles à trouver. Un employé a une relation avec le service dans lequel il travaille. L'entité Employé joue le rôle de travailler dans le département, tandis que l'entité Département joue le rôle de avoir un employé qui y travaille.

Bien entendu, le rôle joué par une entité donnée diffère selon la relation, et une entité peut participer à de nombreuses relations différentes avec de nombreux entités. Nous pouvons donc conclure que toute entité peut jouer un certain nombre de différents rôles dans un modèle donné. Si nous pensons à une entité Employé, nous nous rendons compte qu'elle joue, en fait, d'autres rôles dans d'autres relations, comme le rôle de travailler pour un manager dans sa relation avec une autre entité Salariée, travaillant sur un projet dans son relation avec l'entité de projet, et ainsi de suite. Bien qu'il n'y ait pas de métadonnées exigences pour déclarer le rôle qu'une entité joue, les rôles sont néanmoins toujours utiles car un moyen de comprendre la nature et la structure des relations.

Épisode 149

Directionnalité

Pour avoir des relations, il doit y avoir un moyen de créer, supprimer et maintenir leur. La méthode de base consiste à utiliser une entité ayant un attribut de relation qui fait référence

à son entité apparentée d'une manière qui l'identifie comme jouant l'autre rôle de la relation. Il arrive souvent que l'autre entité, à son tour, ait un attribut qui pointe vers le entité d'origine. Lorsque chaque entité pointe vers l'autre, la relation est bidirectionnelle. Si une seule entité a un pointeur vers l'autre, la relation est dite unidirectionnelle.

Une relation entre un employé et le projet sur lequel il travaille serait bidirectionnel. L'employé doit connaître son projet et le projet doit pointer vers l'employé qui y travaille. Un modèle UML de cette relation est illustré à la figure 4-5 . le les flèches allant dans les deux sens indiquent la bidirectionnalité de la relation.

Figure 4-5. *Employé et projet dans une relation bidirectionnelle*

Un employé et son adresse seraient probablement modélisés comme un relation car l'adresse n'aura jamais besoin de connaître son résident. Si il l'a fait, bien sûr, alors il faudrait qu'il devienne une relation bidirectionnelle. Figure 4-6 montre cette relation. Comme la relation est unidirectionnelle, la flèche pointe de l'employé à l'adresse.

Figure 4-6. *Employé dans une relation unidirectionnelle avec adresse*

Comme vous le verrez plus loin dans le chapitre, bien qu'ils partagent tous deux le même concept de directionnalité, l'objet et les modèles de données le voient chacun un peu différemment en raison de la différence de paradigme. Dans certains cas, les relations unidirectionnelles dans le modèle objet peuvent posent un problème dans le modèle de base de données.

Nous pouvons utiliser la directionnalité d'une relation pour aider à décrire et expliquer un modèle, mais quand il s'agit d'en discuter concrètement, il est logique de penser de chaque relation bidirectionnelle comme une paire de relations unidirectionnelles. Au lieu de ayant une relation bidirectionnelle unique d'un employé travaillant sur un projet, nous aurait une relation de «projet» unidirectionnelle vers laquelle l'employé pointe le projet sur lequel ils travaillent et une autre relation de «travailleur» unidirectionnelle où le Le projet pointe vers l'employé qui y travaille. Chacune de ces relations a une entité

Épisode 150

c'est-à-dire le rôle source ou référent et le côté qui est le rôle cible ou référencé. le la beauté de ceci est que nous pouvons utiliser les mêmes termes quelle que soit la relation que nous entretenons parler et quels que soient les rôles dans la relation. Figure 4-7 montre comment les deux relations ont des entités source et cible, et comment de chaque relation perspective, les entités source et cible sont différentes.

Figure 4-7. *Relations unidirectionnelles entre l'employé et le projet*

Cardinalité

Il n'est pas très fréquent qu'un projet n'ait qu'un seul employé qui y travaille. Nous voudrions pour pouvoir capturer l'aspect du nombre d'entités existant de chaque côté du même instance de relation. C'est ce qu'on appelle la cardinalité de la relation. Chaque rôle dans un relation aura sa propre cardinalité, qui indique s'il ne peut y avoir qu'un seul instance de l'entité ou de nombreuses instances.

Dans notre exemple d'employé et de service, nous pourrions d'abord dire qu'un employé

travaille dans un seul département, donc la cardinalité des deux côtés serait une. Mais les chances est-ce que plus d'un employé travaille dans le département, nous ferions donc relation ont de nombreuses cardinalités du côté de l'employé ou de la source, ce qui signifie que Les instances d'employés peuvent chacune pointer vers le même service. La cible ou le département côté garderait sa cardinalité de un. Figure 4-8 montre cette relation plusieurs-à-un. Le côté «plusieurs» est marqué d'un astérisque (*).

Figure 4-8. *Relation unidirectionnelle plusieurs à un*

Dans notre exemple Employé et Projet, nous avons une relation bidirectionnelle, ou deux directions des relations. Si un employé peut travailler sur plusieurs projets et un projet peut avoir plusieurs employés qui y travaillent, alors nous nous retrouverions avec des cardinalités de «Nombreux» sur les sources et les cibles des deux sens. La Figure 4-9 montre le diagramme UML de cette relation.

Épisode 151

Figure 4-9. *Relation plusieurs-à-plusieurs bidirectionnelle*

Comme le dit le proverbe, une image vaut mille mots, et les décrivant les relations dans le texte sont beaucoup plus difficiles que de montrer une image. En mots, cependant, ce l'image indique ce qui suit:

- Chaque employé peut travailler sur plusieurs projets.
- De nombreux employés peuvent travailler sur le même projet.
- Chaque projet peut avoir un certain nombre d'employés qui y travaillent.
- De nombreux projets peuvent avoir le même employé qui y travaille.

Implicite dans ce modèle est le fait qu'il peut y avoir un partage des employés et des projets instances sur plusieurs instances de relation.

Ordinalité

Un rôle peut être précisé en déterminant s'il peut être présent ou non. C'est ce qu'on appelle l' *ordinalité* , et cela sert à montrer si l'entité cible doit être spécifié lors de la création de l'entité source. Parce que l'ordinalité n'est en réalité qu'un booléen valeur, il est également appelé le caractère facultatif de la relation.

En termes de cardinalité, l'ordinalité serait indiquée par la cardinalité étant une plage au lieu d'une simple valeur, et la plage commencerait par 0 ou 1 selon le ordinalité. Il est cependant plus simple de déclarer simplement que la relation est soit facultative ou obligatoire. Si facultatif, la cible peut ne pas être présente; si obligatoire, une entité source sans référence à son entité cible associée est dans un état non valide.

Présentation des mappages

Maintenant que vous connaissez suffisamment la théorie et que vous avez le contexte conceptuel pour être en mesure de discuter des relations, nous pouvons continuer à expliquer et à utiliser les mappages de relations.

Chacun des mappages est nommé d'après la cardinalité de la source et de la cible rôles. Comme indiqué dans les sections précédentes, une relation bidirectionnelle peut être considérée comme un paire de deux mappages unidirectionnels. Chacun de ces mappages est vraiment un cartographie des relations, et si nous prenons les cardinalités de la source et de la cible relation et combinez-les ensemble dans cet ordre, en les permutant avec les deux

Épisode 152

valeurs possibles de «un» et «plusieurs», nous nous retrouvons avec les noms suivants donnés au mappages:

- Plusieurs à un
- Un par un
- Un-à-plusieurs
- Plusieurs à plusieurs

Ces noms de mappage sont également les noms des annotations utilisées pour indiquent les types de relations sur les attributs mappés. Ils sont les base des annotations de relations logiques et contribuent à la modélisation d'objets aspects de l'entité. Comme les mappages de base, les mappages de relations peuvent être appliqués à soit des champs, soit des propriétés de l'entité.

Associations à valeur unique

Une association d'une instance d'entité à une autre instance d'entité (où la cardinalité de la cible est «un») est appelée une association à valeur unique. Le plusieurs-à-un et un- les mappages de relations à un appartiennent à cette catégorie car l'entité source fait référence à la plupart une entité cible. Nous discutons d'abord de ces relations et de certaines de leurs variantes.

Mappages plusieurs à un

Dans notre discussion de cardinalité sur la relation Employé et Département (montré dans la figure 4-8), nous avons d'abord pensé à un employé travaillant dans un service, alors nous a supposé qu'il s'agissait d'une relation individuelle. Cependant, lorsque nous avons réalisé que plus qu'un employé travaille dans le même service, nous l'avons changé en plusieurs-à-un cartographie des relations. Il s'avère que plusieurs-à-un est la cartographie la plus courante et est celui qui est normalement utilisé lors de la création d'une association avec une entité.

Figure 4-10 montre une relation plusieurs à un entre l'employé et le service. L'employé est le côté «multiple» et la source de la relation, et le ministère est le «Un» côté et la cible. Encore une fois, parce que la flèche pointe dans une seule direction, d'un employé au service, la relation est unidirectionnelle. Notez qu'en UML, le la classe source a un attribut implicite du type de classe cible s'il est possible d'y accéder. Pour exemple, Employee a un attribut appelé department qui contiendra une référence à un instance de département unique. L'attribut réel n'est pas affiché dans la classe Employee mais est impliquée par la présence de la flèche de relation.

Épisode 153

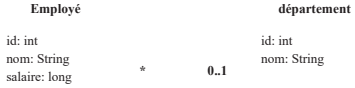

Figure 4-10. *Relation plusieurs à un d'un employé au service*

Un mappage plusieurs-à-un est défini en annotant l'attribut dans l'entité source

(l'attribut qui fait référence à l'entité cible) avec l'annotation @ManyToOne. Référencement4-16
montre comment l'annotation @ManyToOne est utilisée pour mapper cette relation. Le département
Le champ dans Employee est l'attribut source qui est annoté.

Liste 4-16. Relation plusieurs à un d'un employé au service

```
@Entité
Employé de classe publique {
    // ...
    @ManyToOne
    département du département privé;
    // ...
}
```

Nous n'avons inclus que les éléments de la classe qui sont pertinents pour notre discussion, mais vous pouvez voir dans l'exemple précédent que le code était plutôt anticlimatique. Un seul l'annotation était tout ce qui était nécessaire pour cartographier la relation, et il s'est avéré être assez terne, vraiment. Bien sûr, quand il s'agit de configuration, terne est beau.

Les mêmes types de flexibilité d'attribut et d'exigences de modificateur que ceux décrits pour les mappages de base s'appliquent également aux mappages de relations. L'annotation peut être présente sur le terrain ou sur la propriété, selon la stratégie utilisée pour l'entité.

Utilisation des colonnes de jointure

Dans la base de données, un mappage de relation signifie qu'une table a une référence à une autre table. Le terme de base de données pour une colonne qui fait référence à une clé (généralement la clé primaire) dans une autre table se trouve une *colonne de clé étrangère* . Dans JPA, elles sont appelées colonnes de jointure et @ L'annotation JoinColumn est l'annotation principale utilisée pour configurer ces types de Colonnes.

Épisode 154

> Remarquez plus loin dans le chapitre, nous parlons des colonnes de jointure présentes dans d'autres tables appelées tables de jointure. dans le chapitredix, nous couvrons un cas plus avancé d'utilisation d'un table de jointure pour les associations à valeur unique.

Considérez les tableaux EMPLOYÉ et DÉPARTEMENT illustrés à la Figure 4-11 qui correspondent aux entités Employé et Département. La table EMPLOYEE a une colonne de clé étrangère nommé DEPT_ID qui fait référence à la table DEPARTMENT. Du point de vue de l'entité relation, DEPT_ID est la colonne de jointure qui associe l'employé et le service entités.

Figure 4-11. *Tables EMPLOYÉ et DÉPARTEMENT*

Dans presque toutes les relations, indépendamment des côtés source et cible, l'un des deux côtés aura la colonne de jointure dans sa table. Ce côté est appelé le côté propriétaire ou le propriétaire de la relation. Le côté qui n'a pas la colonne de jointure est appelé le non-propriétaire ou côté inverse.

La propriété est importante pour le mappage car les annotations physiques qui définissent les mappages aux colonnes de la base de données (par exemple, @JoinColumn) sont toujours

défini du côté propriétaire de la relation. S'ils ne sont pas là, les valeurs sont
par défaut du point de vue de l'attribut du côté propriétaire.

Notez bien que nous ayons décrit le côté propriétaire comme étant déterminé par le schéma de données, le modèle objet doit indiquer le côté propriétaire via l'utilisation des annotations de mappage de relations. l'absence de l'élément mappedBy dans l'annotation de mappage implique la propriété de la relation, tandis que le la présence de l'élément mappedBy signifie que l'entité se trouve du côté inverse du relation. L'élément mappedBy est décrit dans les sections suivantes.

Épisode 155

Les mappages plusieurs à un sont toujours du côté propriétaire d'une relation, donc s'il y a un @JoinColumn à trouver dans la relation qui a un côté plusieurs-à-un, c'est là il sera localisé. Pour spécifier le nom de la colonne de jointure, l'élément name est utilisé. Par exemple, l'annotation @JoinColumn (name = "DEPT_ID") signifie que le DEPT_ID la colonne de la table d'entités source est la clé étrangère de la table d'entités cible, quel que soit le l'entité cible de la relation se trouve être.

Si aucune annotation @JoinColumn n'accompagne le mappage plusieurs à un, une valeur par défaut le nom de la colonne sera utilisé. Le nom utilisé par défaut est formé d'un combinaison des entités source et cible. C'est le nom de la relation attribut dans l'entité source, qui est department dans notre exemple, plus un trait de soulignement caractère (_), plus le nom de la colonne de clé primaire de l'entité cible. Donc si le L'entité Department a été mappée à une table qui avait une colonne de clé primaire nommée ID, le La colonne join dans la table EMPLOYEE est supposée être nommée DEPARTMENT_ID. Si ce n'est pas réellement le nom de la colonne, l'annotation @JoinColumn doit être définie sur remplacer la valeur par défaut.

Pour en revenir à la figure 4-11 , la colonne de clé étrangère est nommée DEPT_ID au lieu de nom de colonne DEPARTMENT_ID par défaut. Référencement4-17 montre l'annotation @JoinColumn utilisé pour remplacer le nom de la colonne de jointure par DEPT_ID.

Annonce 4-17. Relation plusieurs à un remplaçant la colonne de jointure

```
@Entité
Employé de classe publique {
    @Id id long privé;
    @ManyToOne
    @JoinColumn (nom = "DEPT_ID")
    département du département privé;
    // ...
}
```

Les annotations nous permettent de spécifier @JoinColumn sur la même ligne que @ManyToOne ou sur une ligne distincte, au-dessus ou en dessous. Par convention, le mappage logique doit apparaissent en premier, suivis du mappage physique. Cela rend le modèle d'objet clair car la partie physique est moins importante pour le modèle objet.

Épisode 156

Mappages un à un

Si un seul employé pouvait travailler dans un département, nous reviendrions au one-to-one association à nouveau. Un exemple plus réaliste d'association un-à-un, cependant, être un salarié disposant d'une place de parking. En supposant que chaque employé a été affecté son propre espace de stationnement, nous créerions une relation individuelle avec l'employé à ParkingSpace. Figure 4-12 montre cette relation.

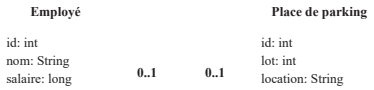

Figure 4-12. *Relation individuelle entre l'employé et le ParkingSpace*

Nous définissons le mappage de la même manière que nous définissons un plusieurs-à-un mapping, sauf que nous utilisons l'annotation @OneToOne au lieu d'un @ManyToOne annotation sur l'attribut parkingSpace. Tout comme avec un mappage plusieurs-à-un, le to- one a une colonne de jointure dans la base de données et doit remplacer le nom du colonne dans une annotation @JoinColumn lorsque le nom par défaut ne s'applique pas. Le défaut name est composé de la même manière que pour les mappages plusieurs-à-un en utilisant le nom du l'attribut source et le nom de la colonne de clé primaire cible.

Figure 4-13 montre les tables mappées par les entités Employee et ParkingSpace. La colonne de clé étrangère de la table EMPLOYEE est nommée PSPACE_ID et fait référence au Table PARKING_SPACE.

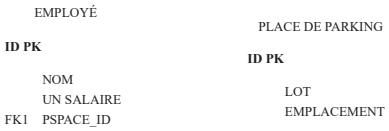

Figure 4-13. *Tables EMPLOYEE et PARKING_SPACE*

En fait, les mappages un à un sont presque les mêmes que les mappages plusieurs à un sauf qu'une seule instance de l'entité source peut faire référence à la même entité cible exemple. En d'autres termes, l'instance d'entité cible n'est pas partagée entre l'entité source instances. Dans la base de données, cela équivaut à avoir une contrainte d'unicité sur la source

Épisode 157

colonne de clé étrangère (c'est-à-dire la colonne de clé étrangère dans la table d'entité source). S'il y a s'il y avait plus d'une valeur de clé étrangère identique, cela contreviendrait à la règle qu'une seule instance d'entité source ne peut pas faire référence à la même instance d'entité cible.

Référencement 4-18 montre le mappage de cette relation. L'annotation @JoinColumn a été utilisé pour remplacer le nom de la colonne de jointure par défaut PARKINGSPACE_ID comme étant PSPACE_ID.

Annonce 4-18. Relation individuelle entre l'employé et l'espace de stationnement

```
@Entité
Employé de classe publique {
    @Id id long privé;
```

```
    nom de chaîne privé;
    @Un par un
    @JoinColumn (nom = "PSPACE_ID")
    Parking privéEspace de stationnementEspace;
    // ...
}
```

Mappages bidirectionnels un à un

L'entité cible du one-to-one a souvent une relation avec l'entité source;
Par exemple, ParkingSpace a une référence à l'employé qui l'utilise. Quand cela
Dans ce cas, on parle de relation bidirectionnelle un-à-un. Comme vous l'avez vu précédemment,
nous avons en fait deux mappages un-à-un séparés, un dans chaque direction, mais le
la combinaison des deux est appelée une relation bidirectionnelle un-à-un. Faire
notre exemple d'employé individuel et d'espace de stationnement bidirectionnel existant, nous avons besoin
ne modifiez le ParkingSpace que pour qu'il pointe vers l'employé. Figure 4-14 montre le
relation bidirectionnelle.

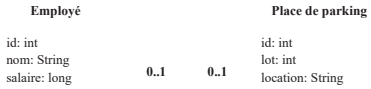

Figure 4-14. *Relation individuelle entre l'employé et ParkingSpace*

Épisode 158

Vous avez déjà appris que la table d'entités qui contient la colonne de jointure détermine
l'entité propriétaire de la relation. Dans une relation bidirectionnelle un-à-un,
les deux mappages sont des mappages un à un, et chaque côté peut être le propriétaire, donc le
la colonne de jointure peut finir par être d'un côté ou de l'autre. Ce serait normalement une donnée
décision de modélisation, pas une décision de programmation Java, et elle serait probablement décidée
basé sur le sens de traversée le plus fréquent.

Considérez la classe d'entité ParkingSpace présentée dans le Listing 4-19. Cet exemple suppose
le mappage de table illustré dans la figure 4-13, et il suppose que l'employé est le propriétaire
côté de la relation. Nous devons maintenant ajouter une référence de ParkingSpace à
Employé. Ceci est réalisé en ajoutant l'annotation de relation @OneToOne sur le
champ employé. Dans le cadre de l'annotation, nous devons ajouter un élément mappedBy pour indiquer
que le côté propriétaire est l'employé, pas le ParkingSpace. Parce que ParkingSpace est le
côté inverse de la relation, il n'a pas à fournir les informations de colonne de jointure.

Liste 4-19. Face inverse d'une relation bidirectionnelle un à un

```
@Entité
public class ParkingSpace {
    @Id id long privé;
    lot int privé;
    emplacement de chaîne privé;
    @OneToOne (mappedBy = "parkingSpace")
    employé privé;
    // ...
}
```

L'élément mappedBy dans le mappage un-à-un de l'attribut employee de
ParkingSpace est nécessaire pour faire référence à l'attribut parkingSpace dans la classe Employee. le
valeur de mappedBy est le nom de l'attribut dans l'entité propriétaire qui pointe vers le

entité inverse.

Épisode 159

Les deux règles pour les associations bidirectionnelles un-à-un sont donc les suivantes:

- L'annotation @JoinColumn va sur le mappage de l'entité qui est mappé à la table contenant la colonne de jointure, ou au propriétaire du relation. Cela pourrait être de chaque côté de l'association.
- L'élément mappedBy doit être spécifié dans @OneToOne annotation dans l'entité qui ne définit pas de colonne de jointure, ou côté inverse de la relation.

Il ne serait pas légal d'avoir une association bidirectionnelle mappée sur les deux côtés, tout comme il serait incorrect de ne pas l'avoir de chaque côté. La différence est que si elle étaient absents des deux côtés de la relation, le fournisseur traiterait chaque côté comme un relation unidirectionnelle indépendante. Ce serait bien sauf que cela supposerait que chaque côté était le propriétaire et que chacun avait une colonne de jointure.

Les relations plusieurs-à-un bidirectionnelles sont expliquées plus loin dans le cadre de la discussion d'associations bidirectionnelles à valeurs multiples.

Associations valorisées par la collection

Lorsque l'entité source fait référence à une ou plusieurs instances d'entité cible, un l'association ou la collection associée est utilisée. Le un-à-plusieurs et plusieurs-à-plusieurs les mappages répondent aux critères d'avoir de nombreuses entités cibles, et bien que le un-à-plusieurs l'association est la plus fréquemment utilisée, les mappages plusieurs-à-plusieurs sont également utiles quand il y a partage dans les deux sens.

Mappages un à plusieurs

Lorsqu'une entité est associée à une collection d'autres entités, elle est le plus souvent dans le forme d'un mappage un-à-plusieurs. Par exemple, un ministère aurait normalement un nombre d'employés. Figure 4-15 montre la relation employé et service que nous avons montré plus tôt dans la section «Mappages plusieurs-à-un», mais cette fois la relation est de nature bidirectionnelle.

Épisode 160

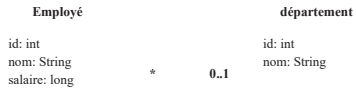

Figure 4-15. *Relation bidirectionnelle entre l'employé et le service*

Comme mentionné précédemment, lorsqu'une relation est bidirectionnelle, il y a en fait deux mappages, un pour chaque direction. Une relation bidirectionnelle un-à-plusieurs toujours implique un mappage plusieurs-à-un vers la source, donc dans notre employé et service exemple, il y a un mappage un-à-plusieurs d'un service à un employé et un un mappage de l'employé au service. On pourrait tout aussi bien dire que le la relation est bidirectionnelle plusieurs-à-un si nous la regardions de l'employé perspective. Ils sont équivalents car les relations plusieurs-à-un bidirectionnelles impliquent un mappage un-à-plusieurs de la cible vers la source, et vice versa.

Lorsqu'une entité source a un nombre arbitraire d'entités cibles stockées dans son collection, il n'existe aucun moyen évolutif de stocker ces références dans la table de base de données correspond à. Comment stockerait-il un nombre arbitraire de clés étrangères sur une seule ligne? Au lieu, il doit laisser les tables des entités de la collection avoir des clés étrangères vers la source table d'entité. C'est pourquoi l'association un-à-plusieurs est presque toujours bidirectionnelle et le côté «un» n'est normalement pas le côté propriétaire.

De plus, si les tables d'entités cibles ont des clés étrangères qui pointent vers la source table d'entité, les entités cibles doivent avoir des associations plusieurs-à-un vers la source objet entité. Avoir une clé étrangère dans une table pour laquelle il n'y a pas d'association dans le le modèle d'objet d'entité correspondant n'est pas fidèle au modèle de données. Il est néanmoins encore possible de configurer, cependant.

Regardons un exemple concret de mappage un-à-plusieurs basé sur l'employé et exemple de département illustré à la figure 4-15. Les tableaux de cette relation sont exactement les mêmes que ceux illustrés à la figure 4-11 , qui montrent une relation plusieurs-à-un. La seule différence entre l'exemple plusieurs-à-un et celui-ci est que nous sommes maintenant mettre en œuvre le côté inverse de la relation. Parce que l'employé a la colonne de jointure et est le propriétaire de la relation, la classe Employé est inchangée par rapport au Listing 4-16 .

Du côté du département de la relation, nous devons cartographier la collection d'employés d'entités Employee en tant qu'association un-à-plusieurs utilisant l'annotation @OneToMany. Référencement 4-20 montre la classe Department qui utilise cette annotation. Notez que parce que cela est le côté inverse de la relation, nous devons inclure l'élément mappedBy, tout comme nous l'avons fait dans l'exemple de la relation bidirectionnelle un-à-un.

Épisode 161

Liste 4-20. Relation un-à-plusieurs

```
@Entité
Département de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    @OneToMany (mappedBy = "département")
    les employés de la collection privée <Employee>;
    // ...
}
```

Il y a quelques points importants à mentionner à propos de cette classe. Le premier est qu'un La collection générique paramétrée par type est utilisée pour stocker les entités Employee. Cette fournit le typage strict qui garantit que seuls les objets de type Employee existeront dans la collection. Ceci est très utile car il ne fournit pas seulement une vérification à la compilation de notre code mais nous évite également d'avoir à effectuer des opérations de cast lorsque nous récupérons

les instances Employee de la collection.

JPA suppose la disponibilité des génériques; cependant, il est toujours parfaitement acceptable de utilisez une collection qui n'est pas paramétrée sur le type. Nous pourrions tout aussi bien définir le Classe Department sans utiliser de génériques mais en définissant uniquement un type de collection simple, comme nous l'aurions fait dans les versions de Java standard antérieures à Java SE 5 (sauf pour JDK 1.0 ou 1.1, lorsque java.util.Collection n'était même pas standardisé!). Si nous le faisions, nous le ferions besoin de spécifier le type d'entité qui sera stocké dans la collection qui est nécessaire par le fournisseur de persistance. Le code est affiché dans la liste4-21 et semble presque identique, sauf pour l'élément targetEntity qui indique le type d'entité.

Liste 4-21. Utilisation de targetEntity

```
@Entité
Département de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    @OneToMany (targetEntity = Employee.class, mappedBy = "département")
    les employés de la collection privée;
    // ...
}
```


Épisode 162

Il y a deux points importants à retenir lors de la définition du bidirectionnel relations un-à-plusieurs (ou plusieurs-à-un):

- Le côté plusieurs-à-un doit être le côté propriétaire, donc la colonne de jointure devrait être défini de ce côté.
- Le mappage un-à-plusieurs doit être le côté inverse, de sorte que le L'élément mappedBy doit être utilisé.

Si vous ne spécifiez pas l'élément mappedBy dans l'annotation @OneToMany, le provider pour le traiter comme une relation un-à-plusieurs unidirectionnelle définie pour être utilisée une table de jointure (décrite plus loin). C'est une erreur facile à faire et devrait être la première chose que vous recherchez si vous voyez une erreur de table manquante avec un nom qui a deux noms d'entité concaténées ensemble.

Mappages plusieurs-à-plusieurs

Lorsqu'une ou plusieurs entités sont associées à une collection d'autres entités, et entités ont des associations qui se chevauchent avec les mêmes entités cibles, nous devons le modéliser comme une relation plusieurs-à-plusieurs. Chacune des entités de chaque côté de la relation ont une association à valeur de collection qui contient des entités du type cible. Figure 4-16 montre une relation plusieurs-à-plusieurs entre l'employé et le projet. Chaque employé peut travailler sur plusieurs projets, et chaque projet peut être travaillé par plusieurs employés.

Figure 4-16. *Relation plusieurs-à-plusieurs bidirectionnelle*

Un mappage plusieurs à plusieurs est exprimé sur les entités source et cible sous la forme Annotation @ManyToMany sur les attributs de la collection. Par exemple, dans Listing4-22 le L'employé a un attribut de projets qui a été annoté avec @ManyToMany. Également, l'entité de projet a un attribut employés qui a également été annoté avec

@Plusieurs à plusieurs.

Épisode 163

Liste 4-22. Relation plusieurs-à-plusieurs entre l'employé et le projet

```
@Entité
Employé de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    @Plusieurs à plusieurs
    projets <Projet> de collection privée;
    // ...
}

@Entité
Projet de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    @ManyToMany (mappedBy = "projets")
    les employés de la collection privée <Employee>;
    // ...
}
```

Il existe des différences importantes entre cette relation plusieurs-à-plusieurs et la relation un-à-plusieurs évoquée plus haut. Le premier est une fatalité mathématique: lorsqu'une relation plusieurs-à-plusieurs est bidirectionnelle, les deux côtés de la relation sont mappages plusieurs à plusieurs.

La deuxième différence est qu'il n'y a pas de colonnes de jointure de chaque côté du relation. Vous verrez dans la section suivante que la seule façon d'implémenter un beaucoup de relations sont avec une table de jointure distincte. La conséquence de ne pas avoir de jointure colonnes dans l'une ou l'autre des tables d'entités est qu'il n'y a aucun moyen de déterminer de quel côté est le propriétaire de la relation. Parce que chaque relation bidirectionnelle doit avoir les deux un côté propriétaire et un côté inverse, nous devons choisir l'une des deux entités pour être le propriétaire. Dans cet exemple, nous avons choisi Employé comme propriétaire de la relation, mais nous pourrions avoir tout aussi facilement choisi Project à la place. Comme dans toute autre relation bidirectionnelle, le Le côté inverse doit utiliser l'élément mappedBy pour identifier l'attribut propriétaire.

Notez que quel que soit le côté désigné comme propriétaire, l'autre côté doit inclure l'élément mappedBy; sinon, le fournisseur pensera que les deux côtés sont propriétaire et que les mappages sont des relations unidirectionnelles distinctes.

Épisode 164

Utilisation des tables de jointure

Parce que la multiplicité des deux côtés d'une relation plusieurs-à-plusieurs est plurielle, ni

des deux tables d'entité peuvent stocker un ensemble illimité de valeurs de clé étrangère dans une seule entité rangée. Il faut utiliser une troisième table pour associer les deux types d'entités. Cette table d'association est appelé une table de jointure, et chaque relation plusieurs-à-plusieurs doit en avoir une. Ils pourraient être utilisé pour les autres types de relations également, mais ne sont pas obligatoires et sont donc moins commun.

Une table de jointure se compose simplement de deux colonnes de clé étrangère ou de jointure pour faire référence à chacun des deux types d'entités dans la relation. Une collection d'entités est ensuite mappée comme plusieurs lignes du tableau, chacune associant une entité à une autre. L'ensemble des lignes qui contiennent un identifiant d'entité donné dans la colonne de clé étrangère source représente le collection d'entités liées à cette entité donnée.

Figure 4-17 montre les tableaux EMPLOYEE et PROJECT pour l'employé et le projet entités et la table de jointure EMP_PROJ qui les associe. La table EMP_PROJ contient uniquement les colonnes de clé étrangère qui constituent sa clé primaire composée. La colonne EMP_ID fait référence à la clé primaire EMPLOYEE, tandis que la colonne PROJ_ID fait référence au PROJET clé primaire.

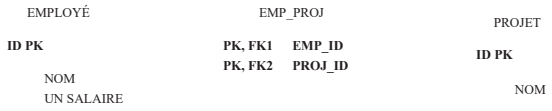

Figure 4-17. *Table de jointure pour une relation plusieurs-à-plusieurs*

Afin de mapper les tableaux décrits dans la figure 4-17, nous devons ajouter des métadonnées à la classe d'employé que nous avons désignée comme propriétaire de la relation. Liste 4 à 23 montre la relation plusieurs à plusieurs avec les annotations de table de jointure associées.

Liste 4-23. Utilisation d'une table de jointure

```
@Entité
Employé de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    @Plusieurs à plusieurs
```


Épisode 165


```
    @JoinTable (nom = "EMP_PROJ",
            joinColumns = @ JoinColumn (nom = "EMP_ID"),
            inverseJoinColumns = @ JoinColumn (name = "PROJ_ID"))
    projets <Projet> de collection privée;
    // ...
}
```

L'annotation @JoinTable est utilisée pour configurer la table de jointure pour la relation. Les deux colonnes de jointure de la table de jointure se distinguent par la propriété et côtés inverses. La colonne join au côté propriétaire est décrite dans joinColumns élément, tandis que la colonne de jointure vers le côté inverse est spécifiée par l'inverseJoinColumns élément. Vous pouvez voir dans le Listing 4-23 que les valeurs de ces éléments sont en fait Annotations @JoinColumn intégrées dans l'annotation @JoinTable. Cela fournit le possibilité de déclarer toutes les informations sur les colonnes de jointure dans la table qui définit leur. Les noms sont au pluriel pour les moments où il peut y avoir plusieurs colonnes pour chaque clé étrangère (soit l'entité propriétaire, soit l'entité inverse a une clé primaire en plusieurs parties). Ce cas plus compliqué est discuté au chapitre dix.

Dans notre exemple, nous avons entièrement spécifié les noms de la table de jointure et de ses colonnes car c'est le cas le plus courant. Mais si nous générions le schéma de base de données des entités, nous n'aurions pas réellement besoin de spécifier ces informations. Nous pourrions avoir

reposait sur les valeurs par défaut qui seraient supposées et utilisées lorsque la persistance
Le fournisseur génère la table pour nous. Lorsqu'aucune annotation @JoinTable n'est présente sur le
côté propriétaire, une table de jointure par défaut nommée <Owner> _ <Inverse> est supposée, où
<Owner> est le nom de l'entité propriétaire et <Inverse> est le nom de l'inverse ou
entité non propriétaire. Bien sûr, le propriétaire est essentiellement choisi au hasard par le développeur,
donc ces valeurs par défaut s'appliqueront en fonction de la façon dont la relation est mappée et
quelle que soit l'entité désignée comme propriétaire.

Les colonnes de jointure seront définies par défaut conformément aux règles par défaut de la colonne de jointure qui
ont été décrits précédemment dans la section «Utilisation des colonnes de jointure». Le nom par défaut du
la colonne de jointure qui pointe vers l'entité propriétaire est le nom de l'attribut à l'inverse
entité qui pointe vers l'entité propriétaire, ajoutée par un trait de soulignement et le nom de
colonne de clé primaire de la table d'entité propriétaire. Donc, dans notre exemple, l'employé est le
entité propriétaire, et le projet a un attribut employés qui contient la collection de
Instances d'employés. L'entité Employee est mappée à la table EMPLOYEE et a un
colonne clé de l'ID, de sorte que le nom par défaut de la colonne de jointure à l'entité propriétaire
être EMPLOYEES_ID. La colonne de jointure inverse serait également définie par défaut sur PROJECTS_ID.

Épisode 166

Il est assez clair que les noms par défaut d'une table de jointure et les colonnes de jointure dans
il est peu probable qu'il corresponde à une table existante. C'est pourquoi nous avons mentionné que le
les valeurs par défaut ne sont vraiment utiles que si le schéma de base de données mappé a été généré par
le fournisseur.

Mappages de collection unidirectionnels

Lorsqu'une entité a un mappage un-à-plusieurs vers une entité cible, mais que @OneToMany
l'annotation n'inclut pas l'élément mappedBy, il est supposé être dans un
relation unidirectionnelle avec l'entité cible. Cela signifie que l'entité cible
n'a pas de mappage plusieurs à un vers l'entité source. La figure 4-18 montre un
association un-à-plusieurs unidirectionnelle entre l'employé et le téléphone.

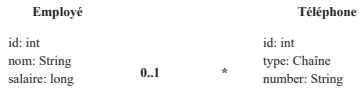

Figure 4-18. *Relation un-à-plusieurs unidirectionnelle*

Considérez le modèle de données de la figure 4-19. Il n'y a pas de colonne de jointure pour stocker le
association de retour du téléphone à l'employé. Par conséquent, nous avons utilisé une table de jointure pour
associez l'entité Phone à l'entité Employee.

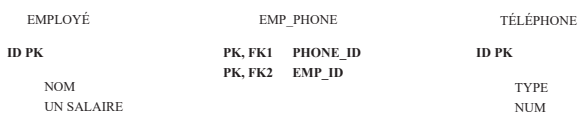

Figure 4-19. *Table de jointure pour une relation un-à-plusieurs unidirectionnelle*

De même, lorsqu'un côté d'une relation plusieurs-à-plusieurs n'a pas de mappage
à l'autre, c'est une relation unidirectionnelle. La table de jointure doit toujours être utilisée; le seul
la différence est que seul l'un des deux types d'entités utilise réellement la table pour charger ses
entités ou le met à jour pour stocker des associations d'entités supplémentaires.

Épisode 167

Dans ces deux cas unidirectionnels à valeur de collection, le code source est similaire aux exemples précédents, mais il n'y a pas d'attribut dans l'entité cible à référencer l'entité source, et l'élément mappedBy ne sera pas présent dans @OneToMany annotation sur l'entité source. La table de jointure doit maintenant être spécifiée dans le cadre du cartographie. Le listing 4-24 montre l'employé avec une relation un-à-plusieurs avec le téléphone à l'aide d'un table de jointure.

Listing 4-24. Relation unidirectionnelle un à plusieurs

```
@Entité
Employé de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    @OneToMany
    @JoinTable (nom = "EMP_PHONE",
            joinColumns = @ JoinColumn (nom = "EMP_ID"),
            inverseJoinColumns = @ JoinColumn (name = "PHONE_ID"))
    les téléphones <Phone> de la collection privée;
    // ...
}
```

Notez que lors de la génération du schéma, la dénomination par défaut des colonnes de jointure est légèrement différent dans le cas unidirectionnel car il n'y a pas d'attribut inverse. le le nom de la table de jointure serait par défaut EMPLOYEE_PHONE et aurait une colonne de jointure nommé EMPLOYEE_ID après le nom de l'entité Employee et sa colonne de clé primaire. La colonne de jointure inverse s'appellerait PHONES_ID, qui est la concaténation de l'attribut téléphones dans l'entité Employé et la colonne de clé primaire ID du PHONE table.

Relations paresseuses

Les sections précédentes ont montré comment configurer un attribut à charger lorsqu'il accédé et pas nécessairement avant. Vous avez appris que le chargement différé au niveau des attributs est normalement pas très bénéfique.

Au niveau des relations, cependant, le chargement paresseux peut être un grand avantage pour améliorer performance. Il peut réduire la quantité de SQL exécutée et accélérer les requêtes et le chargement d'objets considérablement.

Épisode 168

Le mode d'extraction peut être spécifié sur l'un des quatre types de mappage de relations. Lorsqu'il n'est pas spécifié sur une relation à valeur unique, l'objet associé est garanti chargé avec empressement. Les relations à valeur de collection sont chargées par défaut par défaut, mais parce que le chargement paresseux n'est qu'un indice pour le fournisseur, ils peuvent être chargés avec empressement si le fournisseur décide de le faire.

Dans les cas de relation bidirectionnelle, le mode de récupération peut être paresseux d'un côté mais avide de l'autre. Ce type de configuration est en fait assez courant car

les relations sont souvent accessibles de différentes manières selon la direction à partir de laquelle la navigation se produit.

Un exemple de remplacement du mode de récupération par défaut est si nous ne voulons pas charger le ParkingSpace pour un employé à chaque fois que nous chargeons l'employé. Le listing 4-25 montre le L'attribut parkingSpace est configuré pour utiliser le chargement différé.

Liste 4-25. Modification du mode de récupération sur une relation

```
@Entité
Employé de classe publique {
    @Id id long privé;
    @OneToOne (fetch = FetchType.LAZY)
    Parking privéEspace de stationnementEspace;
    // ...
}
```

> Tip une relation qui est spécifiée ou par défaut pour être chargée paresseusement peut ou pourrait ne provoque pas le chargement de l'objet associé lorsque la méthode getter est utilisée pour accéder L'object. l'objet peut être un proxy, il peut donc être nécessaire d'appeler une méthode dessus pour provoquer une faute.

Objets incorporés

Un objet incorporé est un objet qui dépend d'une entité pour son identité. Il n'a pas identité qui lui est propre, mais fait simplement partie de l'état d'entité qui a été stocké dans un objet Java séparé suspendu à l'entité. En Java, les objets embarqués ressemblent aux relations dans la mesure où elles sont référencées par une entité et apparaissent dans le Java sens être la cible d'une association. Dans la base de données, cependant, l'état du

l'objet incorporé est stocké avec le reste de l'état d'entité dans la ligne de base de données, sans distinction entre l'état de l'entité Java et celui de son objet incorporé.

> Astuce bien que les objets incorporés soient référencés par les entités qui les possèdent, on ne dit pas qu'ils sont en relation avec les entités. le terme relation peut ne s'applique que lorsque les deux côtés sont des entités.

Si la ligne de base de données contient toutes les données de l'entité et de son objet incorporé, pourquoi avoir un tel objet de toute façon? Pourquoi ne pas simplement définir les champs de l'entité à référencer tout son état de persistance au lieu de le diviser en un ou plusieurs sous-objets qui sont objets persistants de seconde classe dépendant de l'entité pour leur existence?

Cela nous ramène à l'inadéquation d'impédance relationnelle objet dont nous avons parlé au chapitre 1 . Étant donné que l'enregistrement de base de données contient plus d'un type logique, il rend sens pour rendre cette séparation explicite dans le modèle objet de l'application même si la représentation physique est différente. On pourrait presque dire que l'objet incorporé est une représentation plus naturelle du concept de domaine qu'une simple collection de attributs sur l'entité. De plus, une fois que vous avez identifié un groupement d'état d'entité qui constitue un objet incorporé, vous pouvez partager le même type d'objet incorporé avec d'autres entités qui ont également la même représentation interne. [1]

Les informations d'adresse sont un exemple d'une telle réutilisation. Figure 4-20 montre un EMPLOYÉ table qui contient un mélange d'informations de base sur les employés ainsi que des colonnes correspondent à l'adresse du domicile du salarié.

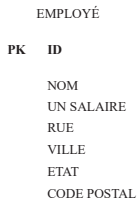

Figure 4-20. *Table EMPLOYEE avec informations d'adresse intégrées*

1 Même si les types incorporés peuvent être partagés ou réutilisés, les instances ne le peuvent pas. Un intégré l'instance d'objet appartient à l'entité qui la référence; et aucune autre instance d'entité, de cette entité type ou tout autre, peut référencer la même instance incorporée.

Épisode 170

Les colonnes STREET, CITY, STATE et ZIP_CODE se combinent logiquement pour former le adresse. Dans le modèle objet, c'est un excellent candidat pour être abstrait dans un un type intégré d'adresse distinct au lieu de lister chaque attribut de la classe d'entité. La classe d'entité aurait alors simplement un attribut d'adresse pointant vers un objet de type Adresse. Figure 4-21 montre comment l'employé et l'adresse se rapportent à chacun autre. L'association de composition UML est utilisée pour indiquer que l'employé possède l'adresse et qu'une instance de l'adresse ne peut être partagée par aucun autre objet autre que l'instance Employee qui en est propriétaire.

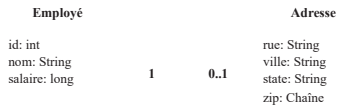

Figure 4-21. *Employé et adresse intégrée*

Avec cette représentation, non seulement les informations d'adresse sont parfaitement encapsulées dans un objet mais si une autre entité telle que la société possède également des informations d'adresse, peut également avoir un attribut qui pointe vers son propre objet Address incorporé. Nous décrivons ce scénario dans la section suivante.

Un type incorporé est marqué comme tel en ajoutant l'annotation @Embeddable au définition de classe. Cette annotation sert à distinguer la classe des autres Java classiques les types. Une fois qu'une classe a été désignée comme intégrable, ses champs et propriétés sera persistant dans le cadre d'une entité. Nous pourrions également vouloir définir le type d'accès de l'objet intégrable afin qu'il soit accédé de la même manière quelle que soit l'entité dont il s'agit intégré dans. Listing 4-26 montre la définition du type intégré d'adresse.

Liste 4-26. Type d'adresse intégrable

```
@Embeddable @Access (AccessType.FIELD)
Adresse de classe publique {
    rue privée String;
    ville privée de String;
    état de chaîne privé;
    @Column (nom = "ZIP_CODE")
    zip de chaîne privé;
    // ...
}
```

Épisode 171

Pour utiliser cette classe dans une entité, l'entité doit avoir uniquement un attribut du type intégrable. L'attribut est éventuellement annoté avec l'annotation @Embedded pour indiquer qu'il s'agit d'un mappage intégré. Référencement4-27 montre la classe Employee utilisant un objet Address intégré.

Liste 4-27. Utilisation d'un objet incorporé

```
@Entité
Employé de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    long salaire privé;
    @ Adresse d'adresse privée intégrée;
    // ...
}
```

Lorsque le fournisseur persiste une instance de Employee, il accède aux attributs de l'objet Address comme s'ils étaient présents sur l'instance d'entité elle-même. Colonne les mappages sur le type d'adresse concernent vraiment les colonnes de la table EMPLOYEE, même bien qu'ils soient répertoriés dans un type différent.

La décision d'utiliser des objets ou des entités incorporés dépend si vous pensez vous aurez jamais besoin de créer des relations avec eux ou à partir d'eux. Les objets incorporés sont pas censé être des entités, et dès que vous commencez à les traiter comme des entités, vous devez probablement en faire des entités de première classe au lieu d'objets incorporés si le modèle de données le permet.

Astuce, il n'est pas portable de définir des objets incorporés dans le cadre de l'héritage hiérarchies. Une fois qu'ils commencent à s'étendre, la complexité de l'intégration elles augmentent et le rapport valeur / coût diminue.

Avant d'en arriver à notre exemple, nous avons mentionné qu'une classe Address pouvait être réutilisée dans les entités des employés et de l'entreprise. Idéalement, nous aimerions que la représentation montrée dans Figure 4-22. Même si les classes Employé et Entreprise comprennent l'adresse classe, ce n'est pas un problème car chaque instance d'Adresse ne sera utilisée que par un seul Instance d'employé ou d'entreprise.

Épisode 172

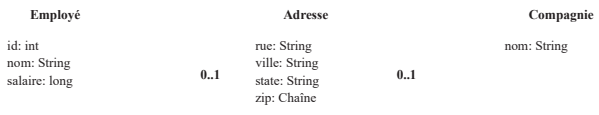

Figure 4-22. *Adresse partagée par deux entités*

Étant donné que les mappages de colonnes du type intégré Adresse s'appliquent aux colonnes de l'entité contenante, vous vous demandez peut-être comment le partage pourrait être possible si les deux les tables d'entité ont des noms de colonne différents pour les mêmes champs. La figure 4-23 montre ce problème. La table COMPANY correspond aux attributs par défaut et mappés du

Type d'adresse défini précédemment, mais la table EMPLOYEE de cet exemple a été remplacée par
correspondent aux exigences d'adresse d'une personne vivant au Canada. Nous avons besoin d'un moyen pour une entité
pour mapper l'objet incorporé en fonction de ses propres besoins de table d'entités, et nous en avons un dans
l'annotation @AttributeOverride.

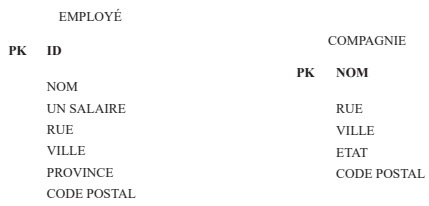

Figure 4-23. *Tables EMPLOYÉ et ENTREPRISE*

Nous utilisons une annotation @AttributeOverride pour chaque attribut de la
objet que nous voulons remplacer dans l'entité. Nous annotons le champ ou la propriété incorporé
dans l'entité et spécifiez dans l'élément de nom le champ ou la propriété dans le
objet que nous ignorons. L'élément column nous permet de spécifier la colonne qui
l'attribut est mappé dans la table d'entités. Nous l'indiquons sous la forme d'un
Annotation @Column imbriquée. Si nous remplaçons plusieurs champs ou propriétés, nous pouvons utiliser
l'annotation @AttributeOverrides au pluriel et imbriquer plusieurs @AttributeOverride
annotations à l'intérieur.

Épisode 173

Référencement 4-28 montre un exemple d'utilisation de l'adresse à la fois dans l'employé et dans l'entreprise. le
L'entité société utilise le type d'adresse sans changement, mais l'entité employé spécifie
deux substitutions d'attributs pour mapper les attributs state et zip de l'adresse à la
Colonnes PROVINCE et POSTAL_CODE de la table EMPLOYEE.

Liste 4-28. Réutilisation d'un objet incorporé dans plusieurs entités

```
@Entité
Employé de classe publique {
    @Id id long privé;
    nom de chaîne privé;
    long salaire privé;
    @Embedded
    @AttributeOverrides ({
        @AttributeOverride (nom = "état", colonne = @ Colonne (nom = "PROVINCE")),
        @AttributeOverride (name = "zip", column = @ Column (name = "POSTAL_CODE"))
    })
    adresse d'adresse privée;
    // ...
}
@Entité
société de classe publique {
    @Id nom de chaîne privé;
    @Embedded
    adresse d'adresse privée;
    // ...
}
```

Résumé

Le mappage d'objets à des bases de données relationnelles est d'une importance cruciale pour la persistance
applications. Le traitement de la discordance d'impédance nécessite une suite sophistiquée de
métadonnées. JPA fournit non seulement ces métadonnées, mais facilite également
développement.

Épisode 174

Dans ce chapitre, nous avons parcouru le processus de cartographie de l'état de l'entité qui comprenait
types Java simples, objets volumineux, types énumérés et types temporels. Nous avons également utilisé le
métadonnées pour effectuer un mappage de rencontre au milieu avec des noms de table et des colonnes spécifiques.

Nous avons expliqué comment les identifiants sont générés et décrit quatre stratégies différentes
de génération. Vous avez vu les différentes stratégies en action et appris à vous différencier
les uns des autres.

Nous avons ensuite passé en revue certains des concepts de relation et les avons appliqués à l'objet
métadonnées de cartographie relationnelle. Nous avons utilisé des colonnes de jointure et des tables de jointure pour mapper des
associations valorisées et valorisées par la collection et a passé en revue quelques exemples. Nous avons aussi
discuté des types spéciaux d'objets appelés éléments incorporables qui sont mappés mais qui n'ont pas
identificateurs et ne peut exister qu'au sein d'entités persistantes.

Le chapitre suivant traite davantage des subtilités de la cartographie à valeur de collection
relations, ainsi que comment mapper des collections d'objets non-entité. Nous plongeons dans le
différents types de collection et la manière dont ces types peuvent être utilisés et mappés, et
voir comment ils affectent les tables de base de données auxquelles le mappage est effectué.

Épisode 175

CHAPITRE 5

Cartographie de la collection

Parfois, une collection est utilisée comme une caisse à lait: c'est juste un simple contenant sans ordre apparent ou organisation prévue. D'autres cas exigent une sorte de système de ordonner et organiser de sorte que la manière dont les objets sont récupérés de la collection ait un sens. Que la collection soit du premier type ou du second, les collections d'objets nécessitent plus d'efforts pour cartographier que des objets isolés, bien qu'en compensation ils offrent plus souplesse.

Dans le dernier chapitre, nous avons commencé le voyage de cartographie des relations valorisées par la collection, spooning uniquement les bases du mappage des collections d'entités à la base de données. Cette Le chapitre explique plus en détail comment nous pouvons mapper des types de collections plus sophistiqués, comme des listes ordonnées de manière persistante et des cartes avec des clés et des valeurs de différentes types d'objets. Nous explorons même comment mapper des collections d'objets qui ne sont pas des entités.

Relations et collections d'éléments

Lorsque nous parlons de mappage de collections, il existe en fait trois types d'objets qui nous pouvons stocker dans des collections mappées. Nous pouvons mapper des collections d'entités, intégrables, ou types de base, et chacun nécessite un certain niveau de compréhension pour être correctement cartographié et utilisé efficacement.

Nous devrions clarifier un point potentiel de confusion sur ces types d'objets lorsqu'ils sont stockés dans des collections. Dans le chapitre précédent, nous avons présenté le concept des relations d'un type d'entité à un autre, et vous avez appris que lorsque le l'entité source a une collection contenant des instances du type d'entité cible qu'elle est appelée une relation à plusieurs valeurs. Cependant, les collections de types intégrables et de base sont pas des relations; ce sont simplement des collections d'éléments que l'on appelle ainsi *élément collections* . Les relations définissent les associations entre entités indépendantes, alors que les collections d'éléments contiennent des objets qui dépendent de l'entité de référence, et peuvent être récupérés uniquement via l'entité qui les contient.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_5

Épisode 176

Une différence pratique entre les relations et les collections d'éléments est la annotation utilisée pour les désigner. Une relation nécessite au minimum annotation de relation, soit @OneToMany ou @ManyToMany, alors qu'un élément La collection est indiquée par l'annotation @ElementCollection. En supposant que Classe intégrable VacationEntry dans la liste 5-1, Le Listing 5-2 montre un exemple de collection d'éléments d'éléments incorporables dans l'attribut vacationBookings, ainsi qu'un collection d'éléments de types de base (String) dans l'attribut nickNames.

Liste 5-1. VacationEntry intégrable

```
@Embeddable
public class VacationEntry {
    @Temporal (TemporalType.DATE)
    calendrier privé startDate;
```

```
    @Column (nom = "JOURS")
    private int daysTaken;
    // ...
}
```

Liste 5-2. Collections d'éléments d'éléments incorporables et de types de base

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;
    // ...

    @ElementCollection (targetClass = VacationEntry.class)
    réservation de vacances Collection privée;

    @ElementCollection
    private Set <String> nickNames;

    // ...
}
```

Épisode 177

Vous pouvez voir dans le Listing 5-2 que, comme les annotations de relation, le L'annotation @ElementCollection comprend un élément targetClass utilisé pour spécifier la classe si la Collection ne définit pas le type d'élément qu'elle contient. Ça aussi inclut un élément fetch pour indiquer si la collection doit être chargée paresseusement.

Un aspect plus intéressant des mappages dans Listing 5-2 est l'absence de tout métadonnées supplémentaires. Rappelez-vous que les éléments qui sont stockés dans les collections ne sont pas des entités et n'ont donc pas de table mappée. Les intégrables sont censés être stockés dans la même table que l'entité qui y fait référence, mais s'il existe une collection de incorporables, comment serait-il possible de stocker une multiplicité d'objets mappés de la même manière dans une seule rangée? De même pour les types de base, nous ne pouvions pas mapper chaque chaîne de pseudonyme à une colonne dans la table EMPLOYEE et prévoyez de stocker plusieurs chaînes sur une seule ligne. Pour cette raison, les collections d'éléments nécessitent une table séparée appelée *table de collection.* Chaque table de collection doit avoir une colonne de jointure qui fait référence à l'entité contenante table. Des colonnes supplémentaires dans la table de collection sont utilisées pour mapper les attributs du élément intégrable, ou l'état de l'élément de base si l'élément est d'un type de base.

Nous pouvons spécifier une table de collection à l'aide d'une annotation @CollectionTable, ce qui permet nous pour désigner le nom de la table, ainsi que la colonne de jointure. Les valeurs par défaut s'appliqueront si l'annotation ou les éléments spécifiques de cette annotation ne sont pas spécifiés. La table name sera par défaut le nom de l'entité de référence, ajouté d'un trait de soulignement et le nom de l'attribut d'entité qui contient la collection d'éléments. La colonne de jointure par défaut est de même le nom de l'entité de référence, ajouté d'un trait de soulignement et le nom de la colonne de clé primaire de la table d'entités. Parce qu'aucune table de collecte n'était spécifié dans l'une ou l'autre des collections d'éléments dans vacationBookings et nickNames attributs de l'entité Employee définie dans le Listing 5-2 , ils sont définis par défaut pour utiliser la collection tables nommées EMPLOYEE_VACATIONBOOKINGS et EMPLOYEE_NICKNAMES, respectivement. le La colonne join dans chacune des tables de collection sera EMPLOYEE_ID, qui est juste le nom de l'entité associée à la colonne de clé primaire Employee mappée.

Nous mappons les champs ou propriétés du type intégrable aux colonnes du table de collection au lieu de la table primaire de l'entité, avec le nom de colonne habituel les règles par défaut s'appliquent. Lorsque la collection d'éléments contient des types de base, les valeurs

sont également stockés dans une colonne de la table de collection, le nom de colonne par défaut étant le nom de l'attribut d'entité. En appliquant cette règle, les surnoms seraient stockés dans le Colonne NICKNAMES. Une fois toutes les valeurs par défaut appliquées, les tables mappées ressembleront à ceux de la figure 5-1 .

Épisode 178

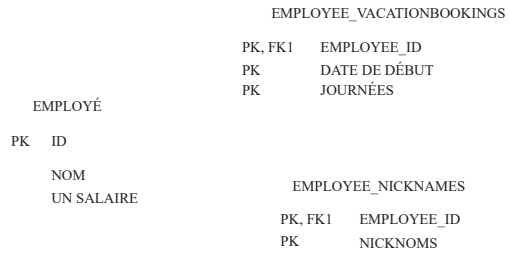

Figure 5-1. *Table d'entité EMPLOYEE et tables de collection mappées*

Lorsque nous avons abordé pour la première fois les éléments incorporables, vous avez vu comment les attributs étaient mappés dans l'objet incorporable mais pourrait être remplacé lorsqu'il est incorporé dans d'autres entités ou éléments incorporables. Nous avons utilisé l'annotation @AttributeOverride pour remplacer les noms des colonnes. La même annotation peut également être utilisée pour remplacer le attributs dans les éléments d'une collection d'éléments. Dans la liste5-3 , les jours l'attribut est en cours de remappage, à l'aide de @AttributeOverride, à partir de son stockage dans les DAYS colonne à être stockée dans la colonne DAYS_ABS. Une différence importante entre en utilisant @AttributeOverride sur des mappages incorporés simples et en l'utilisant pour remplacer le colonnes d'éléments incorporables dans une collection d'éléments est que dans ce dernier cas, la colonne spécifié par @AttributeOverride s'applique en fait à la table de collection, pas à l'entité table.

Liste 5-3. Remplacement des colonnes de la table de collection

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;
    // ...

    @ElementCollection (targetClass = VacationEntry.class)
    @CollectionTable (
        name = "VACATION", joinColumns = @ JoinColumn (name = "EMP_ID"))
```


Épisode 179


```
    @AttributeOverride (name = "daysTaken", column = @ Column (name = "DAYS_ABS"))
    réservation de vacances Collection privée;
```

```
    @ElementCollection
    @Column (nom = "NICKNAME")
    private Set <String> nickNames;
    // ...
}
```

Afin de remplacer le nom de la colonne dans laquelle les surnoms sont stockés, nous peut utiliser l'annotation @Column, en se rappelant à nouveau que le nom spécifie une colonne dans la table de collection, pas la table d'entité. La figure 5-2 montre les tables mappées, y compris la table de collection VACATION remplacée mappée par la collection vacationBookings.

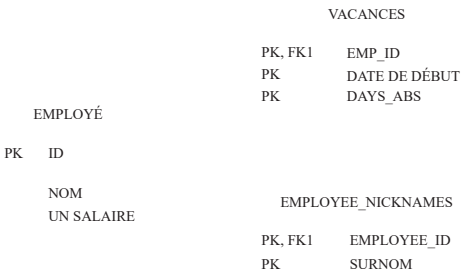

Figure 5-2. *Table d'entité EMPLOYEE et tables de collection mappées avec remplace*

Utilisation de différents types de collections

Nous pouvons utiliser différents types de collections pour stocker des associations d'entités à plusieurs valeurs et collections d'objets. En fonction des besoins de l'application, n'importe lequel de Collection, Set, List et Map peuvent être appropriés. Il existe des règles correspondant au type de collection, cependant, qui guident son utilisation, donc avant d'utiliser un type de collection donné, vous devez être familiarisé avec les règles qui régissent la façon dont ce type peut être mappé et manipulé.

Épisode 180

La première étape consiste à définir la collection comme étant l'un des types d'interface mentionné précédemment. Vous initialisez ensuite l'attribut avec une implémentation concrète classe. Cela peut être fait dans un constructeur ou une méthode d'initialisation de la classe d'entité et vous permet de placer des objets dans la collection d'implémentation d'une entité nouvelle ou non persistante.

Une fois que l'entité devient gérée ou a persisté au moyen d'un EntityManager.persist (), l'interface doit toujours être utilisée lors de l'utilisation la collection, qu'elle ait été lue dans la base de données ou qu'elle ait été détachée du gestionnaire d'entité. En effet, au moment où l'entité devient gérée, le le fournisseur de persistance peut remplacer l'instance concrète initiale par une autre instance d'une classe d'implémentation Collection qui lui est propre.

Ensembles ou collections

Le type de collection le plus couramment utilisé dans les associations est la collection standard superinterface. Ceci est utilisé quand peu importe quelle implémentation est en dessous et lorsque les méthodes de collecte courantes sont suffisantes pour accéder au entités qui y sont stockées.

Un ensemble empêchera l'insertion d'éléments en double et pourrait être plus simple

et un modèle de collection plus concis, tandis qu'une interface de collection vanilla est la plus générique. Aucune de ces interfaces ne nécessite d'annotations supplémentaires au-delà de l'original annotation de mappage pour les spécifier plus en détail. Ils sont utilisés de la même manière que s'ils détenaient objets non persistants. Un exemple d'utilisation d'une interface Set pour une collection d'éléments et une collection pour une autre est dans la liste 5-3.

Listes

Un autre type de collection courant est la liste. Une liste est généralement utilisée lorsque les entités ou les éléments doivent être récupérés dans un ordre défini par l'utilisateur. Parce que la notion d'ordre des lignes dans la base de données n'est pas généralement définie, la tâche de déterminer la commande doit être avec l'application.

Il existe deux façons de déterminer l'ordre de la liste. Le premier est de le cartographier de manière à ce qu'il est ordonné selon l'état existant dans chaque entité ou élément de la liste. C'est le méthode la plus simple et moins intrusive sur le modèle de données. Le second consiste à maintenir l'ordre de la liste dans une colonne de base de données supplémentaire. Il est plus compatible avec Java qu'il prend en charge la sémantique de classement traditionnelle d'une liste Java, mais peut être beaucoup moins performant, comme vous le verrez dans les sections suivantes.

Épisode 181

Classement par entité ou attribut d'élément

L'approche la plus courante pour classer les entités ou éléments dans une liste consiste à spécifier une règle de classement basée sur la comparaison d'un attribut particulier de l'entité ou élément. Si la liste est une relation, l'attribut est le plus souvent la clé primaire du entité cible.

Nous indiquons l'attribut à classer dans l'annotation @OrderBy. La valeur du l'annotation est une chaîne qui contient un ou plusieurs champs ou propriétés séparés par des virgules de l'objet commandé. Chacun des attributs peut éventuellement être suivi d'un Mot clé ASC ou DESC pour définir si l'attribut doit être ordonné par ordre croissant ou ordre décroissant. Si la direction n'est pas spécifiée, la propriété sera commandée dans ordre croissant.

Si la liste est une relation et fait référence à des entités, spécifiez @OrderBy sans champs ou propriétés, ou ne pas les spécifier du tout, entraînera le classement de la liste par le clés primaires des entités de la liste. Dans le cas d'une collection d'éléments de base types, alors la liste sera triée par les valeurs des éléments. Collections d'éléments de les types intégrables entraîneront la valeur par défaut de la liste dans un ordre non défini, généralement dans l'ordre renvoyé par la base de données en l'absence de clause ORDER BY.

L'exemple de retour dans la liste 4-20 du chapitre précédent avait un un-à-plusieurs relation du ministère à l'employé. Si nous voulons que les employés soient dans un order, nous pouvons utiliser une liste au lieu d'une collection. En ajoutant une annotation @OrderBy sur la cartographie, nous pouvons indiquer que nous voulons que les employés soient classés par ordre croissant ordre alphabétique par nom. Le Listing 5-4 montre l'exemple mis à jour.

Liste 5-4. Relation un-à-plusieurs à l'aide d'une liste

```
@Entité
Département de classe publique {
    // ...
    @OneToMany (mappedBy = "département")
    @OrderBy ("nom ASC")
    Liste privée des employés <Employee>;
    // ...
}
```

Nous n'avons pas besoin d'inclure l'ASC dans les annotations @OrderBy car ce serait

croissant par défaut, mais il est judicieux de l'inclure.

Épisode 182

Nous aurions pu tout aussi facilement ordonner la liste des employés par un champ intégré de Employé. Par exemple, si le nom avait été intégré dans un champ Employé intégré appelé info qui était de type Embeddable EmployeeInfo, nous écririons l'annotation comme @OrderBy ("info.name ASC").

Nous pourrions également souhaiter avoir des sous-ordres utilisant plusieurs attributs. Nous pouvons le faire en en spécifiant des paires de <nom d'attribut ASC / DESC> séparées par des virgules dans l'annotation. Pour exemple, si l'employé avait un statut, nous aurions pu le classer par statut puis par nom en utilisant une annotation @OrderBy de @OrderBy ("statut DESC, nom ASC"). Bien sûr, la condition préalable à l'utilisation d'un attribut dans une annotation @OrderBy est que le type d'attribut doit être comparable, ce qui signifie qu'il prend en charge les opérateurs de comparaison.

Si vous deviez simplement changer l'ordre de deux employés dans la liste, cela pourrait apparaître qu'ils occupaient de nouveaux postes dans la Liste. Cependant, si dans une nouvelle persistance contexte dans lequel vous relisez le département et accédez à ses employés, la liste reviendrait dans l'ordre dans lequel il était avant que vous ne le manipuliez [1]. Ceci est dû au fait L'ordre de liste est basé sur l'ordre de classement affirmé par l'annotation @OrderBy. Le simple fait de changer l'ordre des éléments dans une liste en mémoire n'entraînera pas que cet ordre soit stocké dans la base de données au moment de la validation. En fait, l'ordre spécifié dans @OrderBy sera utilisé uniquement lors de la lecture de la liste en mémoire. En règle générale, l'ordre de la liste doit toujours être conservé en mémoire pour être cohérent avec les règles de classement @OrderBy.

Listes ordonnées en permanence

Un autre exemple qui appelle à l'ordre fourni par List est une file d'attente d'impression qui garde un liste des travaux d'impression mis en file d'attente à un moment donné. La PrintQueue est essentiellement une file d'attente FIFO (First In First Out) qui, lorsque l'imprimante est disponible, prend PrintJob depuis le début de la file d'attente et l'envoie à l'imprimante pour impression. En supposant que PrintQueue et PrintJob sont des entités, nous aurions une relation un-à-plusieurs de PrintQueue à PrintJob et une relation plusieurs-à-un.

Étant donné que vous savez comment cartographier les relations, vous venez d'apprendre à mapper les listes ordonnées en utilisant @OrderBy, il semblerait assez simple de mapper cela relation à l'aide d'une liste. L'entité PrintJob, dans le Listing 5-5 , illustre son plusieurs-à-un côté du mappage bidirectionnel.

[1] En supposant que la collection n'a pas été renvoyée à partir d'un cache partagé de deuxième niveau.

Épisode 183

Liste 5-5. Entité PrintJob

```
@Entité
public class PrintJob {
    @Id id int privé;
    // ...
```

```
    @ManyToOne
    file d'attente PrintQueue privée;
    // ...
}
```

Le problème survient lorsque nous découvrons que l'entité PrintJob n'a pas de attribut qui peut être utilisé dans @OrderBy. Parce que l'ordre du travail n'affecte pas vraiment le PrintJob réel qui est traité, la décision a été prise de ne pas stocker la commande d'un travail donné au sein de l'entité PrintJob. La position d'un PrintJob particulier dans la file d'attente est déterminé simplement par sa position dans la liste des emplois.

Les entités PrintJob de la liste Java ne peuvent pas conserver leur ordre à moins qu'un La colonne de base de données a été créée pour la stocker. Nous appelons cette colonne la colonne de commande, et il fournit un ordre persistant plus fort que @OrderBy. C'est dans la colonne de commande que l'ordre de l'objet est stocké et mis à jour lorsqu'il est déplacé d'une position à une autre dans la même liste. Il est transparent pour l'utilisateur en ce que l'utilisateur n'a pas besoin de le manipuler, ou même nécessairement en être conscient, afin d'utiliser la Liste. Il a besoin être connu et considéré comme faisant partie du processus de cartographie, cependant, et est déclaré par signifie une annotation @OrderColumn.

L'utilisation d'une annotation @OrderColumn empêche l'utilisation de @OrderBy, et vice versa. Référencement 5-6 montre comment @OrderColumn peut être utilisé avec notre relation un-à-plusieurs mappage dans PrintQueue. Les mappages de table sont illustrés dans la figure 5-3.

Liste 5-6. Liste un-à-plusieurs de PrintQueue à PrintJob

```
@Entité
classe publique PrintQueue {
    @Id nom de chaîne privé;
    // ...
    @OneToMany (mappedBy = "file d'attente")
```



```
    @OrderColumn (nom = "PRINT_ORDER")
    travaux de <PrintJob> de liste privée;
    // ...
}
```

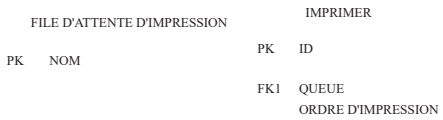

Figure 5-3. *Table PRINTQUEUE et table PRINTJOB cible avec ordre colonne*

Vous avez probablement remarqué quelque chose de différent dans la déclaration de la colonne de commande du côté un-à-plusieurs de la relation. Dans le dernier chapitre, nous avons expliqué la pratique de mapper les colonnes physiques du côté propriétaire car c'est le côté qui possède la table dans laquelle ils s'appliquent. La colonne de commande est une exception à cette règle lorsque la relation est bidirectionnelle car la colonne est toujours définie à côté du Indiquez qu'il est en train de commander, même s'il est dans la table mappée par le propriétaire many-to-un côté entité. Donc dans la liste5-6, l'annotation @OrderColumn est du côté PrintQueue de la relation, mais la colonne nommée PRINT_ORDER fait référence à la table L'entité PrintJob est mappée à.

Bien que l'annotation @OrderColumn doive être présente pour activer la commande position de l'entité à stocker dans une colonne de base de données, les éléments de l'annotation sont facultatifs. Le nom par défaut est simplement le nom de l'attribut d'entité, ajouté par le _ORDER chaîne. Donc, si le nom n'avait pas été remplacé dans la liste5-6 pour être PRINT_ORDER, il aurait été par défaut JOBS_ORDER.

La table dans laquelle la colonne de commande est stockée dépend du mappage dans lequel @OrderColumn est appliqué à. C'est généralement dans la table qui stocke l'entité ou élément en cours de stockage. Comme mentionné, dans notre relation bidirectionnelle un-à-plusieurs dans Annonce 5-6, l'entité stockée est PrintJob et la colonne de commande serait stockée dans la table PRINTJOB. Si le mappage était une collection d'éléments, la colonne de commande serait stocké dans la table de collection. Dans les relations plusieurs-à-plusieurs, l'ordre la colonne est dans la table de jointure.

Épisode 185

Quelques commentaires supplémentaires sur l'utilisation de @OrderColumn sont nécessaires car il s'agit d'un fonctionnalité qui pourrait facilement être mal utilisée. Nous avons dit que la colonne de commande est transparente pour le utilisateur de la liste, mais il s'avère que cette transparence peut avoir des répercussions inattendues pour un utilisateur naïf.

Considérez une entreprise occupée avec beaucoup de monde et de nombreux travaux d'impression soumis et imprimé. Lorsqu'un travail atteint la première position, il est supprimé de la file d'attente et envoyé à l'imprimante. Pendant ce temps, un autre emploi hérite du poste «sur le pont». Chaque fois qu'un travail est traité, tous les autres travaux d'impression restant dans la file d'attente augmentent de une position et est un pas de plus pour être imprimé. En d'autres termes, avec chaque imprimé travail, l'ordre de chaque autre PrintJob change et doit être réenregistré dans le base de données. Dans notre cas, la colonne de commande est stockée dans la table dans laquelle PrintJob les entités sont stockées: la table PRINTJOB.

Inutile de dire que nous envisageons un coût potentiellement élevé, aggravé par la la file d'attente s'allonge. Pour chaque travail ajouté à une file d'attente de taille n, il y aura n SQL supplémentaire mises à jour envoyées à la base de données pour modifier l'ordre de cette tâche avant même qu'elle ne parvienne au imprimante. Cela pourrait sonner l'alarme pour un administrateur de base de données, en particulier un administrateur vigilant avec un penchant pour parcourir les audits SQL.

En guise de commentaire final sur l'utilisation de la liste, il existe un support spécial dans les requêtes JPA qui permet des sous-ensembles commandés ou des éléments individuels d'une liste à consulter et à renvoyer. Vous voyez comment ceci peut être réalisé au chapitre 8.

Plans

La carte est une collection très courante qui est utilisée dans pratiquement toutes les applications et offres la possibilité d'associer un objet clé à un objet de valeur arbitraire. Les différents sous-jacents les implémentations devraient utiliser des techniques de hachage rapide pour optimiser l'accès direct aux clés.

Il y a une grande flexibilité avec les types de carte dans JPA, étant donné que les clés et les valeurs peut être n'importe quelle combinaison d'entités, de types de base et d'éléments incorporables. Permutation des trois types dans les deux positions de clé et de valeur rend neuf types de map distincts. Nous donnons détaillé explications des combinaisons les plus courantes dans les sections suivantes.

Épisode 186

Clés et valeurs

Bien que les types de base, les types intégrables ou les types d'entités puissent être des clés de mappage, rappelez-vous que s'ils jouent le rôle de clé, ils doivent suivre les règles de base des clés. Ils doivent être comparables et répondent de manière appropriée à la méthode hashCode () et aux equals () méthode si nécessaire [2]. Ils doivent également être uniques, au moins dans le domaine d'un instance de collection particulière, de sorte que les valeurs ne soient pas perdues ou écrasées en mémoire. Clés ne doivent pas être modifiés, ni plus précisément les parties de l'objet clé utilisées dans les méthodes hashCode () et equals () ne doivent pas être modifiées tant que l'objet agit comme une clé dans une carte.

En général, pour conserver un java.util.Map ayant des clés / valeurs de types Java de base, le l'annotation @ElementCollection peut être utilisée et elle sera placée sur la référence cartographique dans la classe d'entité. Notez que l'utilisation de @ElementCollection est très similaire à mappage java.util.Collection.

Lorsque les clés sont de type basique ou intégrable, elles sont stockées directement dans la table en cours visé. Selon le type de mappage, il peut s'agir de la table d'entité cible, de la table de jointure, ou table de collecte. Cependant, lorsque les clés sont des entités, seule la clé étrangère est stockée dans le table car les entités sont stockées dans leur propre table et leur identité dans la base de données doit être préservé.

C'est toujours le type de l'objet de valeur dans la carte qui détermine le type de la cartographie doit être utilisée. Si les valeurs sont des entités, la carte doit être mappée comme une relation à plusieurs ou plusieurs à plusieurs, alors que si les valeurs de la carte sont soit les types intégrables ou de base, la carte est mappée en tant que collection d'éléments.

Même si les touches Map n'affectent pas le type de mappage, elles nécessitent toujours des annotations, en plus des annotations de relation ou de collection d'éléments, pour indiquez la ou les colonnes dans lesquelles ils sont stockés. Ces annotations sont couvertes dans le différents cas d'utilisation dans les sections suivantes.

Clé par type de base

Nous avons mentionné dans les sections précédentes que les collections d'éléments de types de base sont stockées dans des tables de collection, et les clés de base sont stockées dans les tables référencées par le cartographie. Si le mappage est une collection d'éléments indexés par un type de base, les clés seront stockées dans la même table de collection dans laquelle les valeurs de la carte sont stockées. De même, s'il s'agit d'un

[2] Consultez Javadoc pour java.util.Map pour plus de détails.

Épisode 187

relation un-à-plusieurs, et la clé étrangère est dans la table d'entité cible, les clés être dans la table des entités cibles. Si le mappage de relations utilise une table de jointure, les clés être dans la table de jointure.

Pour montrer le cas de la table de collection, examinons un exemple de collection d'éléments qui mappe les numéros de téléphone d'un employé. Si nous utilisons une carte, nous pouvons saisir le téléphone type de numéro et enregistrez le numéro de téléphone comme valeur. Ainsi, la clé de chaque entrée de la carte être l'une des catégories Maison, Travail ou Mobile, sous forme de chaîne, et la valeur sera le téléphone associé nombre Chaîne. Référencement5-7 montre le code de mappage de la collection d'éléments.

Liste 5-7. Collection d'éléments de chaînes avec des clés de chaîne

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;

    @ElementCollection
    @CollectionTable (nom = "EMP_PHONE")
    @MapKeyColumn (nom = "PHONE_TYPE")
    @Column (nom = "PHONE_NUM")
    private Map <String, String> phoneNumbers;
    // ...
}
```

Les annotations @ElementCollection et @CollectionTable ne sont pas nouvelles, et le Listing 5-3 a montré que nous pouvons utiliser l'annotation @Column pour remplacer le nom de la colonne qui stocke les valeurs dans la collection. Ici nous faisons la même chose chose, sauf que nous remplaçons la colonne dans laquelle les valeurs de la carte seraient stockées au lieu des éléments d'une collection générique.

La seule nouvelle annotation est @MapKeyColumn, qui est utilisée pour indiquer la colonne dans la table de collection qui stocke la clé de base. Lorsque l'annotation n'est pas spécifiée, le la clé est stockée dans une colonne nommée d'après l'attribut de collection mappé, ajoutée avec le suffixe _KEY. Dans le Listing 5-7 , si nous n'avions pas spécifié @MapKeyColumn, la règle par défaut aurait entraîné le mappage de la clé vers la colonne PHONENUMBERS_KEY dans le Table de collecte EMP_PHONE.

Épisode 188

Remarque Par défaut, les conventions de dénomination spécifiques à Jpa sont utilisées pour le mappage et nous pouvons le personnaliser en utilisant @MapKeyColumn, @CollectionTable et Annotations @Column sur le champ / la propriété de la carte.

Les valeurs de numéro de téléphone peuvent être dupliquées dans la table de collecte (par exemple, plusieurs employés vivant au même domicile et ayant le même numéro de téléphone), La colonne PHONE_NUM ne sera évidemment pas unique dans la table. Les types de numéros de téléphone doivent être uniques uniquement dans une instance Map ou Employee donnée, donc le PHONE_TYPE la colonne ne sera pas non plus la clé primaire. En fait, parce que les types de base n'ont pas d'identité, et dans certains cas, les mêmes entrées clé-valeur peuvent être dupliquées dans plusieurs sources entités, les colonnes clé-valeur ne peuvent pas être les colonnes de clé primaire à elles seules. Unique les tuples dans la table de collection doivent être la combinaison de la colonne clé et de l'élément étranger colonne clé qui fait référence à l'instance d'entité source. La figure 5-4 montre le résultat table de collection, ainsi que la table d'entité EMPLOYEE source à laquelle elle fait référence. Vous pouvez voir la contrainte de clé primaire sur les colonnes EMPLOYEE_ID et PHONE_TYPE.

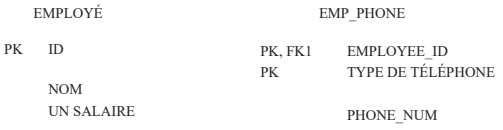

Figure 5-4. *Table d'entités EMPLOYEE et table de collecte EMP_PHONE*

Nous devrions vraiment améliorer notre modèle, car l'utilisation d'une clé String pour stocker

quelque chose qui est contraint d'être l'une des trois valeurs seulement (Domicile, Mobile ou Travail)
ce n'est pas super style. Une amélioration appropriée serait d'utiliser un type énuméré
au lieu de String. Nous pouvons définir notre type énuméré comme suit:

public enum PhoneType {Domicile, Mobile, Travail}

Maintenant, nous avons les options valides en tant que constantes énumérées, et il n'y a aucune chance de erreur de frappe ou types de téléphone non valides. Cependant, il y a une autre amélioration à envisager. Si nous voulons nous protéger des modifications futures de l'énumération les valeurs de type, soit en réorganisant les valeurs existantes, soit en insérant des valeurs supplémentaires, nous devrions remplacer la façon dont la valeur est stockée dans la base de données. Au lieu de se fier à la valeur par défaut approche de stockage de la valeur ordinale de l'élément énuméré, nous voulons stocker le

Épisode 189

Valeur de chaîne, nous obtenons donc le meilleur des deux mondes. La colonne contiendra des valeurs qui correspondent aux paramètres de type de téléphone d'une manière lisible par l'homme, et la carte Java aura un clé fortement tapée.

La manière habituelle de remplacer la stratégie de stockage pour un type énuméré consiste à utiliser le @ Annotation énumérée. Cependant, si nous devions mettre @Enumerated sur notre attribut Map, il s'appliquerait aux valeurs de la collection d'éléments, pas aux clés. C'est pourquoi il y a une annotation spéciale @MapKeyEnumerated (voir l'extrait 5-8). Il y a aussi un équivalent @MapKeyTemporal pour spécifier le type temporel lorsque la clé est de type java.util.Date. Les deux @MapKeyEnumerated et @MapKeyTemporal sont applicables aux clés qui sont de base type, qu'il s'agisse d'une collection d'éléments ou d'une relation.

Liste 5-8. Collection d'éléments de chaînes avec des clés de type énumérées

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;

    @ElementCollection
    @CollectionTable (nom = "EMP_PHONE")
    @MapKeyEnumerated (EnumType.STRING)
    @MapKeyColumn (nom = "PHONE_TYPE")
    @Column (nom = "PHONE_NUM")
    carte privée <PhoneType, String> phoneNumbers;
    // ...
}
```

Référencement 5-4 avaient une relation un-à-plusieurs qui utilisait une liste pour contenir tous les employés dans un service donné. Supposons que nous la changions pour utiliser une carte et garder une trace dont l'employé travaille dans un bureau ou une cabine donné. En tapant sur la cellule nombre (qui peut également contenir des lettres, nous les représentons donc sous forme de chaîne), nous pouvons trouver facilement quel employé travaille dans cette cabine. Parce que c'est un bidirectionnel-relation à plusieurs, il sera mappé en tant que clé étrangère vers DEPARTMENT dans EMPLOYEE table. Les clés de numéro de cellule seront stockées dans une colonne supplémentaire dans le EMPLOYÉ table, chacun stocké dans la ligne correspondant à l'employé associé à ce cabine. Le listing 5-9 montre le mappage un-à-plusieurs.

Épisode 190

Liste 5-9. Relation un-à-plusieurs à l'aide d'une carte avec une clé de chaîne

```
@Entité
Département de classe publique {
    @Id id int privé;

    @OneToMany (mappedBy = "département")
    @MapKeyColumn (nom = "CUB_ID")
    private Map <String, Employee> EmployeesByCubicle;
    // ...
}
```

Et si un employé pouvait partager son temps entre plusieurs départements? Nous serions doivent changer notre modèle en une relation plusieurs-à-plusieurs et utiliser une table de jointure. le @MapKeyColumn sera stocké dans la table de jointure qui référence les deux entités. le la relation est mappée dans l'extrait 5-10.

Liste 5-10. Relation plusieurs à plusieurs à l'aide d'une carte avec des clés de chaîne

```
@Entité
Département de classe publique {
    @Id id int privé;
    nom de chaîne privé;

    @Plusieurs à plusieurs
    @JoinTable (nom = "DEPT_EMP",
        joinColumns = @ JoinColumn (nom = "DEPT_ID"),
        inverseJoinColumns = @ JoinColumn (nom = "EMP_ID"))
    @MapKeyColumn (nom = "CUB_ID")
    private Map <String, Employee> EmployeesByCubicle;
    // ...
}
```

Si nous n'avions pas remplacé la colonne clé avec @MapKeyColumn, cela aurait été défini par défaut comme le nom de l'attribut de collection suffixé par _KEY. Cela aurait a produit une colonne EMPLOYEESBYCUBICLE_KEY d'apparence épouvantable dans la table de jointure, qui n'est pas seulement moche à lire, mais n'indique pas réellement quelle est la clé. Figure 5-5 montre les tableaux résultants.

Épisode 191

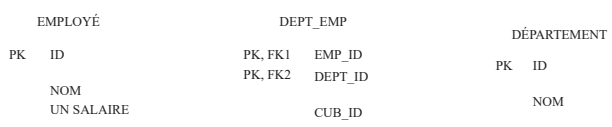

Figure 5-5. *Tables d'entités EMPLOYEE et DEPARTMENT et DEPT_EMP table de jointure*

Remarque Vous ne pouvez utiliser une carte que sur un seul côté d'une relation plusieurs-à-plusieurs; il ne fait aucune différence de quel côté.

Clé par attribut d'entité

Lorsqu'un ensemble d'entités de relations un-à-plusieurs ou plusieurs-à-plusieurs est représenté en tant que Map, il est le plus souvent indexé par un attribut du type d'entité cible. Clé par L'attribut d'entité est en fait un cas particulier de saisie par type de base où le mappage est un relation, et le type de base de la clé est le type de l'attribut (sur lequel nous saisissons) dans l'entité cible. Lorsque ce cas courant se produit, l'annotation @MapKey peut être utilisée pour désigner l'attribut de l'entité cible sur laquelle la saisie est effectuée.

Si chaque service garde une trace des employés qu'il contient, comme dans notre exemple précédent Référencement 5-4, nous pourrions utiliser une carte et une clé sur l'identifiant de l'employé pour une recherche rapide des employés. Le mappage de département mis à jour est affiché dans la liste 5-11.

Annonce 5-11. Relation un-à-plusieurs indexée par attribut d'entité

```
@Entité
Département de classe publique {
    // ...
    @OneToMany (mappedBy = "département")
    @MapKey (nom = "id")
    personnel Map <Integer, Employee> privé;
    // ...
}
```

Épisode 192

L'attribut id de Employee est également l'identifiant ou l'attribut de clé primaire, et il il s'avère que la saisie de l'identifiant est le cas le plus courant de tous. C'est si courant que lorsqu'aucun nom n'est spécifié, les entités seront par défaut indexées par leur identifiant attribut 1. Lorsque l'attribut identificateur est défini par défaut et n'est pas explicitement répertorié, nous avons besoin pour connaître le type d'identifiant afin de pouvoir spécifier correctement le premier paramètre de type de la Map lors de l'utilisation d'une carte paramétrée.

L'une des raisons pour lesquelles l'attribut identifier est utilisé pour la clé est qu'il correspond à la critères clés bien. Il répond aux méthodes de comparaison nécessaires, hashCode () et equals (), et il est garanti qu'il est unique.

Si un autre attribut est utilisé comme clé, il doit également être unique, bien qu'il ne soit pas absolument nécessaire qu'il soit unique dans tout le domaine de ce type d'entité. C'est vraiment doit être unique uniquement dans le cadre de la relation. Par exemple, nous pourrions saisir sur le nom de l'employé tant que nous nous sommes assurés que le nom serait unique dans n'importe quel département.

Dans la section précédente, nous avons indiqué qu'une clé de base est stockée dans la table référencée par la cartographie. Le cas particulier de la saisie par attribut d'entité est une exception à cette règle en ce qu'aucune colonne supplémentaire n'est nécessaire pour stocker la clé. Il est déjà stocké dans le cadre du entité. C'est pourquoi l'annotation @MapKeyColumn n'est jamais utilisée lors de la saisie sur une entité attribut. Un fournisseur peut facilement créer le contenu d'une carte de relations un-à-plusieurs en chargement des entités associées à l'entité source et extraction de l'attribut étant saisi à partir de chacune des entités chargées. Aucune colonne supplémentaire ne doit être lue ou des jointures supplémentaires effectuées.

Clé par type intégrable

Utiliser des éléments intégrés comme clés n'est pas quelque chose que vous devriez rencontrer très souvent. En fait, si vous envisagez de le faire, vous devriez probablement réfléchir à deux fois avant procéder. Ce n'est pas parce que c'est possible que c'est une bonne idée.

Le problème avec les éléments intégrés est qu'ils ne sont pas des entités à part entière. Ils ne sont pas interrogeables dans le sens où ils ne peuvent être découverts ou renvoyés que sous forme d'agrégat

partie de leurs entités englobantes. Bien que cela ne semble pas être une limitation très sévère au début, cela devient souvent un problème plus tard dans le cycle de développement.

[3] L'annotation @MapKey est cependant toujours requise; sinon, les valeurs par défaut de @MapKeyColumn appliquer.

Épisode 193

L'identité des éléments incorporables n'est pas définie en général, mais lorsqu'ils sont utilisés comme clés dans une carte, il doit y avoir une certaine notion d'unicité définie, applicable au moins dans la carte donnée. Cela signifie que la contrainte d'unicité, au moins logiquement, est sur le combinaison des attributs incorporés et de la colonne de clé étrangère à l'entité source.

Les types de clés intégrables sont similaires aux types de clés de base en ce sens qu'ils sont également stockés dans la table référencée par le mappage, mais avec les types intégrables, il existe plusieurs attributs à stocker, pas seulement une valeur. Il en résulte que plusieurs colonnes contribuent à la clé primaire.

Partage de mappages de clés intégrables avec des valeurs

L'exemple de code dans la liste 5-11 montraient une relation bidirectionnelle un-à-plusieurs de Département à Employé qui a été saisi par l'attribut id de Employé. Et si nous avions voulait saisir plusieurs attributs de l'employé? Par exemple, il pourrait être souhaitable pour rechercher les employés par nom dans la carte, en supposant que le nom est unique dans un département donné. Si le nom a été divisé en deux attributs, un pour le prénom et un pour le nom de famille, comme indiqué dans l'extrait 5-12, alors nous aurions besoin d'un objet séparé pour les combiner et agir comme objet clé dans la carte. Un type intégrable, tel que EmployeeName dans la liste 5-13, peut être utilisé à cette fin. Avoir un EmployeeName le type embeddable fournit également une classe utile pour passer le nom complet encapsulé autour du système.

Annonce 5-12. Entité des employés

```
@Entité
Employé de classe publique {
    @Id id int privé;
    @Column (nom = "F_NAME")
    private String firstName;
    @Column (nom = "L_NAME")
    private String lastName;
    long salaire privé;
    // ...
}
```


Épisode 194

Annonce 5-13. EmployeeName intégrable avec des mappages en lecture seule

```
@Embeddable
public class EmployeeName {
    @Column (name = "F_NAME", insertable = false, updatable = false)
    private String first_Name;
    @Column (name = "L_NAME", insertable = false, updatable = false)
    private String last_Name;
    // ...
}
```

Parce que la relation bidirectionnelle un-à-plusieurs d'un service à l'autre est stockée dans la table d'entités cible, la clé d'objet intégrable doit également y être stockée. Cependant, il serait redondant que les deux composants de nom soient stockés deux fois dans chaque ligne, une fois pour les attributs firstName et lastName de Employee et une fois pour le Attributs first_Name et last_Name de l'objet clé EmployeeName. Avec un peu de malin mappage, nous pouvons simplement réutiliser les deux colonnes mappées aux attributs des employés et mapper les en lecture seule dans la clé (paramètre insérable et modifiable sur false). C'est pourquoi dans Référencement 5-13 nous mappons les attributs first_Name et last_Name aux mêmes colonnes que les attributs firstName et lastName de Employee. Du point de vue du Ministère, la relation dans la liste 5-14 ne change pas grand-chose de la liste5-11, sauf que le La carte est saisie par EmployeeName au lieu de par Integer et @MapKey n'est pas utilisé car la clé est un attribut intégrable et non un attribut de Employee.

Liste 5-14. Relation un-à-plusieurs définie par Embeddable

```
@Entité
Département de classe publique {
    // ...
    @OneToMany (mappedBy = "département")
    Carte privée <EmployeeName, Employee> employés;
    // ...
}
```

Remplacement des attributs intégrables

Une autre option de modélisation consiste à combiner les deux colonnes de nom dans l'Employé entité et définissez un attribut incorporé de type EmployeeName, comme indiqué dans la liste 5-15 .

Épisode 195

Liste 5-15. Entité d'employé avec attribut intégré

```
@Entité
Employé de classe publique {
    @Id id int privé;

    @Embedded
    nom privé EmployeeName;
    long salaire privé;
    // ...
}
```

Cette fois, nous ne partageons pas de colonnes, nous devons donc nous assurer que les mappages dans EmployeeName ne sont plus en lecture seule, sinon le nom ne sera jamais écrit dans le base de données. L'embeddable EmployeeName mis à jour se trouve dans le Listing 5-16.

Annonce 5-16. EmployeeName intégrable

```
@Embeddable
```

```
public class EmployeeName {
    @Column (nom = "F_NAME")
    private String first_Name;
    @Column (nom = "L_NAME")
    private String last_Name;
    // ...
}
```

À des fins d'illustration, revenons au modèle plusieurs-à-plusieurs décrit dans le Listing 5-10 , sauf que nous allons saisir le EmployeeName intégrable au lieu du id de cellule. Même si les attributs EmployeeName sont stockés dans la table EMPLOYEE pour chaque employé, les clés de la carte doivent toujours être stockées dans la table de jointure DEPT_EMP. Cette est le résultat du fait que la clé est un type intégrable. Clé par soit un seul attribut du entité ou par un type de base atténuerait ce scénario de données dénormalisées.

Par défaut, les attributs clés seraient mappés aux noms de colonne de la mappages définis dans EmployeeName, mais si la table de jointure existe déjà et les colonnes de la table de jointure n'ont pas ces noms, les noms doivent être remplacés. Référencement 5-17 montre comment les mappages d'attributs intégrables de la clé Map peuvent être remplacé par ce qu'ils sont définis comme étant dans la classe intégrable.

Épisode 196

Annonce 5-17. Carte plusieurs-à-plusieurs identifiée par type intégrable avec remplacement

```
@Entité
Département de classe publique {
    @Id id int privé;

    @Plusieurs à plusieurs
    @JoinTable (nom = "DEPT_EMP",
            joinColumns = @ JoinColumn (nom = "DEPT_ID"),
          inverseJoinColumns = @ JoinColumn (nom = "EMP_ID"))
    @AttributeOverrides ({
        @AttributeOverride (
            name = "first_Name",
                colonne = @ Colonne (nom = "EMP_FNAME")),
        @AttributeOverride (
            name = "last_Name",
                colonne = @ Colonne (nom = "EMP_LNAME"))
    })
    Carte privée <EmployeeName, Employee> employés;
    // ...
}
```

Les tableaux pour le mappage sont présentés dans la figure 5-6, avec les attributs intégrables mappé à la table de jointure pour l'état de clé et dans la table EMPLOYEE pour l'employé Etat. Si le mappage avait été une collection d'éléments, les attributs incorporables être stocké dans une table de collection au lieu d'une table de jointure.

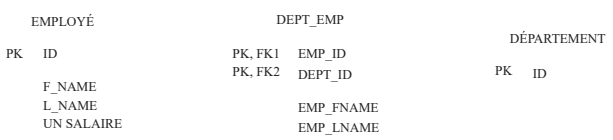

Figure 5-6. *Tables d'entités DEPARTMENT et EMPLOYEE et jointure DEPT_EMP table*

Épisode 197

Comme vous pouvez le voir sur le Listing 5-17, les valeurs par défaut du mappage pour la clé sont remplacé par l'utilisation de @AttributeOverride. Si au lieu d'un plusieurs-à-plusieurs relation nous avions un @ElementCollection d'un type intégrable dans une carte, nous devrait faire la différence entre la clé et la valeur. Nous ferions cela en préfixant le nom de l'attribut avec la clé. ou de la valeur., selon lequel des types intégrables nous dominaient. Une collection d'éléments de types EmployeeInfo incorporés, avec le même les remplacements de clé comme ceux de la relation du Listing 5-17 , utiliseraient la clé suivante préfixes:

```
@ElementCollection
@AttributeOverrides ({
    @AttributeOverride (nom = "key.first_Name",
                        colonne = @ Colonne (nom = "EMP_FNAME")),
    @AttributeOverride (nom = "key.last_Name",
                        colonne = @ Colonne (nom = "EMP_LNAME"))
})
Carte privée <NomEmployé, InfoEmployé> empInfos;
```

Clé par entité

Vous pourriez être réticent à utiliser des entités comme clés, car l'intuition pourrait vous amener à penser à il s'agit d'une option plus gourmande en ressources, avec des coûts de charge et de gestion plus élevés. Tandis que cela peut être vrai dans certains cas, ce n'est pas nécessairement toujours le cas. Assez souvent l'entité que vous envisagent que la saisie soit déjà en mémoire, ou nécessaire de toute façon, et la saisie accède simplement à une instance mise en cache ou entraîne le chargement de l'instance pour une utilisation ultérieure.

L'un des avantages de la saisie par type d'entité est que les instances d'entité sont uniques au monde (au sein de l'unité de persistance) donc il n'y aura pas de problèmes d'identité à traiter à travers différentes relations ou collections. Un corollaire de la propriété d'identité de base d'entités est que seule une clé étrangère doit être stockée dans la table mappée, conduisant à un conception plus normalisée et schéma de stockage des données.

Comme pour les autres types de clés Map, la clé (dans ce cas, une clé étrangère de l'entité en cours de saisie) seront stockées dans la table référencée par le mappage.

Rappelons que le terme utilisé par JPA pour représenter une colonne de clé étrangère est join column, et nous utilisons des colonnes de jointure dans les relations plusieurs-à-un et un-à-un, ainsi que dans joindre des tables de relations valorisées par collection. Nous avons maintenant une situation similaire, sauf qu'au lieu de faire référence à la cible d'une relation, notre colonne de jointure fait référence au clé d'entité dans une entrée de carte. Pour différencier les colonnes de jointure qui pointent vers les clés de mappage de

Épisode 198

celles utilisées dans les relations, une annotation @MapKeyJoinColumn distincte a été créée. Cette L'annotation est utilisée pour remplacer les valeurs par défaut de la colonne de jointure pour une clé d'entité. Quand ce n'est pas spécifié, la colonne de jointure aura le même nom de colonne par défaut que les clés de base (le nom de l'attribut de relation ou de collection d'éléments, ajouté avec la chaîne _KEY).

Pour illustrer le cas d'une entité utilisée comme clé, on peut ajouter la notion de l'ancienneté d'un employé au sein d'un département donné. Nous voulons avoir un lâche association entre un employé et son ancienneté, et l'ancienneté doit être locale à un Département. En définissant une carte de collection d'éléments dans Department, avec l'ancienneté comme les valeurs et les entités Employé comme clés, l'ancienneté de tout Employé dans un Department peut être recherché en utilisant l'instance Employee comme clé. L'ancienneté est stocké dans une table de collecte, et si un employé change de service, aucun des autres Les objets des employés doivent changer. L'indirection de la table de collecte, et le fait que les liens entre le ministère, l'employé et la valeur d'ancienneté sont tous maintenus en vertu de la carte, fournissent juste le bon niveau de couplage. Seulement le les entrées de la table de collection devraient être mises à jour.

Référencement 5-18 montre le mappage de la collection d'éléments, la colonne de jointure étant remplacé à l'aide de l'annotation @MapKeyJoinColumn et de la colonne de valeur de la carte remplacé à l'aide de l'annotation standard @Column.

Liste 5-18. Mappe de collection d'éléments indexée par EntityType

```
@Entité
Département de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    // ...
    @ElementCollection
    @CollectionTable (nom = "EMP_SENIORITY")
    @MapKeyJoinColumn (nom = "EMP_ID")
    @Column (nom = "SENIORITY")
    Carte privée <Employé, Entier> ancienneté;
    // ...
}
```

Figure 5-7 montre que la table de collection n'est rien de plus que les valeurs du Mapper (l'ancienneté) avec une clé étrangère vers la table d'entité source Department et une autre clé étrangère à la table de clé d'entité Employee.

Épisode 199

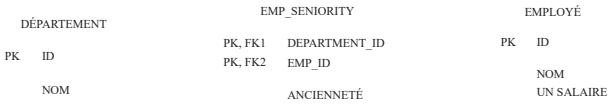

Figure 5-7. *Tables d'entités DEPARTMENT et EMPLOYEE avec EMP_SENIORITY table de collecte*

Cartes non typées

Si nous ne voulions pas (ou n'avons pas pu) utiliser la version de paramètre typé de Map <KeyType, ValueType>, nous le définirions en utilisant le style non paramétré de Map montré dans le listing 5-19.

Annonce 5-19. Relation un-à-plusieurs à l'aide d'une carte non paramétrée

```
@Entité
Département de classe publique {
    // ...
    @OneToMany (targetEntity = Employee.class, mappedBy = "département")
    @Carte clé
    les employés privés de Map;
```

```
    // ...
}
```

L'élément targetEntity indique uniquement le type de la valeur Map. Bien sûr si la carte contient une collection d'éléments et non une relation, l'élément targetClass de @ElementCollection est utilisé pour indiquer le type de valeur de la carte.

Dans la liste 5-19 , le type de la clé Map peut être facilement déduit car le mappage est une relation d'entité. La valeur par défaut de @MapKey est d'utiliser l'attribut identificateur, id, de type int ou Integer. Si @MapKey avait été spécifié et dicté que la clé soit un attribut qui n'était pas un attribut identifiant, le type aurait quand même été déductible car le Les attributs d'entité sont tous mappés avec des types connus. Cependant, si la clé n'est pas un attribut de l'entité cible, @MapKeyClass peut être utilisé à la place de @MapKey. Il indique le type de la classe clé lorsque la carte n'est pas définie de manière typée à l'aide de génériques. Il est également utilisé lorsque la carte fait référence à une collection d'éléments au lieu d'une relation car de base ou les types intégrables n'ont pas d'attributs d'identifiant, et les types de base n'ont même pas les attributs.

Épisode 200

Pour illustrer comment @MapKeyClass est utilisé, prenons la collection d'éléments dans la liste 5-7 et supposons qu'il ne définit pas les paramètres de type sur la carte. La saisie est remplie par l'utilisation de l'annotation @MapKeyClass et de l'élément targetClass dans @ElementCollection, comme indiqué dans la liste 5-20 .

Liste 5-20. Collection d'éléments non typés de chaînes avec des clés de chaîne

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;

    @ElementCollection (targetClass = String.class)
    @CollectionTable (nom = "EMP_PHONE")
    @MapKeyColumn (nom = "PHONE_TYPE")
    @MapKeyClass (String.class)
    @Column (nom = "PHONE_NUM")
    carte privée phoneNumbers;
    // ...
}
```

L'annotation @MapKeyClass doit être utilisée chaque fois que la classe de clé ne peut pas être déduit de la définition d'attribut ou des autres métadonnées de mappage.

Règles pour les cartes

L'apprentissage des différentes variantes de la carte peut être assez déroutant étant donné que vous pouvez choisissez l'un des trois types de clés et trois types de valeur les types. Voici quelques-unes des règles de base d'utilisation d'une carte.

- Utilisez les éléments @MapKeyClass et targetEntity / targetClass des mappages de relation et de collection d'éléments pour spécifier le classes lorsqu'une Map non typée est utilisée.
- Utilisez @MapKey avec une carte de relation un-à-plusieurs ou plusieurs-à-plusieurs qui est indexé sur un attribut de l'entité cible.

Épisode 201

- Utilisez @MapKeyJoinColumn pour remplacer la colonne de jointure de la clé d'entité.
- Utilisez @Column pour remplacer la colonne stockant les valeurs d'un élément collection de types de base.
- Utilisez @MapKeyColumn pour remplacer la colonne stockant les clés lorsque saisie par un type de base.
- Utilisez @MapKeyTemporal et @MapKeyEnumerated si vous avez besoin de qualifiez une clé de base de type temporel ou énuméré.
- Utilisez @AttributeOverride avec une clé. ou valeur. préfixe pour remplacer le colonne d'un type d'attribut intégrable qui est une clé de mappage ou une valeur, respectivement.

Le tableau 5-1 résume certains des différents aspects de l'utilisation d'une carte.

Épisode 202

Annotation de valeur @Colonne ... @Colonne ... Mappé par ... Attribut ... Remplacer ... entité

Annotation clé @MapKey ... @MapKeyColumn ... @MapKeyJoinColumn

p une ga M dans pp une ry de M une m Somme

Cartographie @ElementCollection @OneToMany, @ManyToMany

ble 5-1. Ta

Carte Carte <Basique, Basique> ... Carte <Embarquable, Entité> ... Carte <Entité, Entité>

Épisode 203

Doublons

Lorsque nous avons discuté de l'interface Set, nous avons mentionné qu'elle était idéale pour éviter les doublons. Ce que nous voulions dire, c'est que le type de données Set en Java ne permet pas leur. Nous n'avons vraiment rien dit sur les doublons dans la base de données. En fait, le JPA la spécification ne dit rien sur l'autorisation des doublons dans les collections, dans la base de données ou en mémoire, et dans la plupart des cas, ils ne seront pas pris en charge. À comprendre pourquoi la prise en charge des doublons est difficile, passons au modèle de données et découvrez certains des détails sanglants. Vous préférerez peut-être ignorer cette section si des doublons sont pas intéressant pour vous. Cependant, lisez la suite si vous êtes dans une situation où un l'application peut entrer dans votre dos et insérer un enregistrement en double, par exemple.

Notez que le fait d'avoir une clé étrangère unique pour la table cible signifie que notre collection, comme par exemple List, ne peut pas avoir de doublons.

L'interface Collection est très générale et permet une multitude de Collection sous-interfaces et classes d'implémentation. Il est donc très doux dans sa spécification de savoir si les doublons sont autorisés, ce qui permet à la sous-interface ou à la classe d'implémentation de décider quel comportement convient le mieux à ce type de collection.

Pour une collection qui autorise les doublons, la valeur de la collection les mappages de relations utilisent soit une clé étrangère dans la table d'entité cible, soit une table de jointure. Le premier cas sera toujours un mappage un-à-plusieurs, et dans ce cas il n'y aura que une ligne pour l'entité cible et une seule colonne dans cette ligne pour contenir la clé étrangère à l'entité source. Cela ne laisse aucun moyen de saisir le fait que l'entité cible est dans le

collection plusieurs fois.

Dans le cas d'une table de jointure, chaque ligne stocke une colonne de jointure à l'entité source et une jointure colonne à l'entité cible, et la clé primaire de la table de jointure est composée du combinaison des deux. Seules les lignes en double dans ce modèle peuvent lier plusieurs instances de la cible vers la même source et les lignes en double dans une base de données relationnelle sont fortement mal vu.

Une collection d'éléments est dans une situation similaire, sauf qu'au lieu d'une clé étrangère vers l'entité cible il y a une ou plusieurs colonnes dans la table de collection stockant le basique ou des valeurs intégrables. Ces colonnes se combinent simplement avec la clé étrangère de la source entité pour constituer la clé primaire, et une fois de plus des lignes en double seraient nécessaires pour ont des valeurs en double dans une collection.

Épisode 204

La liste constamment ordonnée est un peu différente, cependant, car elle ajoute une commande colonne au mélange. Si la colonne de commande devait être incluse dans le cadre du clé, plusieurs entrées de relation peuvent exister dans la liste - chacune de leurs lignes respectives ayant potentiellement les mêmes données de valeur d'élément et la même référence de clé étrangère, mais différant uniquement par la valeur de la colonne de commande. Ainsi, l'unicité d'une ligne est identifiée non uniquement par les objets source et cible mais aussi par sa position dans la liste.

Dans le cas de la clé étrangère dans la table cible, ce serait en effet une mauvaise pratique inclure la colonne de commande dans la clé primaire d'une table d'entité, donc nous n'explorerons même pas cela en option. Cependant, lorsqu'une table de jointure ou une table de collection est utilisée, elle est parfaitement chose raisonnable à faire, permettant d'insérer des valeurs en double dans un ordre permanent Liste. Cela n'est possible, cependant, que si le fournisseur inclut la colonne de commande dans le clé primaire de la table ou vous donne la possibilité de la configurer de cette manière.

Avant de vous réjouir que votre fournisseur vous autorise à stocker des doublons, sachez qu'il y a un prix à payer. Ceci peut être vu dans l'exemple d'échange de l'ordre de deux éléments de la liste. Si la colonne de commande n'était qu'une colonne normale, elle ne être trop compliqué pour que le fournisseur optimise les opérations de la base de données simplement mettre à jour les colonnes de commande des deux enregistrements avec les valeurs correctes. Cependant, si le la colonne de commande fait partie de la clé primaire, ce que vous dites vraiment, c'est que la commande de l'objet contenu dans la liste fait partie intégrante de la relation entre le contenant et contenu des objets. Attribuer un nouvel ordre à l'objet contenu n'est pas modifier un aspect de la relation, mais détruisant efficacement la relation et créer un nouveau avec un ordre différent. Cela signifie supprimer les deux anciennes lignes et en créant deux nouveaux.

Une carte a des clés qui doivent être uniques, donc les clés en double n'ont clairement aucun sens. Il est similaire à une liste en ce qui concerne ses valeurs, car elle a une colonne clé qui peut faire partie de la clé primaire. Encore une fois, le cas de la clé étrangère dans la cible table ne permet pas à plusieurs clés de pointer vers la même entité, donc un-à-plusieurs les relations qui sont mappées de cette façon ne permettent tout simplement pas à la même entité d'être mappé à plusieurs clés dans la même carte.

Pour les tables de jointure et les tables de collection, les doublons ne seront possibles dans la carte que si est saisi par autre chose qu'un attribut de l'entité, ou intégrable, et la clé la colonne est incluse dans la clé primaire. Le compromis dans le cas de la carte est similaire à celui dont nous avons discuté avec List, sauf que le prix de l'autorisation des doublons dans une carte est payé lorsque vous souhaitez réaffecter une clé existante à une valeur différente.

Épisode 205

Valeurs nulles

Il est probablement encore moins courant d'insérer des valeurs nulles dans une collection que d'en avoir doublons. C'est l'une des raisons pour lesquelles la spécification JPA n'est pas particulièrement claire sur ce se produit lorsque vous insérez null dans une collection. Comme pour les doublons, les cas sont un peu complexes et nécessitent une considération individuelle.

Les interfaces Set, List et Map rejoignent l'interface Collection en étant générales assez pour être insouciant quand il s'agit de spécifier ce qui se passe quand null est inséré. Ils délèguent simplement la décision à la mise en œuvre, donc un La classe d'implémentation peut choisir de prendre en charge l'insertion de valeurs nulles ou de lancer un exception. JPA ne fait pas mieux; il finit par revenir au proxy du fournisseur particulier implémentation pour autoriser null ou lever une NullPointerException lorsque null est ajouté. Notez que vous ne pouvez pas rendre votre interface de collection autoriser null simplement en l'initialisant avec une instance d'implémentation, telle que HashSet, qui autorise null. Le fournisseur va remplacer l'instance par l'une de ses propres classes d'implémentation la prochaine fois que l'objet devient gérée, et la nouvelle classe d'implémentation peut autoriser ou non null valeurs.

Pour qu'une valeur nulle existe dans la base de données, la ou les colonnes de valeur doivent être nullable. C'est la partie évidente, mais le corollaire pourrait être moins évident, si un peu répétitif. Il affirme une fois de plus que seules les relations et les collections d'éléments qui utilisent une table de jointure ou une table de collection peut avoir des valeurs nulles dans la collection. La preuve est laissée pour vous devez comprendre, mais (à titre indicatif) essayez de créer une entité qui a toutes les valeurs nulles, y compris l'identifiant, sans génération d'identifiant.

Il existe une limitation supplémentaire sur les valeurs nulles en ce qui concerne les collections d'éléments d'objets intégrables. Références d'entité dans une table de jointure ou des collections d'éléments de base les types dans une table de collection sont des valeurs ou des références à une seule colonne. Le problème dans le le cas intégrable est que si une combinaison de colonnes mappées à un élément all null, il n'y a aucun moyen pour le fournisseur de savoir si cela signifie une valeur nulle ou un objet intégrable vide plein de valeurs nulles. Les fournisseurs peuvent supposer qu'il s'agit d'un objet incorporé ou ils peuvent avoir une option contrôlable pour dicter si les valeurs nulles être traité d'une manière ou d'une autre.

Les cartes ne s'engagent pas non plus à autoriser les clés nulles, mais cela ne convient vraiment pas très bien avec le modèle des colonnes clés étant des champs clés primaires. La plupart des bases de données ne autoriser l'un des champs de clé primaire à être nullable, et nous ne le recommanderions même pas pour l'étrange qui le fait.

Épisode 206

Les meilleures pratiques

Avec toutes les options et possibilités qui ont émergé, nous serions vraiment négligents si nous n'offrons pas au moins une certaine mesure d'orientation au voyageur solitaire des collections. Bien sûr, la raison pour laquelle il y a tant d'options est qu'il y en a tellement différents cas à résoudre, il n'est donc pas vraiment approprié de proposer des règles strictes et rapides. Cependant, nous espérons que certaines directives générales vous aideront à choisir la bonne cartographie stratégie pour votre cas d'utilisation d'application spécifique.

- Lorsque vous utilisez une liste, ne supposez pas qu'elle est commandée automatiquement si vous n'avez pas spécifié de commande. L'ordre de liste peuvent être affectés par les résultats de la base de données, qui ne sont que partiellement

déterministe quant à leur ordre. Il n'y a pas
garantit qu'une telle commande sera la même sur plusieurs
exécutions.

- Il sera généralement possible de commander les objets par l'un des leurs
les attributs. L'utilisation de l'annotation @OrderBy sera toujours la meilleure
approche par rapport à une liste persistante qui doit maintenir
l'ordre des articles qu'il contient en mettant à jour une colonne de commande spécifique.
Utilisez la colonne de commande uniquement lorsqu'il est impossible de faire autrement.

- Les types de carte sont très utiles, mais ils peuvent être relativement compliqués à
configurer correctement. Une fois que vous atteignez ce stade, cependant, la modélisation
les capacités qu'ils offrent et le soutien des associations qui
peuvent être exploités en font des candidats idéaux pour divers types de
relations et collections d'éléments.

- Comme pour la liste, l'utilisation préférée et la plus efficace d'une carte est de
utiliser un attribut de l'objet cible comme clé, en créant une carte d'entités
saisie par un type d'attribut de base le plus courant et le plus utile. Ce sera
résolvent souvent la plupart des problèmes que vous rencontrez. Une carte des clés de base
et les valeurs peuvent être une configuration utile pour associer un élément de base
objet avec un autre.

- Évitez d'utiliser des objets incorporés dans une carte, en particulier comme clés, car
leur identité n'est généralement pas définie. Les intégrables en général devraient
être traité avec soin et utilisé uniquement lorsque cela est absolument nécessaire.

Épisode 207

- La prise en charge des valeurs dupliquées ou nulles dans les collections n'est pas garantie,
et n'est pas recommandé même lorsque cela est possible. Ils provoqueront certains
types d'opérations sur le type de collection pour être plus lentes et plus
base de données intensive, parfois équivalente à une combinaison de
suppression et insertion d'enregistrements au lieu de simples mises à jour.

Résumé

Dans ce chapitre, nous avons examiné plus en détail les différentes manières de mapper des collections
à la base de données. Nous avons regardé comment le contenu de la collection détermine comment elle est
mappé, et a noté qu'il existe de nombreuses options flexibles pour stocker différents types de
objets dans divers types de collections.

Nous avons montré que la différence entre les relations et les collections d'éléments était
si des entités ou des types de base / intégrables y étaient stockés. Nous sommes allés à
examiner les différents types de collections et comment utiliser Collection et Set pour
à des fins de conteneur simples, tandis que List peut être utilisé pour maintenir des collections ordonnées. nous
a montré qu'il existe deux approches différentes pour utiliser une liste et que le maintien d'un
Une liste persistante est possible, mais ce n'est généralement pas la meilleure stratégie.

Nous avons ensuite développé tous les types de cartes, expliquant comment les combinaisons de base,
embeddable, et les types d'entité peuvent être utilisés comme clés et valeurs. Nous avons expérimenté et
ont montré des exemples d'utilisation de nombreuses combinaisons différentes de types de clé et de valeur,
illustrant comment chacun a changé la façon dont la collection a été cartographiée. Nous avons ensuite esquissé, dans
liste, les règles de base d'utilisation d'un type de carte.

Nous avons terminé les collections en examinant les cas secondaires d'ajout de doublons et
NULL aux collections et décrit les cas où la prise en charge peut être raisonnable.
Certaines bonnes pratiques et conseils pratiques sur l'utilisation des collections ont suivi.

Le chapitre suivant traite de l'utilisation des gestionnaires d'entités et des contextes de persistance dans

des moyens plus avancés que nous ne l'avons fait auparavant, en nous plongeant dans les pratiques et les nuances de en les injectant et en les utilisant dans les environnements Java EE et Java SE.

Épisode 208

CHAPITRE 6

Gestionnaire d'entités

Les entités ne persistent pas lorsqu'elles sont créées. Ils ne se retirent pas non plus à partir de la base de données lorsqu'ils sont récupérés. C'est la logique de l'application qui doivent manipuler les entités pour gérer leur cycle de vie persistant. JPA fournit le Interface EntityManager à cet effet afin de laisser les applications gérer et rechercher pour les entités de la base de données relationnelle.

Au début, cela peut sembler une limitation de JPA. Si le runtime de persistance sait quels objets sont persistants, pourquoi l'application devrait-elle être impliquée dans le processus? Soyez assuré que cette conception est à la fois délibérée et bien plus avantageuse pour le application que toute solution de persistance transparente. La persévérance est un partenariat entre l'application et le fournisseur de persistance. JPA apporte un niveau de contrôle et flexibilité qui ne pourrait être obtenue sans la participation active de l'application.

Dans le chapitre 2, nous avons présenté l'interface EntityManager et décrit certains des opérations de base qu'il prévoit pour opérer sur des entités. Nous avons prolongé cette discussion au chapitre 3 pour inclure une vue d'ensemble de l'environnement Java EE et des types de services qui ont un impact sur les applications de persistance. Enfin, dans les chapitres4 et5 nous avons décrit l'objet la cartographie relationnelle, la clé pour construire des entités à partir d'objets. Avec ce travail de fond en où nous sommes prêts à revoir les gestionnaires d'entités, les contextes de persistance et les unités de persistance, et d'entamer une discussion plus approfondie de ces concepts.

Contextes de persistance

Commençons par réintroduire les termes fondamentaux de JPA. Une *unité de persistance* est un nommé configuration des classes d'entités. Un contexte de persistance est un ensemble géré d'instances d'entité. Chaque contexte de persistance est associé à une unité de persistance, restreignant les classes de les instances gérées à l'ensemble défini par l'unité de persistance. Dire qu'une entité l'instance est *gérée* signifie qu'elle est contenue dans un contexte de persistance et qu'elle peut être agi par un responsable d'entité. C'est pour cette raison que nous disons qu'un gestionnaire d'entité gère un contexte de persistance.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_6

Épisode 209

Comprendre le contexte de persistance est la clé pour comprendre l'entité directeur. L'inclusion ou l'exclusion d'une entité d'un contexte de persistance déterminera le résultat de toute opération persistante sur celui-ci. Si le contexte de persistance participe à une transaction, l'état en mémoire des entités gérées sera synchronisé avec le base de données. Pourtant, malgré le rôle important qu'il joue, le contexte de persistance n'est jamais effectivement visible par l'application. Il est toujours accessible indirectement via l'entité manager et supposé être là quand nous en avons besoin.

Jusqu'ici tout va bien, mais comment le contexte de persistance est-il créé et quand est-ce se produire? Comment le gestionnaire d'entité figure-t-il dans l'équation? C'est là que ça commence à arriver intéressant.

Gestionnaires d'entités

Jusqu'à présent, nous n'avons démontré que les opérations de base des gestionnaires d'entités dans les deux les environnements Java SE et Java EE. Cependant, nous avons atteint un point où nous peut enfin révéler la gamme complète des configurations de gestionnaire d'entités. JPA n'en définit pas moins que trois types différents de gestionnaires d'entités, chacun ayant une approche différente gestion du contexte de persistance adaptée à un besoin d'application différent. Comme toi verra, le contexte de persistance n'est qu'une partie du puzzle.

Gestionnaires d'entités gérés par des conteneurs

Dans l'environnement Java EE, le moyen le plus courant d'acquérir un gestionnaire d'entités est de en utilisant l'annotation @PersistenceContext pour en injecter une. Un responsable d'entité obtenu en cette méthode est appelée *gestion* du conteneur car le conteneur gère le cycle de vie du gestionnaire d'entités, généralement en renvoyant par proxy celui qu'il obtient du fournisseur de persistance. L'application n'a pas besoin de la créer ni de la fermer. C'est le style du gestionnaire d'entités nous avons démontré au chapitre 3.

Les gestionnaires d'entités gérées par conteneurs se déclinent en deux variétés. Le style d'un conteneur-Le gestionnaire d'entités géré détermine comment il fonctionne avec les contextes de persistance. La première et le style le plus courant est appelé *à portée de transaction* . Cela signifie que la persistance les contextes gérés par le gestionnaire d'entités sont délimités par la transaction JTA active, se terminant lorsque la transaction est terminée. Le deuxième style est appelé *étendu* . Élargi les gestionnaires d'entités travaillent avec un contexte de persistance unique lié au cycle de vie d'un bean session avec état et sont étendues à la durée de vie de ce bean session avec état, potentiellement couvrant plusieurs transactions.

Épisode 210

Notez que par défaut, un contexte de persistance géré par l'application associé avec un gestionnaire d'entités JTA et est créé dans le cadre d'une transaction active sera joint automatiquement à cette transaction

Portée par transaction

Tous les exemples de gestionnaire d'entités que nous avons montrés jusqu'à présent pour l'environnement Java EE ont été des gestionnaires d'entités à vocation transactionnelle. Un gestionnaire d'entités transactionnelles est renvoyée chaque fois que la référence créée par l'annotation @PersistenceContext est résolu. Comme nous l'avons mentionné dans le chapitre 3 , un gestionnaire d'entités transactionnelles est sans état, ce qui signifie qu'il peut être stocké en toute sécurité sur n'importe quel composant Java EE. Parce que le container le gère pour nous, il est également fondamentalement sans entretien.

Encore une fois, introduisons un bean session sans état qui utilise un gestionnaire d'entité. Référencement6-1 montre la classe de bean pour un bean session qui gère Renseignements sur le projet. Le gestionnaire d'entités est injecté dans le champ em en utilisant le @PersistenceContext et est ensuite utilisée dans les méthodes métier du bean.

Liste 6-1. Le bean de session ProjectService

```
@Apatride
public class ProjectService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    public void assignEmployeeToProject (int empId, int projectId) {
        Projet project = em.find (Project.class, projectId);
        Employé employé = em.find (Employee.class, empId);
        project.getEmployees (). add (employé);
        employee.getProjects (). add (projet);
    }

    // ...
}
```

Nous avons décrit le gestionnaire d'entités transactionnelles comme étant sans état. Si c'est le cas, comment cela peut-il fonctionner avec un contexte de persistance? La réponse réside dans la transaction JTA. Tout les gestionnaires d'entités gérés par des conteneurs dépendent des transactions JTA car ils peuvent utiliser la transaction comme moyen de suivre les contextes de persistance. Chaque fois qu'une opération est appelée

Épisode 211

sur le gestionnaire d'entités, le proxy de conteneur de ce gestionnaire d'entités vérifie si un contexte de persistance est associé à la transaction JTA du conteneur. S'il en trouve un, le Le gestionnaire d'entités utilisera ce contexte de persistance. S'il n'en trouve pas, il crée un nouveau contexte de persistance et l'associe à la transaction. Lorsque la transaction se termine, le contexte de persistance disparaît.

Voyons un exemple. Considérez la méthode assignEmployeeToProject () de la liste 6-1. La première chose que fait la méthode est de rechercher l'employé et le projet instances utilisant l'opération find (). Lorsque la première méthode find () est appelée, le conteneur vérifie une transaction. Par défaut, le conteneur s'assurera qu'un transaction est active à chaque fois qu'une méthode de bean session démarre, donc le gestionnaire d'entités dans cet exemple en trouvera un prêt. Il vérifie ensuite un contexte de persistance. C'est le premier chaque fois qu'un appel de gestionnaire d'entités s'est produit, il n'y a donc pas encore de contexte de persistance. le Le gestionnaire d'entités en crée un nouveau et l'utilise pour trouver le projet.

Lorsque le responsable d'entité est utilisé pour rechercher l'employé, il vérifie transaction à nouveau et cette fois trouve celui qu'elle a créé lors de la recherche du projet. Il puis réutilise ce contexte de persistance pour rechercher l'employé. À ce stade, l'employé et project sont tous deux des instances d'entité gérées. L'employé est ensuite ajouté au projet, en mettant à jour à la fois l'employé et les entités du projet. Lorsque l'appel de méthode se termine, la transaction est validée. Parce que les instances d'employé et de projet étaient géré, le contexte de persistance peut détecter tout changement d'état dans ces derniers et il se met à jour la base de données pendant la validation. Lorsque la transaction est terminée, le contexte de persistance s'en va.

Ce processus est répété à chaque fois qu'une ou plusieurs opérations du gestionnaire d'entités sont invoqué dans une transaction.

Élargi

Afin de décrire le gestionnaire d'entités étendu, il faut d'abord parler un peu de stateful haricots de session. Comme vous l'avez appris au chapitre 3, les beans session avec état sont conçus pour contenir état conversationnel. Une fois acquise par un client, la même instance de bean est utilisée pour le durée de la conversation jusqu'à ce que le client appelle l'une des méthodes marquées @Remove sur le haricot. Pendant que la conversation est active, les méthodes commerciales du client peuvent stocker

et accéder aux informations en utilisant les champs du bean.

Essayons d'utiliser un bean session avec état pour aider à gérer un département. Notre objectif est pour créer un objet métier pour une entité Department qui fournit des opérations métier relative à cette entité. Référencement6-2 montre notre première tentative. La méthode métier init ()

Épisode 212

est appelé par le client pour initialiser l'ID de service. Nous stockons ensuite cet identifiant de département sur l'instance du bean, et la méthode addEmployee () l'utilise pour trouver le département et apportez les modifications nécessaires. Du point de vue du client, ils n'ont qu'à définir l'ID de service une fois, puis les opérations suivantes se réfèrent toujours au même département.

Liste 6-2. Première tentative chez Department Manager Bean

```
@Stateful
Public class DepartmentManager {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;
    int deptId;

    public void init (int deptId) {
        this.deptId = deptId;
    }

    public void setName (String name) {
        Département dept = em.find (Department.class, deptId);
        dept.setName (nom);
    }

    public void addEmployee (int empId) {
        Département dept = em.find (Department.class, deptId);
        Employé emp = em.find (Employee.class, empId);
        dept.getEmployees (). add (emp);
        emp.setDepartment (dept);
    }

    // ...

    @Retirer
    public void done () {
    }
}
```


Épisode 213

La première chose qui devrait ressortir en regardant ce haricot est qu'il semble inutile de chercher le département à chaque fois. Après tout, nous avons le ID de service, alors pourquoi ne pas simplement stocker l'instance d'entité Department? Annonce 6-3

révise notre première tentative en recherchant le département une fois pendant la méthode init ()
puis réutiliser l'instance d'entité pour chaque méthode métier.

Liste 6-3. Deuxième tentative chez Department Manager Bean

```
@Stateful
Public class DepartmentManager {
        @PersistenceContext (unitName = "EmployeeService")
        EntityManager em;
        Département départemental;

        public void init (int deptId) {
                dept = em.find (Department.class, deptId);
        }

        public void setName (String name) {
                dept.setName (nom);
        }

        public void addEmployee (int empId) {
                Employé emp = em.find (Employee.class, empId);
                dept.getEmployees (). add (emp);
                emp.setDepartment (dept);
        }

        // ...

        @Retirer
        public void done () {
        }
}
```

Cette version semble mieux adaptée aux capacités d'un bean session avec état. Il est certainement plus naturel de réutiliser l'instance d'entité Department au lieu de rechercher à chaque fois. Mais il y a un problème. Le gestionnaire d'entité dans le Listing 6-3 est transaction-portée. En supposant qu'il n'y a pas de transaction active du client, chaque méthode sur le

Épisode 214

bean va démarrer et valider une nouvelle transaction car l'attribut de transaction par défaut pour chaque méthode est REQUIS. Puisqu'il y a une nouvelle transaction pour chaque méthode, le Le gestionnaire d'entités utilisera un contexte de persistance différent à chaque fois.

Même si l'instance Department existe toujours, le contexte de persistance qui a utilisé pour le gérer, il a disparu lorsque la transaction associée à l'appel init () s'est terminée. nous désigner l'entité Department dans ce cas comme étant *détachée* d'un contexte de persistance. L'instance est toujours là et peut être utilisée, mais toute modification de son état sera ignorée. Par exemple, appeler setName () changera le nom dans l'instance d'entité, mais le les changements ne seront jamais reflétés dans la base de données.

C'est la situation que le gestionnaire d'entités étendu est conçu pour résoudre. Conçu spécifiquement pour les beans session avec état, il empêche les entités de se détacher lorsque les transactions se terminent. Avant d'aller trop loin, introduisons notre troisième et dernier tentative de haricot de chef de service. Référencement6-4 montre notre exemple précédent mis à jour pour utiliser un contexte de persistance étendu.

Liste 6-4. Utilisation d'un Extended Entity Manager

```
@Stateful
Public class DepartmentManager {
        @PersistenceContext (unitName = "EmployeeService",
```

```
                                type = PersistenceContextType.EXTENDED)
    EntityManager em;
    Département départemental;

    public void init (int deptId) {
        dept = em.find (Department.class, deptId);
    }

    public void setName (String name) {
        dept.setName (nom);
    }

    public void addEmployee (int empId) {
        Employé emp = em.find (Employee.class, empId);
        dept.getEmployees (). add (emp);
        emp.setDepartment (dept);
    }
```


Épisode 215


```
    // ...

    @Retirer
    public void done () {
    }
}
```

Comme vous pouvez le voir, nous n'avons modifié qu'une seule ligne. L'annotation @PersistenceContext que nous avons introduit dans le chapitre 3 a un attribut de type spécial qui peut être défini sur TRANSACTION ou PROLONGÉ. Ces constantes sont définies par le PersistenceContextType type énuméré. TRANSACTION est la valeur par défaut et correspond à la portée de la transaction des gestionnaires d'entités que nous utilisons jusqu'à présent. EXTENDED signifie qu'une entité étendue manager doit être utilisé.

Une fois cette modification apportée, le bean de chef de service fonctionne désormais comme prévu. Les gestionnaires d'entités étendus créent un contexte de persistance lorsqu'un bean session avec état est créée et dure jusqu'à ce que le bean soit supprimé. Contrairement au contexte de persistance d'un gestionnaire d'entités à portée transaction, qui commence au début de la transaction et dure jusqu'à la fin d'une transaction, le contexte de persistance d'une entité étendue le manager durera toute la durée de la conversation. Parce que l'entité Department est toujours géré par le même contexte de persistance, chaque fois qu'il est utilisé dans une transaction les modifications seront automatiquement écrites dans la base de données.

Le contexte de persistance étendu permet d'écrire des beans session avec état dans un plus adapté à leurs capacités. Plus tard, nous discuterons des limitations spéciales sur le gestion des transactions des gestionnaires d'entités étendues, mais dans l'ensemble, ils sont bien adapté au type d'exemple que nous avons montré ici.

Gestionnaires d'entités gérées par les applications

Au chapitre 2, nous avons présenté JPA avec un exemple écrit à l'aide de Java SE. le gestionnaire d'entités dans cet exemple, et tout gestionnaire d'entités créé à partir du createEntityManager () appel d'une instance EntityManagerFactory, est ce que nous appelons un gestionnaire d'entités *géré par l'application* . Ce nom vient du fait que le l'application, plutôt que le conteneur, gère le cycle de vie du gestionnaire d'entités. Remarque que tous les gestionnaires d'entités ouverts, qu'ils soient gérés par des conteneurs ou gérés par des applications, sont associés à une instance EntityManagerFactory. L'usine utilisée pour créer le gestionnaire d'entités est accessible à partir de l'appel getEntityManagerFactory () sur le

Interface EntityManager.
198

Épisode 216

Bien que nous nous attendions à ce que la majorité des applications soient écrites à l'aide de conteneurs-
les gestionnaires d'entités gérées, les gestionnaires d'entités gérées par les applications ont encore un rôle à jouer.
Il s'agit du seul type de gestionnaire d'entités disponible dans Java SE et, comme vous le verrez, ils peuvent
être également utilisé dans Java EE.

La création d'un gestionnaire d'entités géré par une application est assez simple. Tout ce dont tu as besoin c'est
un EntityManagerFactory pour créer l'instance. Ce qui sépare Java SE et Java EE pour
les gestionnaires d'entités gérés par l'application, ce n'est pas la façon dont vous créez le gestionnaire d'entités mais
vous obtenez l'usine. Le Listing 6-5 illustre l'utilisation de la classe Persistence pour amorcer
une instance EntityManagerFactory qui est ensuite utilisée pour créer un gestionnaire d'entités.

Liste 6-5. Gestionnaires d'entités gérés par des applications dans Java SE

```
public class EmployeeClient {
    public static void main (String [] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory ("EmployeeService");
        EntityManager em = emf.createEntityManager ();

        List <Employee> emps = em.createQuery ("SELECT e FROM Employee e")
                                                    .getResultList ();
        for (Employee e: emps) {
            System.out.println (e.getId () + "," + e.getName ());
        }

        em.close ();
        emf.close ();
    }
}
```

La classe Persistence offre deux variantes du même createEntityManager ()
méthode qui peut être utilisée pour créer une instance EntityManagerFactory pour un
nom de l'unité de persistance. Le premier, en spécifiant uniquement le nom de l'unité de persistance, renvoie le
fabrique créée avec les propriétés par défaut définies dans le fichier persistence.xml. le
la seconde forme de l'appel de méthode permet de transmettre une carte de propriétés, d'ajouter à,
ou remplacer les propriétés spécifiées dans persistence.xml. Ce formulaire est utile lorsque
les propriétés JDBC requises peuvent ne pas être connues avant le démarrage de l'application, peut-être
avec des informations fournies en tant que paramètres de ligne de commande. L'ensemble des propriétés actives

Épisode 217

pour un gestionnaire d'entités peut être déterminé via la méthode getProperties () sur le
Interface EntityManager. Nous discutons des propriétés des unités de persistance au chapitre 14 .

La meilleure façon de créer un gestionnaire d'entités géré par une application dans Java EE est d'utiliser
l'annotation @PersistenceUnit pour déclarer une référence à EntityManagerFactory
pour une unité de persistance. Une fois acquise, l'usine peut être utilisée pour créer une entité
manager, qui peut être utilisé comme dans Java SE. Référencement6-6 démontre
injection d'un EntityManagerFactory dans un servlet et son utilisation pour créer un

gestionnaire d'entités afin de vérifier un ID utilisateur.

Liste 6-6. Gestionnaires d'entités gérés par des applications dans Java EE

```
La classe publique LoginServlet étend HttpServlet {
    @PersistenceUnit (unitName = "EmployeeService")
    EntityManagerFactory emf;

    protected void doPost (requête HttpServletRequest,
                                                    Réponse HttpServletResponse) {
        String userId = request.getParameter ("utilisateur");

        // vérifier l'utilisateur valide
        EntityManager em = emf.createEntityManager ();
        essayez {
            Utilisateur utilisateur = em.find (User.class, userId);
            if (utilisateur == null) {
                // retourne la page d'erreur
                // ...
            }
        } enfin {
            em.close ();
        }

        // ...
    }
}
```

Une chose commune à ces deux exemples est que le gestionnaire d'entités est explicitement fermé avec l'appel close () lorsqu'il n'est plus nécessaire. C'est l'un des cycles de vie exigences d'un responsable d'entité qui doivent être exécutées manuellement en cas de

Épisode 218

gestionnaires d'entités gérées par des applications; il est normalement pris en charge automatiquement par gestionnaires d'entités gérés par des conteneurs. De même, l'instance EntityManagerFactory doit également être fermé, mais uniquement dans l'application Java SE. Dans Java EE, le conteneur se ferme l'usine automatiquement, donc aucune étape supplémentaire n'est requise.

En termes de contexte de persistance, le gestionnaire d'entités géré par l'application est similaire à un gestionnaire d'entités géré par conteneur étendu. Lorsqu'une application gestionnaire d'entités gérées est créé, il crée son propre contexte de persistance privé qui dure jusqu'à la fermeture du gestionnaire d'entités. Cela signifie que toutes les entités gérées par le Le gestionnaire d'entité le restera, indépendamment de toute transaction.

Le rôle du gestionnaire d'entités géré par l'application dans Java EE est quelque peu spécialisé. Si des transactions locales aux ressources sont requises pour une opération, une application Le gestionnaire d'entités géré est le seul type de gestionnaire d'entités qui peut être configuré avec ce type de transaction dans le serveur. Comme nous le décrivons dans la section suivante, le les exigences de transaction d'un gestionnaire d'entité étendu peuvent les rendre difficiles à traiter avec dans certaines situations. Les gestionnaires d'entités gérés par des applications peuvent être utilisés en toute sécurité des beans session avec état pour atteindre des objectifs similaires.

Gestion des transactions

Développer une application de persistance concerne autant la gestion des transactions que à propos du mappage objet-relationnel. Les transactions définissent quand elles sont nouvelles, modifiées ou supprimées les entités sont synchronisées avec la base de données. Comprendre comment les contextes de persistance

interagir avec les transactions est un élément fondamental du travail avec JPA.

Notez que nous avons dit contextes de persistance, pas gestionnaires d'entités. Il y a plusieurs différents types de gestionnaires d'entités, mais tous utilisent un contexte de persistance en interne. L'entité le type de gestionnaire détermine la durée de vie d'un contexte de persistance, mais toute persistance les contextes se comportent de la même manière lorsqu'ils sont associés à une transaction.

Il existe deux types de gestion des transactions pris en charge par JPA. Le premier est les transactions locales aux ressources, qui sont les transactions natives des pilotes JDBC qui sont référencés par une unité de persistance. Le deuxième type de gestion des transactions est JTA transactions, qui sont les transactions du serveur Java EE, prenant en charge plusieurs ressources participantes, gestion du cycle de vie des transactions et XA distribué transactions.

Épisode 219

Les gestionnaires d'entités gérés par des conteneurs utilisent toujours les transactions JTA, tandis que les les gestionnaires d'entités gérées peuvent utiliser l'un ou l'autre type. Parce que JTA n'est généralement pas disponible dans Applications Java SE, le fournisseur doit prendre en charge uniquement les transactions locales aux ressources dans cet environnement. Le type de transaction par défaut et préféré pour les applications Java EE est JTA. Comme nous le décrivons dans la section suivante, propager des contextes de persistance avec JTA transactions est un avantage majeur pour les applications de persistance d'entreprise.

Le type de transaction est défini pour une unité de persistance et est configuré à l'aide de fichier persistence.xml. Nous discutons de ce paramètre et de son application au chapitre 14 .

Gestion des transactions JTA

Pour parler des transactions JTA, nous devons d'abord discuter de la différence entre synchronisation de transaction, association de transaction et propagation de transaction. *La synchronisation des transactions* est le processus par lequel un contexte de persistance est enregistré avec une transaction afin que le contexte de persistance puisse être notifié lorsqu'une transaction s'engage. Le fournisseur utilise cette notification pour s'assurer qu'un contexte de persistance donné est correctement vidé dans la base de données. *L'association de transaction* est l'acte de lier un contexte de persistance à une transaction. Vous pouvez également considérer cela comme la persistance *active* contexte dans le cadre de cette transaction. *La propagation des transactions* est le processus de partage d'un contexte de persistance entre plusieurs gestionnaires d'entités gérés par des conteneurs dans un transaction unique.

Il ne peut y avoir qu'un seul contexte de persistance associé et propagé à travers un Transaction JTA. Tous les gestionnaires d'entités gérées par conteneur dans la même transaction doivent partagent le même contexte de persistance propagé.

Contextes de persistance à l'échelle des transactions

Comme son nom l'indique, un contexte de persistance transactionnel est lié au cycle de vie de la transaction. Il est créé par le conteneur lors d'une transaction et sera fermé lorsque la transaction est terminée. Les gestionnaires d'entités à portée de transaction sont responsables pour créer automatiquement des contextes de persistance transactionnels en cas de besoin. Nous disons uniquement lorsque cela est nécessaire, car la création de contexte de persistance à l'échelle de la transaction est *paresseuse* . Un Le gestionnaire d'entités créera un contexte de persistance uniquement lorsqu'une méthode est appelée sur le gestionnaire d'entités et lorsqu'il n'y a pas de contexte de persistance disponible.

Lorsqu'une méthode est appelée sur le gestionnaire d'entités à portée transaction, elle doit d'abord voir s'il existe un contexte de persistance propagé. S'il en existe un, le gestionnaire d'entités

Épisode 220

utilise ce contexte de persistance pour effectuer l'opération. S'il n'en existe pas, l'entité
le gestionnaire demande un nouveau contexte de persistance au fournisseur de persistance, puis
marque ce nouveau contexte de persistance comme le contexte de persistance propagé pour le
transaction avant d'exécuter l'appel de méthode. Toutes les transactions ultérieures
les opérations du gestionnaire d'entités, dans ce composant ou dans tout autre, utiliseront par la suite ce
a créé un contexte de persistance. Ce comportement fonctionne indépendamment du fait que le conteneur-
la démarcation des transactions gérées ou gérées par bean a été utilisée.

La propagation du contexte de persistance simplifie la construction d'entreprise
applications. Lorsqu'une entité est mise à jour par un composant à l'intérieur d'une transaction, tout
les références ultérieures à la même entité correspondront toujours à l'instance correcte,
quel que soit le composant qui obtient la référence d'entité. Propager la persistance
contexte donne aux développeurs la liberté de créer des applications faiblement couplées, sachant
qu'ils obtiendront toujours les bonnes données même s'ils ne partagent pas la même entité
instance de gestionnaire.

Pour illustrer la propagation d'un contexte de persistance transactionnel,
nous introduisons un bean de service d'audit qui stocke des informations sur un
transaction terminée. Référencement6-7 montre l'implémentation complète du bean. le
La méthode logTransaction () garantit qu'un identifiant d'employé est valide en essayant de trouver
l'employé utilisant le gestionnaire d'entités.

Liste 6-7. Bean de session AuditService

```
@Apatride
public class AuditService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    public void logTransaction (int empId, String action) {
        // vérifier que le numéro d'employé est valide
        if (em.find (Employee.class, empId) == null) {
            throw new IllegalArgumentException ("Identifiant d'employé inconnu");
        }
        LogRecord lr = nouveau LogRecord (empId, action);
        em.persist (lr);
    }
}
```


Épisode 221

Considérons maintenant le fragment de l'exemple de bean de session EmployeeService
montré dans l'extrait 6-8 . Après la création d'un employé, la méthode logTransaction () du
Le bean session AuditService est appelé pour enregistrer l'événement «employé créé».

Liste 6-8. Journalisation des transactions des employés

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;
```

```
    Audit @EJB AuditService;
    public void createEmployee (Employee emp) {
        em.persist (emp);
        audit.logTransaction (emp.getId (), "employé créé");
    }

    // ...
}
```

Même si l'employé nouvellement créé n'est pas encore dans la base de données, le bean d'audit peut trouver l'entité et vérifier qu'elle existe. Cela fonctionne car les deux beans sont en fait partageant le même contexte de persistance. L'attribut de transaction de createEmployee () La méthode est OBLIGATOIRE par défaut car aucun attribut n'a été défini explicitement. le container garantira qu'une transaction est démarrée avant que la méthode ne soit appelée. Lorsque persist () est appelé sur le gestionnaire d'entités, le conteneur vérifie si un contexte de persistance est déjà associé à la transaction. Supposons dans ce au cas où il s'agissait de la première opération du gestionnaire d'entités dans la transaction, le conteneur crée un nouveau contexte de persistance et le marque comme étant le contexte propagé.

Lorsque la méthode logTransaction () démarre, elle émet un appel find () sur l'entité manager de l'AuditService. Nous sommes assurés d'être dans une transaction car le L'attribut transaction est également OBLIGATOIRE et la transaction gérée par le conteneur depuis createEmployee () a été étendu à cette méthode par le conteneur. Lorsque la recherche () est appelée, le conteneur vérifie à nouveau un contexte de persistance actif. Il trouve celui créé dans la méthode createEmployee () et utilise ce contexte de persistance pour recherchez l'entité. Parce que la nouvelle instance Employee est gérée par ce contexte de persistance, il est renvoyé avec succès.

Épisode 222

Considérons maintenant le cas où logTransaction () a été déclaré avec le REQUIRES_NEW attribut de transaction au lieu de REQUIRED par défaut. Avant le L'appel de la méthode logTransaction () démarre, le conteneur suspendra la transaction hérité de createEmployee () et démarrer une nouvelle transaction. Lorsque la recherche () est invoquée sur le gestionnaire d'entités, il vérifiera la transaction en cours pour un contexte de persistance active uniquement pour déterminer qu'il n'en existe pas. Une nouvelle persistance le contexte sera créé en commençant par l'appel find (), et ce contexte de persistance sera être le contexte de persistance actif pour le reste de l'appel logTransaction (). Étant donné que la transaction démarrée dans createEmployee () n'a pas encore été validée, le nouveau l'instance Employee créée n'est pas dans la base de données et n'est donc pas visible par ce nouveau contexte de persistance. La méthode find () retournera null, et le logTransaction () la méthode lèvera une exception en conséquence.

La règle de base pour la propagation du contexte de persistance est que le contexte de persistance se propage au fur et à mesure que la transaction JTA se propage. Par conséquent, il est important de comprendre non seulement lorsque les transactions commencent et se terminent, mais aussi lorsqu'une méthode commerciale attend hériter du contexte de transaction d'une autre méthode et ce serait Incorrect. Avoir un plan clair de gestion des transactions dans votre application est essentiel pour tirer le meilleur parti de la propagation du contexte de persistance.

Contextes de persistance étendus

Le cycle de vie d'un contexte de persistance étendue est lié au bean session avec état auquel il est lié. Contrairement à un gestionnaire d'entités à portée de transaction qui crée un nouveau contexte de persistance pour chaque transaction, le gestionnaire d'entités étendu d'un état Le bean session utilise toujours le même contexte de persistance. Le bean session avec état est associé à un seul contexte de persistance étendue créé lorsque le bean l'instance est créée et fermée lorsque l'instance du bean est supprimée. Cela a des implications

pour les caractéristiques d'association et de propagation de la persistance étendue
le contexte.

L'association de transaction pour les contextes de persistance étendue est *impatiente* . Dans le cas des transactions gérées par le conteneur, dès qu'un appel de méthode démarre sur le bean, le container associe automatiquement le contexte de persistance à la transaction. également dans le cas de transactions gérées par bean, dès que UserTransaction.begin () est invoqué dans une méthode bean, le conteneur intercepte l'appel et effectue la même chose association.

Épisode 223

Parce qu'un gestionnaire d'entités à portée de transaction utilisera une persistance existante contexte associé à la transaction avant qu'elle ne crée un nouveau contexte de persistance, il est possible de partager un contexte de persistance étendu avec d'autres gestionnaires d'entités. Tant que le contexte de persistance étendue est propagé avant tous les gestionnaires d'entités à portée de transaction sont accessibles, la même persistance étendue le contexte sera partagé par tous les composants.

Similaire au bean d'audit EmployeeService présenté dans la liste 6-8, considérer la même modification apportée à un bean session avec état DepartmentManager pour auditer lorsqu'un l'employé est ajouté à un service. Le listing 6-9 montre cet exemple.

Liste 6-9. Modifications du service de journalisation

```
@Stateful
Public class DepartmentManager {
    @PersistenceContext (unitName = "EmployeeService",
                                type = PersistenceContextType.EXTENDED)
    EntityManager em;
    Département départemental;
    Audit @EJB AuditService;

    public void init (int deptId) {
        dept = em.find (Department.class, deptId);
    }

    public void addEmployee (int empId) {
        Employé emp = em.find (Employee.class, empId);
        dept.getEmployees (). add (emp);
        emp.setDepartment (dept);
        audit.logTransaction (emp.getId (),
                                "ajouté au département" + dept.getName ());
    }

    // ...
}
```

La méthode addEmployee () a un attribut de transaction par défaut REQUIRED. Étant donné que le conteneur associe avec empressement des contextes de persistance étendus, le le contexte de persistance stocké sur le bean session sera immédiatement associé au

Épisode 224

transaction au démarrage de l'appel de méthode. Cela entraînera la relation entre le
géré les entités Department et Employee à conserver dans la base de données lorsque
la transaction est validée. Cela signifie également que le contexte de persistance étendue sera désormais
partagé par d'autres contextes de persistance transactionnels utilisés dans les méthodes appelées depuis
addEmployee ().

La méthode logTransaction () de cet exemple héritera du contexte de transaction
de addEmployee () car son attribut de transaction est le REQUIRED par défaut, et un
transaction est active lors de l'appel à addEmployee (). Lorsque la méthode find () est
invoqué, le gestionnaire d'entités à portée transaction vérifie un contexte de persistance actif
et trouvera le contexte de persistance étendue à partir du DepartmentManager. Il sera alors
utilisez ce contexte de persistance pour exécuter l'opération. Toutes les entités gérées du
Le contexte de persistance étendue devient visible pour le gestionnaire d'entités à portée de transaction.

Collision de contexte de persistance

Nous avons dit plus tôt qu'un seul contexte de persistance pouvait être propagé avec un JTA
transaction. Nous avons également dit que le contexte de persistance étendue essaierait toujours de
se faire le contexte de persistance active. Cela peut conduire à des situations dans lesquelles les deux
les contextes de persistance se heurtent. Considérez, par exemple, qu'un apatride
Le bean session avec un gestionnaire d'entités à portée transaction crée une nouvelle persistance
context puis invoque une méthode sur un bean session avec état avec un
contexte de persistance. Au cours de l'association impatiente du contexte de persistance étendue,
le conteneur vérifiera s'il existe déjà un contexte de persistance actif.
Si tel est le cas, il doit être le même que le contexte de persistance étendue qu'il essaie de
associé, ou une exception sera levée. Dans cet exemple, le bean session avec état sera
trouver le contexte de persistance transactionnel créé par le bean session sans état,
et l'appel à la méthode du bean session avec état échouera. Il ne peut y avoir qu'un seul actif
contexte de persistance pour une transaction. Figure 6-1 illustre ce cas.

Épisode 225

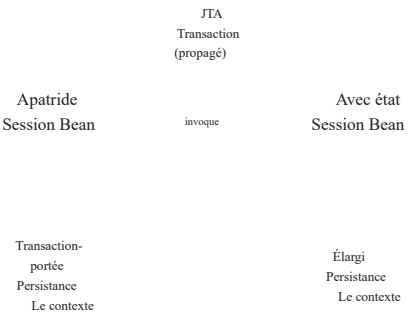

Figure 6-1. *Collision de contexte de persistance*

Alors que la propagation du contexte de persistance étendue est utile si un bean session avec état
avec un contexte de persistance étendu est le premier EJB à être appelé dans une chaîne d'appels, il

limite les situations dans lesquelles d'autres composants peuvent appeler le bean session avec état
s'ils utilisent également des gestionnaires d'entités. Cela peut être courant ou non selon
sur l'architecture de votre application, mais c'est quelque chose à garder à l'esprit lors de la planification
dépendances entre les composants.

Une façon de contourner ce problème consiste à modifier l'attribut de transaction par défaut
pour le bean session avec état qui utilise le contexte de persistance étendue. Si la valeur par défaut
l'attribut de transaction est REQUIRES_NEW, toute transaction active sera suspendue avant
la méthode du bean session avec état démarre, lui permettant d'associer sa persistance étendue
contexte avec la nouvelle transaction. C'est une bonne stratégie si le bean session avec état appelle
in à d'autres beans session sans état et doit propager le contexte de persistance. Remarque
qu'une utilisation excessive de l'attribut de transaction REQUIRES_NEW peut conduire à l'application
problèmes de performances car beaucoup plus de transactions que la normale seront créées,
et les transactions actives seront suspendues et reprises.

Si le bean session avec état est en grande partie autonome; c'est-à-dire qu'il n'appelle pas d'autres
session et n'a pas besoin de propager son contexte de persistance, une valeur par défaut
Le type d'attribut de transaction NOT_SUPPORTED peut valoir la peine d'être pris en compte. Dans ce cas, tout
la transaction active sera suspendue avant le démarrage de la méthode du bean session avec état, mais
aucune nouvelle transaction ne sera lancée. S'il existe des méthodes pour lesquelles écrire des données
la base de données, ces méthodes peuvent être remplacées pour utiliser la transaction REQUIRES_NEW
attribut.

Épisode 226

Référencement 6-10 répète le bean DepartmentManager, cette fois avec quelques
méthodes getter et attributs de transaction personnalisés. Nous avons défini la valeur par défaut
attribut transaction à REQUIRES_NEW pour forcer une nouvelle transaction par défaut lorsque
une méthode métier est invoquée. Pour la méthode getName (), nous n'avons pas besoin d'un nouveau
transaction car aucune modification n'est en cours, elle a donc été définie sur NOT_SUPPORTED.
Cela suspendra la transaction en cours, mais n'entraînera pas une nouvelle transaction
créé. Avec ces modifications, le bean DepartmentManager est accessible dans n'importe quel
situation, même s'il existe déjà un contexte de persistance actif.

Listing 6-10. Personnalisation des attributs de transaction pour éviter les collisions

```
@Stateful
@TransactionAttribute (TransactionAttributeType.REQUIRES_NEW)
Public class DepartmentManager {
    @PersistenceContext (unitName = "EmployeeService",
                                    type = PersistenceContextType.EXTENDED)
    EntityManager em;
    Département départemental;
    Audit @EJB AuditService;

    public void init (int deptId) {
        dept = em.find (Department.class, deptId);
    }

    @TransactionAttribute (TransactionAttributeType.NOT_SUPPORTED)
    public String getName () {return dept.getName (); }
    public void setName (nom de la chaîne) {dept.setName (nom); }

    public void addEmployee (int empId) {
        Employé emp = em.find (empId, Employee.class);
        dept.getEmployees (). add (emp);
        emp.setDepartment (dept);
```

```
            audit.logTransaction (emp.getId (), "ajouté au département" + dept.getName ());
        }
        // ...
    }
```


Épisode 227

Enfin, une dernière option à considérer consiste à utiliser un gestionnaire d'entités géré par l'application au lieu d'un gestionnaire d'entités étendu. S'il n'est pas nécessaire de propager la persistance contexte, le gestionnaire d'entités étendu n'ajoute pas beaucoup de valeur sur une application gestionnaire d'entités gérées. Le bean session avec état peut créer une application en toute sécurité. gestionnaire d'entités géré, stockez-le sur l'instance du bean et utilisez-le pour la persistance opérations sans avoir à se soucier de savoir si une transaction active a déjà un contexte de persistance propagé. Un exemple de cette technique est présenté plus loin dans la section «Contextes de persistance gérés par les applications».

Héritage du contexte de persistance

La restriction d'un seul bean session avec état avec un contexte de persistance étendu pouvoir participer à une transaction JTA peut entraîner des difficultés dans certaines situations. Par exemple, le modèle que nous avons suivi plus tôt dans ce chapitre pour la persistance étendue Le contexte était d'encapsuler le comportement d'une entité derrière une façade de session avec état. Dans notre exemple, les clients ont travaillé avec un bean session DepartmentManager au lieu du instance réelle de l'entité Department. Parce qu'un département a un manager, c'est logique d'étendre cette façade à l'entité Salarié également.

Référencement 6-11 montre les modifications apportées au bean DepartmentManager afin qu'il retourne un bean EmployeeManager de la méthode getManager () pour représenter le directeur du département. Le bean EmployeeManager est injecté puis initialisé lors de l'appel de la méthode init ().

Annonce 6-11. Créer et renvoyer un bean session avec état

```
@Stateful
Public class DepartmentManager {
    @PersistenceContext (unitName = "EmployeeService",
                         type = PersistenceContextType.EXTENDED)
    EntityManager em;
    Département départemental;
    @EJB EmployeeManager manager;

    public void init (int deptId) {
        dept = em.find (Department.class, deptId);
        manager.init ();
    }
```


Épisode 228


```
    public EmployeeManager getManager () {
        gestionnaire de retour;
```

```
        }

        // ...
}
```

La méthode init () doit-elle réussir ou échouer? Jusqu'à présent, sur la base de ce que nous avons décrit, il semble que cela devrait échouer. Lorsque init () est appelé sur le bean DepartmentManager, son Le contexte de persistance étendue sera propagé avec la transaction. Dans la suite appel à init () sur le bean EmployeeManager, il tentera d'associer sa propre extension contexte de persistance avec la transaction, provoquant une collision entre les deux.

Peut-être étonnamment, cet exemple fonctionne réellement. Lorsqu'un bean session avec état avec un contexte de persistance étendu crée un autre bean session avec état qui utilise également un contexte de persistance étendu, l'enfant héritera du contexte de persistance du parent. le Le bean EmployeeManager hérite du contexte de persistance du DepartmentManager bean lorsqu'il est injecté dans l'instance DepartmentManager. Les deux haricots peuvent maintenant être utilisés ensemble dans la même transaction.

Contextes de persistance gérés par les applications

Comme les contextes de persistance gérés par les conteneurs, la persistance gérée par les applications les contextes peuvent être synchronisés avec les transactions JTA. Synchroniser la persistance contexte avec la transaction signifie qu'un vidage se produira si la transaction est validée, mais le contexte de persistance ne sera considéré comme associé par aucun conteneur géré gestionnaires d'entités. Il n'y a pas de limite au nombre de persistance gérée par l'application contextes qui peuvent être synchronisés avec une transaction, mais un seul conteneur géré le contexte de persistance sera toujours associé. C'est l'une des différences les plus importantes entre les gestionnaires d'entités gérés par l'application et gérés par conteneur.

Un gestionnaire d'entités géré par une application participe à une transaction JTA dans l'un des deux façons. Si le contexte de persistance est créé à l'intérieur de la transaction, la persistance Le fournisseur synchronisera automatiquement le contexte de persistance avec la transaction. Si le contexte de persistance a été créé plus tôt (en dehors d'une transaction ou dans une transaction qui s'est terminée depuis), le contexte de persistance peut être synchronisé manuellement avec la transaction en appelant joinTransaction () sur l'interface EntityManager. Une fois que synchronisé, le contexte de persistance sera automatiquement vidé lorsque la transaction s'engage.

Épisode 229

Référencement 6-12 montre une variante du DepartmentManager du Listing 6-11 qui utilise un gestionnaire d'entités géré par l'application au lieu d'un gestionnaire d'entités étendu.

Annonce 6-12. Utilisation de gestionnaires d'entités gérés par des applications avec JTA

```
@Stateful
Public class DepartmentManager {
      @PersistenceUnit (unitName = "EmployeeService")
      EntityManagerFactory emf;
      EntityManager em;
      Département départemental;

      public void init (int deptId) {
            em = emf.createEntityManager ();
            dept = em.find (Department.class, deptId);
      }

      public String getName () {
            return dept.getName ();
```

```
    }
    public void addEmployee (int empId) {
        em.joinTransaction ();
        Employé emp = em.find (Employee.class, empId);
        dept.getEmployees (). add (emp);
        emp.setDepartment (dept);
    }

    // ...

    @Retirer
    public void done () {
        em.close ();
    }
}
```

Au lieu d'injecter un gestionnaire d'entités, nous injectons une fabrique de gestionnaires d'entités. Avant de rechercher l'entité, nous créons manuellement une nouvelle entité gérée par l'application

Épisode 230

gestionnaire utilisant l'usine. Parce que le conteneur ne gère pas son cycle de vie, nous doivent le fermer plus tard lorsque le bean est supprimé lors de l'appel à finish (). Comme le contexte de persistance étendue géré par le conteneur, l'entité Department reste géré après l'appel à init (). Lorsque addEmployee () est appelé, il y a l'étape supplémentaire de appelant joinTransaction () pour notifier au contexte de persistance qu'il doit se synchroniser lui-même avec la transaction JTA actuelle. Sans cet appel, les changements au département ne serait pas vidé dans la base de données lorsque la transaction est validée.

Étant donné que les gestionnaires d'entités gérés par l'application ne se propagent pas, le seul moyen de partager des entités gérées avec d'autres composants consiste à partager l'instance EntityManager. Ceci peut être réalisé en passant le gestionnaire d'entité comme argument à local méthodes ou en stockant le gestionnaire d'entités dans un endroit commun tel qu'une session HTTP ou haricot singleton. Le listing 6-13 montre un servlet créant une application gérée gestionnaire d'entités et l'utiliser pour instancier la classe EmployeeService que nous avons définie au chapitre 2 . Dans ces cas, il faut veiller à ce que l'accès à l'entité manager est fait de manière thread-safe. Alors que les instances EntityManagerFactory sont thread-safe, les instances EntityManager ne le sont pas. De plus, le code d'application ne doit pas appeler joinTransaction () sur le même gestionnaire d'entités dans plusieurs transactions simultanées.

Annonce 6-13. Partage d'un gestionnaire d'entités géré par une application

```
Public class EmployeeServlet étend HttpServlet {
    @PersistenceUnit (unitName = "EmployeeService")
    EntityManagerFactory emf;
    @Resource UserTransaction tx;

    protected void doPost (requête HttpServletRequest,
                           Réponse HttpServletResponse)
            lance ServletException, IOException {
        // ...
        int id = Integer.parseInt (request.getParameter ("id"));
        String name = request.getParameter ("name");
        salaire long = Long.parseLong (request.getParameter ("salaire"));
        tx.begin ();
        EntityManager em = emf.createEntityManager ();
```

Épisode 231


```
            essayez {
                EmployeeService service = nouveau EmployeeService (em);
                service.createEmployee (identifiant, nom, salaire);
            } enfin {
                em.close ();
            }
            tx.commit ();
            // ...
        }
}
```

Référencement 6-13 montre une caractéristique supplémentaire de la gestion par application
gérant d'entité en présence de transactions. Si le contexte de persistance devient
synchronisées avec une transaction, les modifications seront toujours écrites dans la base de données lorsque le
la transaction est validée, même si le gestionnaire d'entités est fermé. Cela permet aux gestionnaires d'entités
à fermer au point où ils sont créés, supprimant ainsi le besoin de s'inquiéter
les fermer après la fin de la transaction. Notez que la fermeture d'une entité gérée par l'application
manager empêche toujours toute utilisation ultérieure du gestionnaire d'entités. Ce n'est que la persistance
contexte qui continue jusqu'à ce que la transaction soit terminée.

Il y a un danger à mélanger plusieurs contextes de persistance dans le même JTA
transaction. Cela se produit lorsque plusieurs contextes de persistance gérés par l'application
se synchroniser avec la transaction ou lorsque la persistance gérée par l'application
les contextes se mélangent aux contextes de persistance gérés par le conteneur. Quand le
transaction commits, chaque contexte de persistance recevra une notification du
gestionnaire de transactions que les modifications doivent être écrites dans la base de données. Cela causera
chaque contexte de persistance à vider.

Que se passe-t-il si une entité avec la même clé primaire est utilisée dans plusieurs
contexte de persistance? Quelle version de l'entité est stockée? La réponse malheureuse
est qu'il n'y a aucun moyen de savoir avec certitude. Le conteneur ne garantit aucune commande
lors de la notification aux contextes de persistance de l'achèvement de la transaction. En conséquence, *il est essentiel pour l'intégrité des données que les entités ne soient jamais utilisées par plus d'un contexte de persistance dans le même transaction. Lors de la conception de votre application, nous vous recommandons de choisir un* ***seul stratégie de contexte de persistance*** *(gérée par conteneur ou gérée par application) et collage à cette stratégie de manière cohérente.*

Épisode 232

Contextes de persistance non synchronisés

Dans des circonstances normales, un gestionnaire d'entités JTA est synchronisé avec le JTA
La transaction et ses modifications d'entité gérée seront enregistrées lors de la validation de la transaction.
Cependant, une exception à cette règle a été illustrée dans le Listing 6-12 lorsqu'une application-
le gestionnaire d'entités gérées créé lors d'une transaction précédente devait être

joint explicitement à une transaction ultérieure à l'aide de l'appel joinTransaction () dans l'ordre pour que ses modifications d'entité gérée soient validées de manière transactionnelle. Il se trouve que il existe une autre option de contexte de persistance qui présente un comportement similaire. Une entité le gestionnaire peut être explicitement spécifié pour avoir un contexte de persistance non synchronisé, l'obligeant à rejoindre manuellement toute transaction JTA à laquelle il souhaite participer.

Avant de décrire comment configurer et utiliser une telle bête, nous devons d'abord fournir une certaine motivation pour son existence en premier lieu. Pourquoi, parmi toutes les variétés de gestionnaires d'entités et contextes de persistance qui les accompagnent, y a-t-il encore un autre degré de paramétrage? La réponse réside dans le cas d'utilisation proverbial «conversationnel», caractérisé par le scénario suivant.

Une application souhaite effectuer un certain nombre d'opérations de persistance sur ce qui est peut-être une période de temps prolongée, mais ne souhaite pas maintenir une seule transaction pendant toute la durée. Ce groupe d'opérations peut être appelé une conversation car il implique interactions multiples entre le client et le serveur. En raison de la gestion des conteneurs démarcation des transactions, plusieurs transactions peuvent avoir été commencées et terminées au cours de la conversation, mais les opérations de persistance ne doivent pas être enrôlés dans les transactions car ils ne devraient pas être persistés jusqu'à la conversation le statut a été déterminé. À un moment donné, la conversation prend fin et le l'application souhaite valider ou annuler toutes les opérations en tant que groupe. À ceci point toutes les modifications contenues dans le contexte de persistance doivent être transactionnelles écrit dans le magasin de données ou supprimé.

Ce scénario pourrait presque être satisfait par une entité JTA gérée par une application directeur sauf pour deux choses. Le premier est que les gestionnaires d'entités gérées par les applications sont automatiquement synchronisées avec la transaction si elles sont créées lors d'une transaction c'est actif. Bien que cela puisse être contourné simplement en s'assurant que le responsable de l'entité est toujours créé en dehors du cadre d'une transaction, l'apparence d'un ostensiblement gestionnaire d'entités JTA synchronisé, qui en raison d'une technicité n'arrive pas à être synchronisé, n'est pas une solution très propre au cas d'utilisation de la conversation. Ajouté à cela le deuxième point, que ce serait vraiment bien pour le modèle de programmation si l'entité gestionnaire pourrait être injecté dans des composants, ou en d'autres termes être un gestionnaire de conteneurs

Épisode 233

gestionnaire d'entité. La solution la plus appropriée était simplement de permettre aux responsables d'entités possibilité d'avoir un contexte de persistance qui n'est synchronisé avec la transaction que s'il est explicitement joint.

Pour obtenir un gestionnaire d'entités géré par conteneur avec une persistance non synchronisée context, une valeur de UNSYNCHRONIZED, une constante d'énumération SynchronizationType, peut être fourni dans l'élément de synchronisation de l'annotation @PersistenceContext.

```
@PersistenceContext (unitName = "EmployeeService",
                                synchronisation = NON SYNCHRONISÉ)
EntityManager em;
```

Un gestionnaire d'entités géré par une application peut être créé par programme avec un contexte de persistance non synchronisé en passant la même valeur UNSYNCHRONIZED au surchargé EntityManagerFactory méthode createEntityManager ().

```
@PersistenceUnit (unitName = "EmployeeService")
EntityManagerFactory emf;
...
EntityManager em = emf.createEntityManager (UNSYNCHRONIZED);
```

Aucun de ces contextes de persistance ne sera synchronisé avec la transaction JTA à moins qu'ils ne soient explicitement joints à l'aide de joinTransaction (). Une fois joint à la transaction, ils resteront joints jusqu'à ce que cette transaction soit terminée, mais étant être joint à une transaction n'implique pas d'être joint à des transactions ultérieures.

Un contexte de persistance non synchronisé doit être explicitement joint à chaque
transaction dans laquelle il souhaite être enrôlé.

Compte tenu du cas d'utilisation de la conversation et de la possibilité de transactions multiples
se produisant en cours de route, cela a clairement le plus de sens pour la persistance non synchronisée
contextes à utiliser avec les gestionnaires d'entités étendus. La persistance, la suppression et l'actualisation
les opérations peuvent alors être exécutées à l'intérieur ou à l'extérieur de contextes transactionnels, ce qui
il est plus facile de mettre en file d'attente les changements de contexte de persistance avant de finalement rejoindre une transaction
pour les faire écrire.

Remarque Lorsqu'un contexte de persistance non synchronisé n'a pas été joint à un transaction, pas d'écritures dans la base de données, telles que celles résultant d'un flush () peut se produire. Les tentatives de le faire entraîneront la levée d'une exception.

Épisode 234

Pour un exemple simple de contexte de persistance non synchronisé, revenez sur le
exemple de panier (voir liste 3-25 et liste 3-26 au chapitre3). Nous définissons le
L'attribut de transaction doit être facultatif pour la méthode addItems () du ShoppingCart
bean car une transaction n'était pas requise pour ajouter les éléments. C'était bien pour un assez
exemple simpliste. Nous n'avons pas eu besoin de conserver quoi que ce soit ou d'enregistrer les
entités à un contexte de persistance dans le cadre des interactions utilisateur. Prolonger cet exemple
pour utiliser la persistance pour créer / modifier / supprimer des entités indépendantes, nous pourrions utiliser un
contexte de persistance étendue non synchronisé. Le code mis à jour avec quelques
méthodes est montré dans la liste 6-14.

Liste 6-14. Utilisation d'un contexte de persistance non synchronisé

```
@Stateful
ShoppingCart de classe publique {
    @PersistenceContext (unitName = "productInventory",
                         type = ÉTENDU,
                         synchronisation = NON SYNCHRONISÉ)
    EntityManager em;
    Commande CustomerOrder;

    public void addItem (String itemName, Nombre entier) {
        if (ordre == null) {
            order = new CustomerOrder (); em.persist (ordre);
        }
        OrderItem item = order.getItem (itemName);
        if (item == null) {
            item = new OrderItem (itemName);
            item.setOrder (ordre);
            order.addItem (élément);
            em.persist (élément);
        }
        item.setQuantity (item.getQuantity () + quantité) ;;
    }

    @Retirer
    processus public void () {
```

Épisode 235


```
            // Traite la commande. Rejoignez le tx et nous avons terminé.
              em.joinTransaction ();
        }

        @Retirer
        public void cancel () {
              em.clear ();
        }

        // ...
}
```

Il y a quelques éléments intéressants dans la liste 6-14 mérite d'être souligné. Le premier est que le
L'opération process () est triviale. Tout ce qu'il fait est d'appeler joinTransaction () pour que l'actif
la transaction gérée par le conteneur entraînera les modifications du contexte de persistance
écrit lorsque la méthode se termine. La seconde est que la méthode cancel () efface le
contexte de persistance. Bien que inutile dans ce cas, puisque le bean sera supprimé à
ce point de toute façon, nous sommes couverts au cas où nous déciderions d'utiliser le haricot pendant une période plus longue
de temps et supprimez l'annotation @Remove.

Plus tôt dans le chapitre, nous avons discuté de la propagation des contextes de persistance et de la manière dont ils
se propage avec la transaction JTA. La même règle est vraie avec non synchronisé
contextes de persistance. Indépendamment du fait que le contexte de persistance a été joint ou non
à la transaction ou non, le contexte de persistance non synchronisé sera propagé
lorsque la transaction JTA est propagée. Il existe cependant une exception à cette règle
pour les contextes de persistance non synchronisés. Un contexte de persistance non synchronisé,
qu'il soit joint ou non, il n'est jamais propagé dans un synchronisé.

Remarque L'option de contexte de persistance non synchronisée a été introduite dans JPA 2.1.

Transactions de ressources locales

Les transactions locales aux ressources sont contrôlées explicitement par l'application. L'application
le serveur, s'il y en a un, ne participe pas à la gestion de la transaction. Applications
interagir avec les transactions locales des ressources en acquérant une implémentation du
javax.persistence.EntityTransaction du gestionnaire d'entités. le
La méthode getTransaction () de l'interface EntityManager est utilisée à cet effet.

Épisode 236

L'interface EntityTransaction est conçue pour imiter l'UserTransaction
interface définie par JTA, et les deux se comportent de manière très similaire. La principale différence est que
Les opérations EntityTransaction sont implémentées en termes de méthodes de transaction
sur l'interface de connexion JDBC. Le Listing 6-15 montre l'entityTransaction complète
interface.

Liste 6-15. L'interface EntityTransaction

```
interface publique EntityTransaction {
      public void begin ();
```

```
    public void commit ();
    public void rollback ();
    public void setRollbackOnly ();
    public booléen getRollbackOnly ();
    public booléen isActive ();
}
```

Il existe seulement six méthodes sur l'interface EntityTransaction. le
La méthode begin () démarre une nouvelle transaction de ressources. Si une transaction est active,
isActive () retournera true. Tenter de démarrer une nouvelle transaction pendant une transaction
est active entraînera la levée d'une IllegalStateException. Une fois actif, le
transaction peut être validée en appelant commit () ou annulée en invoquant
retour en arriere(). Les deux opérations échoueront avec une IllegalStateException s'il n'y a pas
transaction active. Une PersistenceException sera lancée si une erreur se produit pendant
rollback, tandis qu'une RollbackException, une sous-classe PersistenceException, sera lancée
pour indiquer que la transaction a été annulée en raison d'un échec de validation.

Si une opération de persistance échoue alors qu'une EntityTransaction est active, le fournisseur
le marquera pour la restauration. Il est de la responsabilité de l'application de s'assurer que la restauration
se produit en fait en appelant rollback (). Si la transaction est marquée pour la restauration, et un
commit est tenté, une RollbackException sera lancée. Pour éviter cette exception,
la méthode getRollbackOnly () peut être appelée pour déterminer si la transaction est
dans un état défaillant. Tant que la transaction n'est pas annulée, elle est toujours active et entraînera
la validation ultérieure ou l'opération de démarrage échouent.

Référencement 6-16 montre une application Java SE qui utilise l'API EntityTransaction pour
effectuer un changement de mot de passe pour les utilisateurs qui n'ont pas réussi à mettre à jour leurs mots de passe avant
expiré.

Épisode 237

Annonce 6-16. Utilisation de l'interface EntityTransaction

```
public class ExpirePasswords {
    public static void main (String [] args) {
        int maxAge = Integer.parseInt (args [0]);
        Chaîne defaultPassword = args [1];

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory ("admin");
        essayez {
            EntityManager em = emf.createEntityManager ();

            Calendrier cal = Calendar.getInstance ();
            cal.add (Calendar.DAY_OF_YEAR, -maxAge);

            em.getTransaction (). begin ();
            Collection expirée =
                em.createQuery ("SELECT u FROM User u WHERE
                u.lastChange <=? 1 ")
                    .setParameter (1, cal)
                    .getResultList ();
            for (Iterator i = expired.iterator (); i.hasNext ();) {
                Utilisateur u = (Utilisateur) i.next ();
                System.out.println ("Mot de passe expirant pour" + u.getName ());
                u.setPassword (defaultPassword);
            }
            em.getTransaction (). commit ();
```

```
            em.close ();
        } enfin {
            emf.close ();
        }
    }
}
```

Dans le serveur d'applications, la gestion des transactions JTA est la valeur par défaut et doit être utilisé par la plupart des applications. Un exemple d'utilisation de transactions locales aux ressources dans Java L'environnement EE peut être utilisé pour la journalisation. Si votre application nécessite un journal d'audit stocké dans la base de données qui doit être écrite quel que soit le résultat des transactions JTA,

Épisode 238

un gestionnaire d'entités locales aux ressources peut être utilisé pour conserver les données en dehors de transaction. Les transactions sur les ressources peuvent être lancées et validées librement fois dans une transaction JTA sans affecter l'état des transactions JTA.

Référencement 6-17 montre un exemple de bean session sans état qui fournit une journalisation d'audit cela réussira même si la transaction JTA active échoue.

Annonce 6-17. Utilisation de transactions locales de ressources dans l'environnement Java EE

```
@Apatride
public class LogService {
    @PersistenceUnit (unitName = "journalisation")
    EntityManagerFactory emf;

    public void logAccess (int userId, String action) {
        EntityManager em = emf.createEntityManager ();
        essayez {
            LogRecord lr = nouveau LogRecord (userId, action);
            em.getTransaction (). begin ();
            em.persist (lr);
            em.getTransaction (). commit ();
        } enfin {
            em.close ();
        }
    }
}
```

Bien sûr, vous pouvez faire valoir que c'est exagéré pour une simple journalisation haricot. Direct JDBC fonctionnerait probablement aussi facilement, mais ces mêmes enregistrements de journal peuvent ont des utilisations ailleurs dans l'application. C'est un compromis dans la configuration (définir un unité de persistance complètement séparée afin d'activer les transactions locales aux ressources) par rapport à la commodité d'avoir une représentation orientée objet d'un enregistrement de journal.

Annulation de transaction et état de l'entité

Lorsqu'une transaction de base de données est annulée, toutes les modifications apportées pendant la transaction sont abandonnés. La base de données revient à l'état dans lequel elle se trouvait avant la transaction a commencé. Mais comme mentionné au chapitre2 , le modèle de mémoire Java n'est pas transactionnel.

Épisode 239

Il n'y a aucun moyen de prendre un instantané de l'état de l'objet et d'y revenir plus tard si quelque chose se passe faux. L'une des parties les plus difficiles de l'utilisation d'une solution de mappage relationnel objet est que tandis que vous pouvez utiliser la sémantique transactionnelle dans votre application pour contrôler si les données sont engagé dans la base de données, vous ne pouvez pas vraiment appliquer les mêmes techniques à la mémoire contexte de persistance qui gère vos instances d'entité.

Chaque fois que vous travaillez avec des modifications qui doivent être conservées dans la base de données à à un moment précis, vous travaillez avec un contexte de persistance synchronisé avec une transaction. À un moment donné de la durée de la transaction, généralement juste avant commits, les modifications dont vous avez besoin seront traduites dans les instructions SQL appropriées et envoyé à la base de données. Que vous utilisiez des transactions JTA ou des ressources locales les transactions ne sont pas pertinentes. Vous disposez d'un contexte de persistance participant à une transaction avec les changements qui doivent être apportés.

Si cette transaction est annulée, deux choses se produisent. Le premier est que la base de données la transaction sera annulée. La prochaine chose qui se produit est que la persistance le contexte est effacé, détachant toutes nos instances d'entité gérées. Si le contexte de persistance était à portée de transaction, il est supprimé.

Étant donné que le modèle de mémoire Java n'est pas transactionnel, il vous reste fondamentalement un tas d'instances d'entités détachées. Plus important encore, ces instances détachées reflètent l'état de l'entité exactement tel qu'il était au moment où la restauration s'est produite. Face à une transaction annulée et des entités détachées, vous pourriez être tenté de démarrer une nouvelle transaction, fusionnez les entités dans le nouveau contexte de persistance et recommencez. le Les problèmes suivants doivent être pris en compte dans ce cas:

- S'il existe une nouvelle entité qui utilise la génération automatique de clé primaire, il peut y avoir une valeur de clé primaire attribuée à l'entité détachée. Si cette clé primaire a été générée à partir d'une séquence ou d'une table de base de données, l'opération de génération du numéro a peut-être été annulée avec la transaction. Cela signifie que le même numéro de séquence pourrait être redistribuée à un objet différent. Effacer la clé primaire avant de tenter à nouveau de conserver l'entité, et ne comptez pas sur le valeur de clé primaire dans l'entité détachée.
- Si votre entité utilise un champ de version à des fins de verrouillage, maintenu automatiquement par le fournisseur de persistance, il peut être défini sur une valeur incorrecte. La valeur de l'entité ne correspondra pas au valeur correcte stockée dans la base de données. Nous couvrons le verrouillage et le versionnage dans le chapitre 12.

Épisode 240

Si vous avez besoin de réappliquer certaines des modifications qui ont échoué et sont actuellement les entités détachées, envisagez de copier sélectivement les données modifiées dans de nouvelles entités. Cela garantit que l'opération de fusion ne sera pas compromise par des données périmées laissé dans l'entité détachée. Pour fusionner des entités ayant échoué dans un nouveau contexte de persistance, certains les fournisseurs peuvent proposer des options supplémentaires qui évitent certains ou tous ces problèmes. Le coffre-fort et L'approche sûre consiste à s'assurer que les limites de la transaction sont suffisamment bien définies pour en cas d'échec, la transaction peut être réessayée, y compris la récupération de tous les états gérés et réappliquer les opérations transactionnelles.

Un dernier point à mentionner est que les annulations n'ont aucun effet direct sur la persistance contextes qui ne sont pas synchronisés avec la transaction annulée. Les changements les contextes de persistance non synchronisés ne sont pas transactionnels et sont donc à l'abri des échecs de transaction. On dit surtout immunisé parce qu'il y a un cas, décrit dans le paragraphe suivant, quand il peut être effectué.

Si une unité de persistance est configurée pour accéder à une source de données transactionnelle (par exemple, par définissant l'élément jta-data-source dans persistence.xml), cette source de données sera utilisé pour lire les entités de la base de données, même par les gestionnaires d'entités avec des contextes de persistance. Si une transaction est active et que des modifications ont été apportées à une entité par un autre gestionnaire d'entités indépendant avec un contexte de persistance synchronisé, alors ces changements d'entité seront visibles par, et éventuellement lus dans, le non synchronisé contexte de persistance via une connexion à partir de la source de données transactionnelles. Si la la transaction est annulée, les modifications d'entité non validées seront laissées dans le contexte de persistance non synchronisé.

La situation que nous venons de décrire est un cas de coin assez improbable. Cependant, si vous pensez que ce cas de coin pourrait s'appliquer à vous, il existe un remède. Vous pouvez définir un non-source de données transactionnelles dans le serveur et référencez-la dans l'unité de persistance (en définissant l'élément non-jta-data-source dans persistence.xml aux données non transactionnelles source) en plus de la transaction, comme indiqué ici:

```
<persistance>
    <persistence-unit name = "EmployeeService">
        <jta-data-source> jdbc / EmployeeJtaDS </jta-data-source>
        <non-jta-data-source> jdbc / EmployeeNonJtaDS </non-jta-data-source>
    </persistence-unit>
</persistence>
```


Épisode 241

Le contexte de persistance non synchronisé émettrait alors des requêtes via connexions à partir de la source de données non transactionnelle, étant ainsi isolée de tout modifications transactionnelles simultanées effectuées par d'autres. Uniquement lorsque le contexte de persistance est devenu synchronisé avec une transaction allait-il continuer à utiliser les données transactionnelles la source.

Choisir un gestionnaire d'entités

Avec tous les différents types de gestionnaires d'entités, chacun avec un cycle de vie différent et règles sur l'association et la propagation des transactions, tout cela peut être un peu écrasant. Quel style convient à votre application? Géré par application ou par conteneur? Transaction-étendue ou étendue? Synchronisé ou non synchronisé?

De manière générale, nous pensons que la gestion des conteneurs, la portée des transactions les gestionnaires d'entités seront un modèle très pratique et approprié pour de nombreux applications. C'est le design qui a inspiré JPA à l'origine et c'est le modèle qui les fournisseurs de persistance commerciale utilisent depuis des années. La sélection de ce style pour être la valeur par défaut pour les applications Java EE n'était pas un accident. Il offre la meilleure combinaison de propagation flexible des transactions avec une sémantique facile à comprendre.

Les contextes de persistance étendue gérés par conteneur offrent une programmation différente modèle, les entités restant gérées après la validation, mais elles sont liées au cycle de vie de un composant Java EE - dans ce cas, le bean session avec état. Il y a des de nouvelles techniques possibles avec le contexte de persistance étendue (dont certaines décrites plus loin dans ce chapitre), mais elles peuvent ne pas s'appliquer à toutes les applications.

Dans certaines applications d'entreprise, les gestionnaires d'entités gérées par les applications peuvent à utiliser s'ils doivent être accédés par des classes non managées. Le manque de moyens de propagation qu'ils doivent être transmis comme arguments de méthode ou stockés dans un objet partagé dans afin de partager le contexte de persistance. Évaluer les gestionnaires d'entités gérés par des applications en fonction de vos besoins transactionnels attendus, de la taille et de la complexité de votre application.

Plus que tout, nous vous recommandons d'essayer d'être cohérent dans la façon dont l'entité
les gestionnaires sont sélectionnés et appliqués. Mélanger différents types de gestionnaires d'entités dans un
l'application sera difficile à comprendre et à suivre pour un responsable d'application,
et sera probablement frustrant à déboguer car les différents types de gestionnaires d'entités peuvent
se croisent de manière inattendue.

Épisode 242

Opérations du gestionnaire d'entités

Armé d'informations sur les différents types de gestionnaires d'entités et leur fonctionnement
avec des contextes de persistance, nous pouvons maintenant revoir les opérations de base du gestionnaire d'entités
nous avons présenté dans le chapitre 2 et révéler plus de détails. Les sections suivantes
décrire les opérations de Entity Manager par rapport aux différents Entity Manager
et les types de contexte de persistance. Modes de verrouillage et variantes de verrouillage des éléments suivants
les opérations sont examinées au chapitre 12 .

Persistance d'une entité

La méthode persist () de l'interface EntityManager accepte une nouvelle instance d'entité
et le fait devenir géré. Si l'entité à persister est déjà gérée par
le contexte de persistance, il est ignoré. L'opération contains () peut être utilisée pour vérifier
si une entité est déjà gérée, mais il est très rare que cela soit requis. Il
ne devrait pas surprendre l'application pour savoir quelles entités sont gérées
et qui ne le sont pas. La conception de l'application dicte à quel moment les entités deviennent
géré.

Le fait qu'une entité soit gérée ne signifie pas qu'elle est conservée dans la base de données à droite
une façon. Le SQL réel pour créer les données relationnelles nécessaires ne sera pas généré avant
le contexte de persistance est synchronisé avec la base de données, généralement uniquement lorsque le
la transaction est validée. Cependant, une fois qu'une nouvelle entité est gérée, toute modification apportée à cette entité
peut être suivi par le contexte de persistance. Quel que soit l'état existant sur l'entité lorsque le
transaction commits est ce qui sera écrit dans la base de données.

Lorsque persist () est invoqué en dehors d'une transaction, le comportement dépend
sur le type de gestionnaire d'entités. Un gestionnaire d'entités à portée transaction lancera un
TransactionRequiredException car aucun contexte de persistance n'est disponible dans
pour rendre l'entité gérée. Gestionnaires d'entités étendus et gérés par les applications
acceptera la demande persistante, ce qui entraînera la gestion de l'entité, mais aucun
une action sera entreprise jusqu'à ce qu'une nouvelle transaction commence et que le contexte de persistance devienne
synchronisé avec la transaction. En effet, cela met en file d'attente le changement pour qu'il se produise à un
plus tard. Ce n'est que lorsque la transaction est validée que les modifications seront écrites dans le
base de données.

L'opération persist () est destinée aux nouvelles entités qui ne
existent dans la base de données. Si le fournisseur détermine immédiatement que ce n'est pas vrai,
une EntityExistsException sera levée. Si le fournisseur ne fait pas cela

Épisode 243

détermination (car il a différé le contrôle d'existence et l'insert jusqu'à ce qu'il
commit time), et la clé primaire est en fait un doublon, une exception sera levée
lorsque le contexte de persistance est synchronisé avec la base de données.

Jusqu'à présent, nous avons discuté de la persistance des entités uniquement sans
des relations. Mais, comme nous l'avons appris au chapitre4 , JPA prend en charge une grande variété de relations
les types. En pratique, la plupart des entités sont en relation avec au moins une autre entité.
Considérez la séquence d'opérations suivante:

```
Département dept = em.find (Department.class, 30);
Employé emp = nouvel employé ();
emp.setId (53);
emp.setName ("Peter");
emp.setDepartment (dept);
dept.getEmployees (). add (emp);
em.persist (emp);
```

Malgré la brièveté de cet exemple, nous avons abordé de nombreux points relatifs à
persister dans une relation. Nous commençons par récupérer une instance Department préexistante. UNE
une nouvelle instance Employee est alors créée, fournissant la clé primaire et les informations de base
à propos de l'employé. Nous affectons ensuite l'employé au service, en définissant le
l'attribut department de l'employé pour pointer vers l'instance Department que nous avons récupérée
plus tôt. Comme la relation est bidirectionnelle, nous ajoutons ensuite la nouvelle instance Employee
à la collection des employés dans l'instance Department. Enfin le nouvel employé
instance est persistée avec l'appel à persist (). En supposant qu'une transaction soit validée,
et le contexte de persistance y est synchronisé, la nouvelle entité sera stockée dans le
base de données.

Une chose intéressante à propos de cet exemple est que le ministère est un
participant malgré l'ajout de l'instance Employee à sa collection. L'employé
l'entité est le propriétaire de la relation car elle est dans une relation plusieurs-à-un avec
le département. Comme nous l'avons mentionné au chapitre4, le côté source de la relation est
le propriétaire, tandis que la cible est l'inverse dans ce type de relation. Quand l'employé
persiste, la clé étrangère du département est écrite dans la table mappée
par l'Employé, et aucune modification réelle n'est apportée au
représentation. Si nous avions seulement ajouté l'employé à la collection et pas mis à jour le
de l'autre côté de la relation, rien n'aurait été conservé dans la base de données.

Épisode 244

Trouver une entité

La méthode toujours présente find () est la bête de somme du gestionnaire d'entités. Chaque fois qu'un
l'entité doit être localisée par sa clé primaire, find () est généralement la meilleure solution. ne pas
seulement il a une sémantique simple, mais la plupart des fournisseurs de persistance optimiseront également cela
opération pour utiliser un cache en mémoire qui minimise les déplacements vers la base de données.

L'opération find () renvoie une instance d'entité gérée dans tous les cas sauf lorsque
invoqué en dehors d'une transaction sur un gestionnaire d'entités à portée transaction. Dans ce cas, le
l'instance d'entité est renvoyée dans un état détaché. Il n'est associé à aucune persistance
le contexte.

Il existe une version spéciale de find () qui peut être utilisée dans une situation particulière.
Cette situation se produit lorsqu'une relation est créée entre deux entités dans un
relation un ou plusieurs à un dans laquelle l'entité cible existe déjà et sa principale
la clé est bien connue. Parce que nous ne faisons que créer une relation, cela peut ne pas être nécessaire
pour charger complètement l'entité cible afin de créer la référence de clé étrangère vers elle. Seulement sa clé primaire
est requis. L'opération getReference () peut être utilisée à cette fin. Prendre en compte
exemple suivant:

```
Département dept = em.getReference (Department.class, 30);
Employé emp = nouvel employé ();
emp.setId (53);
emp.setName ("Peter");
emp.setDepartment (dept);
dept.getEmployees (). add (emp);
em.persist (emp);
```

La seule différence entre cette séquence d'opérations et celles que nous démontré précédemment est que l'appel find () a été remplacé par un appel à getReference (). Lorsque l'appel getReference () est appelé, le fournisseur peut renvoyer un proxy vers l'entité Department sans la récupérer dans la base de données. Aussi longtemps que seule sa clé primaire est accessible, les données du service n'ont pas besoin d'être extraites. Au lieu, lorsque l'employé est conservé, la valeur de la clé primaire sera utilisée pour créer le touche à l'entrée Département correspondante. L'appel getReference () est en fait un l'optimisation des performances qui supprime le besoin de récupérer l'instance d'entité cible.

Il y a quelques inconvénients à utiliser getReference () qui doivent être compris. le tout d'abord, si un proxy est utilisé, il peut lever une exception EntityNotFoundException s'il est incapable de localiser l'instance d'entité réelle lorsqu'un attribut autre que la clé primaire

Épisode 245

est accédé. L'hypothèse avec getReference () est que vous êtes sûr que l'entité avec le la clé primaire correcte existe. Si, pour une raison quelconque, un attribut autre que la clé primaire est accédé, et l'entité n'existe pas, une exception sera levée. Un corollaire à ceci est que l'objet retourné par getReference () pourrait ne pas être sûr à utiliser s'il ne l'est plus géré. Si le fournisseur renvoie un proxy, cela dépendra de l'existence d'un contexte de persistance pour charger l'état de l'entité.

Étant donné la situation très spécifique dans laquelle getReference () peut être utilisé, find () doit être utilisé dans pratiquement tous les cas. Le cache en mémoire d'une bonne persistance le fournisseur est suffisamment efficace pour que le coût de performance de l'accès à une entité via sa clé primaire ne sera généralement pas remarquée. Dans le cas d'EclipseLink, il dispose d'un cache d'objets partagés intégré, donc non seulement la gestion du contexte de persistance locale efficace mais aussi tous les threads sur le même serveur peuvent bénéficier du contenu partagé de le cache. L'appel getReference () est une optimisation des performances qui doit être utilisée seulement lorsqu'il y a des preuves suggérant que cela profitera réellement à l'application.

Suppression d'une entité

La suppression d'une entité n'est pas une tâche complexe, mais elle peut nécessiter plusieurs étapes selon le nombre de relations dans l'entité à supprimer. Dans sa forme la plus élémentaire, supprimer un entity est simplement un cas de passage d'une instance d'entité gérée à la méthode remove () de un gestionnaire d'entité. Dès que le contexte de persistance associé est synchronisé avec une transaction et des validations, l'entité est supprimée. Au moins c'est ce que nous voudrions aiment arriver. Comme nous le montrerons bientôt, supprimer une entité nécessite une certaine attention relations, ou bien l'intégrité de la base de données peut être compromise dans le processus.

Passons en revue un exemple simple. Considérez l'employé et l'espace de stationnement relation que nous avons démontrée au chapitre 4 . L'employé a un unidirectionnel relation individuelle avec l'entité ParkingSpace. Imaginez maintenant que nous exécutons le code suivant dans une transaction, où empId correspond à une clé primaire Employee:

```
Employé emp = em.find (Employee.class, empId);
em.remove (emp.getParkingSpace ());
```

Lorsque la transaction est validée, nous voyons l'instruction DELETE pour PARKING_SPACE table est générée, mais nous obtenons une exception contenant une erreur de base de données qui montre

que nous avons violé une contrainte de clé étrangère. Il s'avère qu'une intégrité référentielle
une contrainte existe entre la table EMPLOYEE et la table PARKING_SPACE. La ligne

Épisode 246

a été supprimée de la table PARKING_SPACE, mais la clé étrangère correspondante dans le
La table EMPLOYEE n'a pas été définie sur NULL. Pour corriger le problème, nous devons définir explicitement le
L'attribut parkingSpace de l'entité Employee a la valeur null avant la validation de la transaction.

```
Employé emp = em.find (Employee.class, empId);
ParkingSpace ps = emp.getParkingSpace ();
emp.setParkingSpace (null);
em.remove (ps);
```

Le maintien de la relation est à la charge de l'application. Nous répétons ceci
déclaration au cours de ce livre, mais elle ne saurait être suffisamment soulignée. Presque
chaque problème lié à la suppression d'une entité revient toujours à ce problème. Si l'entité
à supprimer est la cible des clés étrangères dans d'autres tables, ces clés étrangères doivent être
effacé pour que la suppression réussisse. L'opération de suppression échouera comme ici ou
il en résultera des données obsolètes dans les colonnes de clé étrangère faisant référence à la suppression
en cas d'absence d'intégrité référentielle.

Une entité ne peut être supprimée que si elle est gérée par un contexte de persistance. Ça signifie
qu'un gestionnaire d'entités à portée transaction ne peut être utilisé pour supprimer une entité que s'il y a
une transaction active. Tenter d'invoquer remove () alors qu'il n'y a pas de transaction sera
résultent en une exception TransactionRequiredException. Comme l'opération persist ()
décrit précédemment, les gestionnaires d'entités étendus et gérés par l'application peuvent supprimer un
entité en dehors d'une transaction, mais le changement n'aura pas lieu dans la base de données jusqu'à ce qu'un
transaction, avec laquelle le contexte de persistance est synchronisé, est validée.

Une fois la transaction validée, toutes les entités supprimées
les transactions sont laissées dans l'état dans lequel elles se trouvaient avant d'être supprimées. A supprimé
l'instance d'entité peut être persistante à nouveau avec l'opération persist (), mais les mêmes problèmes
avec l'état généré dont nous avons discuté dans la section "Annulation de transaction et état de l'entité"
section s'appliquent également ici.

Opérations en cascade

Par défaut, chaque opération de gestionnaire d'entités s'applique uniquement à l'entité fournie en tant que
argument de l'opération. L'opération ne se répercutera pas sur d'autres entités qui ont
une relation avec l'entité exploitée. Pour certaines opérations, telles que
remove (), c'est généralement le comportement souhaité. Nous ne voudrions pas que le gestionnaire d'entité
faire des hypothèses incorrectes sur les instances d'entité à supprimer en tant que côté

Épisode 247

effet d'une autre opération. Mais il n'en va pas de même pour des opérations telles que
persister(). Il y a de fortes chances que si nous avons une nouvelle entité et qu'elle a une relation avec une autre
nouvelle entité, les deux doivent être persistés ensemble.

Considérez la séquence d'opérations du Listing 6-18 qui sont nécessaires pour créer un

nouvelle entité Employee avec une entité Address associée et rendez les deux persistants.
Le deuxième appel à persist () qui rend l'entité Address gérée est gênant.
Une entité Address est couplée à l'entité Employee qui la conserve. Chaque fois qu'un nouveau
L'employé est créé, il est logique de mettre en cascade l'opération persist () à l'adresse
entité si elle est présente. Dans le Listing 6-18, nous sommes en cascade manuellement au moyen d'un
persist () appelle l'adresse associée.

Liste 6-18. Entités d'employé et d'adresse persistantes

```
Employé emp = nouvel employé ();
emp.setId (2);
emp.setName ("Rob");
Adresse addr = nouvelle adresse ();
addr.setStreet ("645 Stanton Way");
addr.setCity ("Manhattan");
addr.setState ("NY");
emp.setAddress (addr);
em.persist (addr);
em.persist (emp);
```

Heureusement, JPA fournit un mécanisme pour définir quand des opérations telles que persist ()
doit être automatiquement mis en cascade entre les relations. L'attribut cascade, dans
toutes les annotations de relations logiques (@OneToOne, @OneToMany, @ManyToOne et @
ManyToMany), définit la liste des opérations du gestionnaire d'entités à mettre en cascade.

Les opérations du gestionnaire d'entités sont identifiées à l'aide du type énuméré CascadeType
lorsqu'il est répertorié dans le cadre de l'attribut cascade. Le PERSIST, REFRESH, REMOVE, MERGE,
et les constantes DETACH appartiennent à l'opération de gestionnaire d'entités du même nom. le
constant ALL est un raccourci pour déclarer que les cinq opérations doivent être en cascade. Par
par défaut, les relations ont un ensemble de cascade vide.

Les sections suivantes définiront le comportement en cascade de persist () et
opérations remove (). Nous introduisons les opérations detach () et merge () et leurs
comportement en cascade plus loin dans ce chapitre dans la section «Fusion d'entités détachées».
De même, nous introduisons l'opération refresh () et son comportement en cascade dans le chapitre 12.

Épisode 248

Cascade persiste

Pour commencer, considérons les changements nécessaires pour faire en cascade l'opération persist ()
de l'employé à l'adresse. Dans la définition de la classe Employee, il y a un @ManyToOne
annotation définie pour la relation d'adresse. Pour activer la cascade, il faut ajouter le
Opération PERSIST à la liste des opérations en cascade pour cette relation. Annonce 6-19
montre un fragment de l'entité Employee qui illustre ce changement.

Liste 6-19. Activation de la persistance en cascade

```
@Entité
Employé de classe publique {
    // ...
    @ManyToOne (cascade = CascadeType.PERSIST)
    Adresse adresse;
    // ...
}
```

Pour tirer parti de ce changement, nous devons simplement nous assurer que l'entité Address a été
défini sur l'instance Employee avant d'appeler persist () sur celle-ci. En tant que responsable d'entité
rencontre l'instance Employee et l'ajoute au contexte de persistance, il naviguera
à travers la relation d'adresse à la recherche d'une nouvelle entité d'adresse à gérer également. Dans

comparaison avec l'approche du Listing 6-18, ce changement nous évite d'avoir à persister l'adresse séparément.

Les paramètres de cascade sont unidirectionnels. Cela signifie qu'ils doivent être explicitement définis sur les deux côtés d'une relation si le même comportement est destiné aux deux situations. Pour exemple, dans l'extrait 6-19, nous avons seulement ajouté le paramètre de cascade à la relation d'adresse dans l'entité Employé. Si liste6-18 ont été modifiés pour ne conserver que l'entité Address, pas l'entité Employé, l'entité Employé ne deviendrait pas gérée car l'entité le gestionnaire n'a pas été invité à sortir des relations définies sur le Entité d'adresse.

Même s'il est légal de le faire, il est encore peu probable que nous ajoutions la mise en cascade opérations de l'entité Address vers l'entité Employee, car il s'agit d'un enfant de Entité des employés. Tout en faisant en sorte que l'instance Employee soit gérée en tant que côté l'effet de la persistance de l'instance Address est inoffensif, le code d'application ne s'attend pas la même chose pour l'opération remove (), par exemple. Par conséquent, nous devons être judicieux

Épisode 249

en appliquant des cascades parce qu'il y a une attente d'appropriation dans les relations qui influence les attentes des développeurs lorsqu'ils interagissent avec ces entités.

Dans la section "Persistance d'une entité", nous avons mentionné que l'instance d'entité est ignoré s'il est déjà persistant. C'est vrai, mais le gestionnaire d'entité respectera toujours le PERSISTENT en cascade dans cette situation. Par exemple, considérons à nouveau notre entité Employé. Si la L'instance Employee est déjà gérée et une nouvelle instance Address y est définie, appelant persist () à nouveau sur l'instance Employee fera devenir l'instance Address géré. Aucune modification ne sera apportée à l'instance Employee car elle est déjà géré.

Parce que l'ajout de la cascade PERSIST est un comportement très courant et souhaitable pour relations, il est possible d'en faire le paramètre de cascade par défaut pour toutes les relations dans l'unité de persistance. Nous discutons de cette technique au chapitredix.

Supprimer en cascade

À première vue, avoir le gestionnaire d'entités en cascade automatiquement les opérations remove () peut sembler attrayant. Selon la cardinalité de la relation, elle pourrait élimine le besoin de supprimer explicitement plusieurs instances d'entité. Et pourtant, pendant que nous pourrait mettre en cascade cette opération dans un certain nombre de situations, cela ne devrait être appliqué que dans certains cas. Il n'y a vraiment que deux cas dans lesquels la mise en cascade de remove () opération a du sens: relations un-à-un et un-à-plusieurs, dans lesquelles il y a une relation parent-enfant claire. Il ne peut pas être appliqué aveuglément à tous les un-à-un et un-à plusieurs relations car les entités cibles peuvent également participer à d'autres relations ou pourraient avoir un sens en tant qu'entités autonomes. Des précautions doivent être prises lorsque en utilisant l'option REMOVE cascade.

Avec cet avertissement donné, examinons une situation dans laquelle la mise en cascade de remove () l'opération a du sens. Si une entité Employé est supprimée (espérons-le occurrence!), il peut être judicieux de mettre en cascade l'opération remove () à la fois Entités ParkingSpace et Phone liées à l'employé. Ce sont les deux cas dans lesquels l'employé est le parent des entités cibles, ce qui signifie qu'elles ne sont pas référencées par d'autres entités du système. Référencement6-20 montre les modifications apportées à la classe d'entités Employee qui permet ce comportement. Notez que nous avons ajouté la cascade REMOVE en plus de l'option PERSIST existante. Les chances sont, si une relation propriétaire est sûre d'utiliser REMOVE, il est également sûr d'utiliser PERSIST.

Épisode 250

Liste 6-20. Activation de la suppression en cascade

```
@Entité
Employé de classe publique {
    // ...
    @OneToOne (cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    ParkingSpace parkingSpace;
    @OneToMany (mappedBy = "employé",
                cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    Collection de téléphones <Phone>;
    // ...
}
```

Maintenant, prenons un peu de recul et regardons ce que cela signifie de mettre en cascade le remove () opération. Lors du traitement de l'instance Employee, le gestionnaire d'entités naviguera à travers les relations parkingSpace et téléphones et invoquez remove () sur ces entités instances également. Comme l'opération remove () sur une seule entité, il s'agit d'une base de données et n'a aucun effet sur les liens en mémoire entre les instances d'objet.
Lorsque l'instance Employee est détachée, sa collection de téléphones contiendra toujours tous les instances Phone qui étaient présentes avant que l'opération remove () n'ait lieu. Le téléphone les instances sont détachées car elles ont également été supprimées, mais le lien entre les deux les instances restent.

Étant donné que l'opération remove () ne peut être mise en cascade en toute sécurité que d'un parent à un enfant, cela ne peut pas aider la situation rencontrée précédemment dans la section «Suppression d'une entité». Aucun paramètre ne peut être appliqué à une relation d'une entité à une autre cela entraînera sa suppression d'un parent sans supprimer également le parent dans le processus. Par exemple, en essayant de supprimer l'entité ParkingSpace, nous atteignons une intégrité violation de contrainte de la base de données à moins que le champ parkingSpace dans Employee L'entité est définie sur null. Définition de l'option de cascade REMOVE sur l'annotation @OneToOne dans l'entité ParkingSpace n'entraînerait pas sa suppression de l'employé; au lieu, cela entraînerait la suppression de l'instance Employee elle-même. Ce n'est clairement pas le comportement que nous désirons. Il n'y a pas de raccourcis vers la maintenance des relations.

Épisode 251

Effacer le contexte de persistance

Parfois, il peut être nécessaire d'effacer un contexte de persistance de ses entités gérées. Cela n'est généralement requis que pour les contextes de persistance étendue et gérés par l'application qui durent longtemps et sont devenus trop grands. Par exemple, considérons une application-gestionnaire d'entités géré qui émet une requête renvoyant plusieurs centaines d'instances d'entités. Une fois que des modifications sont apportées à une poignée de ces instances et que la transaction est validée, vous avez laissé en mémoire des centaines d'objets que vous n'avez pas l'intention de changer

plus loin. Si vous ne souhaitez pas fermer le contexte de persistance, vous devez pouvoir effacer les entités gérées, sinon le contexte de persistance continuera de croître avec le temps.

La méthode clear () de l'interface EntityManager peut être utilisée pour effacer le contexte de persistance. À bien des égards, cela équivaut sémantiquement à une transaction retour en arriere. Toutes les instances d'entité gérées par le contexte de persistance deviennent détachées avec leur état laissé exactement tel qu'il était lorsque l'opération clear () a été invoquée. Si un la transaction a été lancée à ce stade puis validée, rien ne serait écrit dans la base de données car le contexte de persistance est vide. L'opération clear () est tout ou rien. Annulation sélective de la gestion de toute instance d'entité particulière le contexte de persistance est toujours ouvert est obtenu via l'opération detach (). Nous discutons ceci plus loin dans la section «Détachement et fusion».

Bien que techniquement possible, effacer le contexte de persistance lorsqu'il y a les modifications non validées constituent une opération dangereuse. Le contexte de persistance est un la structure de la mémoire, et l'effacer détache simplement les entités gérées. Si vous êtes une transaction et des modifications ont déjà été écrites dans la base de données, elles ne seront pas annulé lorsque le contexte de persistance est effacé. Les entités détachées qui en résultent du nettoyage du contexte de persistance souffrent également de tous les effets négatifs causés par une annulation de transaction même si la transaction est toujours active. Par exemple, identifiant la génération et la gestion des versions doivent être considérées comme suspectes pour toute entité détachée en tant que résultat de l'utilisation de l'opération clear ().

Synchronisation avec la base de données

Chaque fois que le fournisseur de persistance génère du SQL et l'écrit dans la base de données une connexion JDBC, nous disons que le contexte de persistance a été vidé. Tout en attente changements qui nécessitent une instruction SQL pour faire partie des changements transactionnels dans la base de données a été écrite et sera rendue permanente lorsque la base de données

Épisode 252

la transaction est validée. Cela signifie également que toute opération SQL ultérieure qui a lieu après le rinçage incorporera ces changements. Ceci est particulièrement important pour SQL requêtes exécutées dans une transaction qui modifie également les données d'entité.

S'il existe des entités gérées avec des modifications en attente dans une persistance synchronisée contexte, un flush est garanti dans deux situations. Le premier est lorsque la transaction s'engage. Un vidage de toutes les modifications requises se produira avant que la transaction de base de données n'ait terminé. Le seul autre moment où un vidage est garanti est lorsque le gestionnaire d'entités L'opération flush () est appelée. Cette méthode permet aux développeurs de déclencher manuellement le même processus que le gestionnaire d'entités utilise en interne pour vider le contexte de persistance.

Cela dit, un vidage du contexte de persistance peut se produire à tout moment si la persistance le fournisseur le juge nécessaire. Un exemple de ceci est lorsqu'une requête est sur le point d'être exécutée et cela dépend d'entités nouvelles ou modifiées dans le contexte de persistance. Certains fournisseurs va vider le contexte de persistance pour garantir que la requête intègre tous les changements. Un fournisseur peut également vider souvent le contexte de persistance s'il utilise un rédiger l'approche des mises à jour des entités. La plupart des fournisseurs de persistance reportent la génération SQL au dernier moment possible pour des raisons de performances, mais ce n'est pas garanti.

Maintenant que nous avons couvert les circonstances dans lesquelles un flush peut se produire, regardons exactement ce que signifie vider le contexte de persistance. Un flush se compose essentiellement de trois composants: de nouvelles entités qui doivent être persistantes, des entités modifiées qui ont besoin à mettre à jour et les entités supprimées qui doivent être supprimées de la base de données. Tout ça les informations sont gérées par le contexte de persistance. Il maintient des liens vers tous les les entités qui seront créées ou modifiées ainsi que la liste des entités qui doivent être supprimées.

Lorsqu'un vidage se produit, le gestionnaire d'entités effectue d'abord une itération sur les entités gérées et recherche de nouvelles entités qui ont été ajoutées aux relations avec cascade persist

activée. Cela équivaut logiquement à appeler à nouveau persist () sur chaque
entité juste avant le vidage. Le responsable de l'entité vérifie également l'intégrité
de toutes les relations. Si une entité pointe vers une autre entité qui n'est pas gérée ou qui
été supprimée, une exception peut être levée.

Les règles pour déterminer si le vidage échoue en présence d'un non géré
l'entité peut être compliquée. Passons en revue un exemple qui démontre le plus
problèmes communs. La figure 6-2 montre un diagramme d'objets pour une instance Employee et
des objets auxquels il est lié. Les objets d'entité emp et ps sont gérés par le
contexte de persistance. L'objet addr est une entité détachée d'une transaction précédente,
et les objets Phone sont de nouveaux objets qui n'ont fait partie d'aucune persistance
opération jusqu'à présent.

Épisode 253

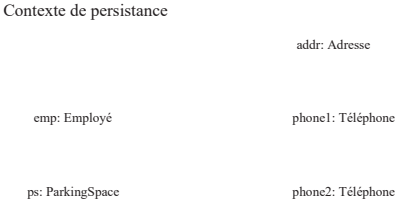

Figure 6-2. *Liens vers des entités non gérées à partir d'un contexte de persistance*

Pour déterminer le résultat du vidage du contexte de persistance compte tenu de la disposition
illustré à la Figure 6-2 , nous devons d'abord examiner les paramètres en cascade de l'entité Employee.
Référencement 6-21 montre les relations implémentées dans l'entité Employé. Seulement le
la relation téléphones a l'option de cascade PERSIST. Les autres relations sont toutes
par défaut, ils ne seront pas en cascade.

Liste 6-21. Paramètres de cascade de relations pour l'employé

```
@Entité
Employé de classe publique {
    // ...
    @Un par un
    ParkingSpace parkingSpace;
    @OneToMany (mappedBy = "employé", cascade = CascadeType.PERSIST)
    Collection de téléphones <Phone>;
    @ManyToOne
    Adresse adresse;
    // ...
}
```

En commençant par l'objet emp, parcourons le processus de vidage comme si nous étions le
fournisseur de persistance. L'objet emp est géré et a des liens vers quatre autres objets.
La première étape du processus consiste à parcourir les relations à partir de cette entité comme si nous étions
en invoquant persist () dessus. Le premier objet que nous rencontrons dans ce processus est l'objet ps
à travers la relation parkingSpace. Parce que ps est également géré, nous n'avons pas à faire
rien de plus.

Épisode 254

Ensuite, nous naviguons dans la relation entre les téléphones et les deux objets Phone. Ces entités sont nouveaux, et cela provoquerait normalement une exception, mais parce que la cascade PERSIST option a été définie, nous effectuons l'équivalent d'appeler persist () sur chaque téléphone objet. Cela rend les objets gérés, en les intégrant au contexte de persistance.
Les objets Téléphone n'ont pas d'autres relations pour mettre en cascade l'opération de persistance, nous avons donc terminé ici aussi.

Ensuite, nous atteignons l'objet addr à travers la relation d'adresse. Parce que cet objet est détaché, nous lancerions normalement une exception, mais cette relation particulière est un cas particulier dans l'algorithme de vidage. Chaque fois qu'un objet détaché qui est la cible du une relation un-à-un ou plusieurs-à-un est rencontrée lorsque l'entité source est le propriétaire, le rinçage continuera parce que l'acte de persistance de l'entité propriétaire ne dépendent de la cible. L'entité propriétaire a la colonne de clé étrangère et doit stocker uniquement la valeur de clé primaire de l'entité cible.

Ceci termine le rinçage de l'objet emp. L'algorithme se déplace ensuite vers l'objet ps et recommence le processus. Parce qu'il n'y a pas de relations entre l'objet ps et aucun autre, le processus de vidage se termine. Donc, dans cet exemple, même si trois des objets pointés depuis l'objet emp ne sont pas gérés, le vidage global se termine avec succès en raison des paramètres de cascade et des règles de l'algorithme de vidage.

Idéalement, lors d'un vidage, tous les objets pointés par une entité gérée seront également les entités gérées elles-mêmes. Si ce n'est pas le cas, la prochaine chose dont nous devons être conscients de est le paramètre de cascade PERSIST. Si la relation a ce paramètre, ciblez les objets dans le la relation sera également persistante, ce qui les rendra gérées avant la fin du vidage.
Si l'option de cascade PERSIST n'est pas définie, une exception IllegalStateException sera jeté chaque fois que la cible de la relation n'est pas gérée, sauf cas particulier liées aux relations un-à-un et plusieurs-à-un que nous avons décrites précédemment.

À la lumière du fonctionnement de l'opération de vidage, il est toujours plus sûr de mettre à jour les relations pointant vers les entités qui seront supprimées avant d'effectuer l'opération remove ().
Un vidage peut se produire à tout moment, donc invoquer remove () sur une entité sans effacer toute relation pointant vers l'entité supprimée peut entraîner un Exception IllegalStateException si le fournisseur décide de vider la persistance contexte avant de mettre à jour les relations.

Dans le chapitre 7 , nous discutons également des techniques pour configurer les exigences d'intégrité des données de requêtes afin que le fournisseur de persistance soit mieux en mesure de déterminer quand un vidage du le contexte de persistance est vraiment nécessaire.

Épisode 255

Détachement et fusion

En termes simples, une *entité détachée* est celle qui n'est plus associée à une persistance le contexte. Il a été géré à un moment donné, mais le contexte de persistance a peut-être pris fin ou l'entité peut avoir été transformée de sorte qu'elle a perdu son association avec le contexte de persistance utilisé pour le gérer. Le contexte de persistance, s'il y en a encore un, ne suit plus l'entité. Les modifications apportées à l'entité ne seront pas conservées la base de données, mais tout l'état qui était présent sur l'entité lorsqu'elle a été détachée peut encore être utilisé par l'application. Une entité détachée ne peut être utilisée avec aucun gestionnaire d'entités opération qui nécessite une instance gérée.

Le contraire du détachement fusionne. La fusion est le processus par lequel une entité manager intègre l'état de l'entité détachée dans un contexte de persistance. Tout changement

à l'état d'entité qui a été fait sur l'entité détachée écrase les valeurs actuelles
dans le contexte de la persistance. Lorsque la transaction est validée, ces modifications seront
persisté. La fusion permet aux entités d'être modifiées «hors ligne», puis d'avoir ces modifications
incorporé plus tard.

Les sections suivantes décrivent le détachement et comment les entités détachées peuvent être
fusionné dans un contexte de persistance.

Détachement

Il y a deux vues du détachement. D'une part, c'est un outil puissant qui peut être
exploité par les applications pour travailler avec des applications distantes ou pour prendre en charge
l'accès aux données d'entité longtemps après la fin d'une transaction. D'un autre côté, cela peut être un
problème frustrant lorsque le modèle de domaine contient de nombreux attributs à chargement différé et
les clients utilisant les entités détachées doivent accéder à ces informations.

Une entité peut se détacher de plusieurs manières. Chacun des éléments suivants
les situations conduiront à des entités détachées:

- Lorsque la transaction qu'une persistance de portée transaction
  contexte est associé aux commits, toutes les entités gérées par le
  le contexte de persistance se détache.
- Si un contexte de persistance géré par une application est fermé, tous ses
  les entités gérées se détachent.

Épisode 256

- Si un bean session avec état avec un contexte de persistance étendu est
  supprimée, toutes ses entités gérées se détachent.
- Si la méthode clear () d'un gestionnaire d'entités est utilisée, elle détache tous les
  entités dans le contexte de persistance géré par ce gestionnaire d'entités.
- Si la méthode detach () d'un gestionnaire d'entités est utilisée, elle détache un
  instance d'entité unique du contexte de persistance géré par cette
  gestionnaire d'entité.
- Lorsque l'annulation de transaction se produit, toutes les entités en tout
  contextes de persistance associés à la transaction à devenir
  détaché.
- Lorsqu'une entité est sérialisée, la forme sérialisée de l'entité est
  détaché de son contexte de persistance.

Certaines de ces situations peuvent être intentionnelles et planifiées, comme le détachement
après la fin de la transaction ou de la sérialisation. D'autres peuvent être inattendus, tels que
détachement en raison de la restauration.

Le détachement explicite d'une entité est réalisé via l'opération detach (). contrairement à
l'opération clear () évoquée précédemment, si une instance d'entité est passée en paramètre, le
L'opération detach () sera limitée à une seule entité et à ses relations. Comme les autres
opérations en cascade, l'opération detach () naviguera également à travers les relations qui
définissez les options de cascade DETACH ou ALL, en détachant les entités supplémentaires le cas échéant.
Notez que passer une entité nouvelle ou supprimée à detach () a un comportement différent d'un
entité gérée normale. L'opération ne détache ni les entités nouvelles ni les entités supprimées,
mais il tentera toujours, lorsqu'il est configuré en cascade, de passer en cascade entre les relations
sur les entités supprimées et détachez toutes les entités gérées qui en sont la cible
des relations.

Au chapitre 4, nous avons introduit le type de récupération LAZY qui peut être appliqué à n'importe quel

cartographie ou relation. Cela a pour effet d'indiquer au fournisseur que le chargement de
un attribut de base ou de relation doit être différé jusqu'à ce qu'il soit accédé pour la première fois.
Bien qu'il ne soit pas couramment utilisé sur les mappages de base, le marquage des mappages de relations
La charge différée est une partie importante du réglage des performances.

Nous devons cependant tenir compte de l'impact du détachement sur le chargement paresseux. Considérer
l'entité Employé indiquée dans le Listing 6-22. La relation d'adresse sera avec impatience
charge parce que les relations plusieurs-à-un se chargent avec impatience par défaut. Dans le cas du
l'attribut parkingSpace, qui se chargerait également normalement avec empressement, nous avons explicitement

Épisode 257

a marqué la relation comme étant paresseusement chargée. La relation téléphonique, en tant que personne
beaucoup de relations, sera également lazy load par défaut.

Liste 6-22. Employé avec mappages à chargement différé

```
@Entité
Employé de classe publique {
    // ...
    @ManyToOne
    adresse d'adresse privée;
    @OneToOne (fetch = FetchType.LAZY)
    Parking privéEspace de stationnementEspace;
    @OneToMany (mappedBy = "employé")
    les téléphones <Phone> de la collection privée;
    // ...
}
```

Tant que l'entité Employé est gérée, tout fonctionne comme prévu. Quand le
l'entité est extraite de la base de données, seule l'entité Address associée sera avec empressement
chargé. Le fournisseur récupérera les entités nécessaires la première fois que le parkingSpace et
les relations téléphoniques sont accessibles.

Si cette entité se détache, le résultat de l'accès au parkingSpace et
les relations téléphoniques sont soudainement un problème plus complexe. Si les relations étaient
accédé alors que l'entité était encore gérée, les entités cibles peuvent également être
accessible lorsque l'entité Employé est détachée. Si les relations n'ont pas été consultées
alors que l'entité était gérée, nous avons un problème.

Le comportement d'accès à un attribut déchargé lorsque l'entité est détachée n'est pas
défini. Certains fournisseurs peuvent tenter de résoudre la relation, tandis que d'autres peuvent
lève simplement une exception ou laisse l'attribut non initialisé. Si l'entité a été détachée
en raison de la sérialisation, il n'y a pratiquement aucun espoir de résoudre la relation. Le seul
la chose portable à faire avec les attributs qui sont déchargés est de les laisser tranquilles. Bien sûr, ce
implique que vous savez quels attributs ont été chargés, et ce n'est pas toujours facile ou
pratique selon l'endroit où se trouve l'entité (voir la méthode isLoaded () dans la section «Utilitaire
Classes »dans le chapitre 12).

Dans le cas où les entités n'ont pas d'attributs de chargement différé, le détachement n'est pas un gros
traiter. Tout l'état d'entité qui était présent dans la version gérée est toujours disponible et prêt
à utiliser dans la version détachée de l'entité. En présence d'attributs de chargement différé,

Épisode 258

il faut veiller à ce que toutes les informations dont vous avez besoin pour accéder hors ligne soient disponible. Lorsque cela est possible, essayez de définir l'ensemble des attributs d'entité détachés qui peuvent être accessible par le composant hors ligne. Le fournisseur des entités doit traiter cet ensemble comme un contrat et l'honorer en déclenchant ces attributs alors que l'entité est toujours gérée.
Plus loin dans le chapitre, nous présentons un certain nombre de stratégies pour planifier et travailler avec, entités détachées, y compris comment provoquer le chargement des attributs déchargés.

Fusion d'entités détachées

L'opération merge () est utilisée pour fusionner l'état d'une entité détachée en une persistance le contexte. La méthode est simple à utiliser, ne nécessitant que l'entité détachée instance comme argument. Il y a quelques subtilités dans l'utilisation de merge () qui en font différent à utiliser par rapport aux autres méthodes du gestionnaire d'entités. Prenons l'exemple suivant, qui montre une méthode de bean session qui accepte un paramètre Employee détaché et le fusionne dans le contexte de persistance actuel:

```
public void updateEmployee (Employee emp) {
        em.merge (emp);
        emp.setLastAccessTime (nouvelle date ());
}
```

En supposant qu'une transaction commence et se termine par cet appel de méthode, tout changement effectué sur l'instance Employee alors qu'elle était détachée sera écrit dans la base de données. Ce qui ne sera pas écrit, cependant, est le changement de l'heure du dernier accès. L'argument to merge () n'est pas géré à la suite de la fusion. Une gestion différente entité (soit une nouvelle instance, soit une version gérée existante déjà dans la persistance context) est mis à jour pour correspondre à l'argument, puis cette instance est renvoyée par le merge () méthode. Par conséquent, pour capturer ce changement, nous devons utiliser la valeur de retour de merge () car il s'agit de l'entité gérée. L'exemple suivant montre le bon la mise en oeuvre:

```
public void updateEmployee (Employee emp) {
        Employé managedEmp = em.merge (emp);
        managedEmp.setLastAccessTime (nouvelle date ());
}
```


Épisode 259

Le renvoi d'une instance gérée autre que l'entité d'origine est un élément essentiel de le processus de fusion. Si une instance d'entité avec le même identifiant existe déjà dans le contexte de persistance, le fournisseur écrasera son état par l'état de l'entité qui est en cours de fusion, mais la version gérée qui existait déjà doit être renvoyée au client afin qu'il puisse être utilisé. Si le fournisseur n'a pas mis à jour l'instance Employee dans le contexte de persistance, toute référence à cette instance deviendra incompatible avec le nouvel état en cours de fusion.

Lorsque merge () est invoqué sur une nouvelle entité, il se comporte de la même manière que persist () opération. Il ajoute l'entité au contexte de persistance, mais au lieu d'ajouter l'original instance d'entité, il crée une nouvelle copie et gère cette instance à la place. La copie qui est créé par l'opération merge () est conservé comme si la méthode persist () était appelée dessus.

En présence de relations, l'opération merge () tentera de mettre à jour le entité gérée pour pointer vers les versions gérées des entités référencées par le détaché entité. Si l'entité a une relation avec un objet qui n'a pas d'identité persistante,

le résultat de l'opération de fusion n'est pas défini. Certains fournisseurs peuvent autoriser
copie gérée pour pointer vers l'objet non persistant, alors que d'autres peuvent lancer un
exception immédiatement. L'opération merge () peut être éventuellement mise en cascade dans ces
cas pour éviter qu'une exception ne se produise. Nous couvrons la cascade de la fusion ()
opération plus loin dans cette section. Si une entité en cours de fusion pointe vers une entité supprimée, un
Une exception IllegalArgumentException sera levée.

Les relations de chargement différé sont un cas particulier dans l'opération de fusion. Si un chargement paresseux
relation n'a pas été déclenchée sur une entité avant qu'elle ne se détache, cette relation
sera ignoré lors de la fusion de l'entité. Si la relation a été déclenchée pendant
gérée, puis définie sur null pendant le détachement de l'entité, la version gérée du
l'entité verra également la relation effacée lors de la fusion.

Pour illustrer le comportement de merge () avec les relations, considérez le diagramme d'objets
illustré à la figure 6-3 . L'objet emp détaché a des relations avec trois autres objets.
Les objets addr et dept sont des entités détachées d'une transaction précédente, alors que
l'entité phone1 a été créée récemment et a persisté à l'aide de l'opération persist () et
est désormais géré en conséquence. Dans le contexte de persistance, il y a actuellement un employé
instance avec une relation avec une autre adresse gérée. L'existant géré
L'instance Employee n'a pas de relation avec l'instance Phone nouvellement gérée.

Épisode 260

Figure 6-3. *État de l'entité avant la fusion*

Considérons l'effet de l'invocation de merge () sur l'objet emp. La première chose qui
se produit est que le fournisseur vérifie le contexte de persistance pour une entité préexistante
instance avec le même identifiant. Dans cet exemple, l'objet emp1 de la persistance
context correspond à l'identifiant de l'objet emp que nous essayons de fusionner. Donc,
l'état de base de l'objet emp écrase l'état de l'objet emp1 dans la persistance
context, et l'objet emp1 sera renvoyé par l'opération merge ().

Le fournisseur considère ensuite les entités Téléphone et Département vers lesquelles pointé emp.
L'objet phone1 est déjà géré, le fournisseur peut donc mettre à jour emp1 en toute sécurité pour pointer vers
cette instance. Dans le cas de l'objet dept, le fournisseur vérifie s'il y a
déjà une entité Department persistante avec le même identifiant. Dans ce cas, il en trouve un
dans la base de données et le charge dans le contexte de persistance. L'objet emp1 est ensuite mis à jour
pour pointer vers cette version de l'entité Département. L'objet dept détaché ne

redevenir géré.

Enfin, le fournisseur vérifie l'objet addr référencé depuis emp. Dans ce cas, il trouve un objet géré préexistant addr1 avec le même identifiant. Parce que l'objet emp1 pointe déjà vers l'objet addr1, aucune autre modification n'est effectuée. À ce stade, regardons l'état du modèle objet après la fusion. Figure 6-4 montre ces changements.

Épisode 261

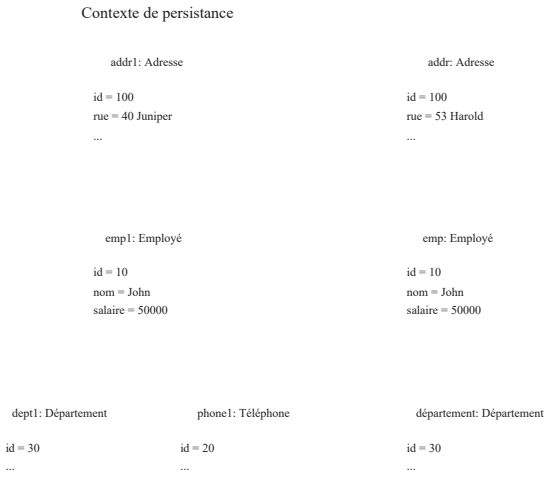

Figure 6-4. *État de l'entité après la fusion*

En chiffres 6-4 on voit que l'objet emp1 a été mis à jour pour refléter l'état changements de emp. L'objet dept1 est nouveau dans le contexte de persistance après avoir été chargé à partir de la base de données. L'objet emp1 pointe maintenant à la fois vers l'objet phone1 et le dept1 object afin de faire correspondre les relations de l'objet emp. L'objet addr1 n'a pas changé du tout. Le fait que l'objet addr1 n'ait pas changé peut être une surprise. Après tout, l'objet addr avait des modifications en attente et il était indiqué par l'objet emp qui a été fusionné.

Pour comprendre pourquoi, il faut revenir sur la question des opérations en cascade avec le gestionnaire d'entité. Par défaut, aucune opération n'est mise en cascade lorsqu'un gestionnaire d'entités l'opération est appliquée à une instance d'entité. L'opération merge () n'est pas différente dans ce cas qui concerne. Pour que la fusion se déroule en cascade dans les relations d'un employé, le Le paramètre de cascade MERGE doit être défini sur les mappages de relations. Sinon, nous le ferions doivent invoquer merge () sur chaque objet associé.

En revenant à notre exemple, le problème avec l'entité Address mise à jour était que l'entité Employee ne lui a pas transmis l'opération de fusion () en cascade. Cela a eu le malheureux effet secondaire de la suppression effective des modifications que nous avions apportées à l'entité Address dans faveur de la version déjà dans le contexte de persistance. Pour obtenir le comportement que nous

Épisode 262

prévu, nous devons soit invoquer merge () explicitement sur l'objet addr, soit changer le mappages de relations de l'entité Employé pour inclure l'option en cascade MERGE. Référencement 6-23 montre la classe d'employé modifiée.

Liste 6-23. Entité d'employé avec paramètre de fusion en cascade

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;
    @ManyToOne (cascade = CascadeType.MERGE)
    adresse d'adresse privée;
    @ManyToOne
    département du département privé;
    @OneToMany (mappedBy = "employé", cascade = CascadeType.MERGE)
    les téléphones <Phone> de la collection privée;
    // ...
}
```

Avec l'entité Employé modifiée de cette manière, l'opération de fusion sera en cascade aux entités Address et Phone vers lesquelles pointent les instances Employee. Cela équivaut à appeler merge () sur chaque instance individuellement. Notez que nous n'a pas répercuté l'opération de fusion sur l'entité du Ministère. Nous cascade généralement opérations uniquement vers le bas du parent à l'enfant, pas vers le haut de l'enfant au parent. Faire ce n'est donc pas dangereux, mais cela demande plus d'efforts de la part du fournisseur de persistance pour rechercher les changements. Si l'entité Département change également, il est préférable de mettre en cascade la fusion du Département à ses instances Employé associées, puis ne fusionner qu'un seul Instance de service au lieu de plusieurs instances d'employé.

La fusion d'entités détachées avec des relations peut être une opération délicate. Idéalement, nous souhaitez fusionner la racine d'un graphe d'objets et fusionner toutes les entités associées le processus. Cela peut fonctionner, mais uniquement si le paramètre de cascade MERGE a été appliqué à tous relations dans le graphique. Si ce n'est pas le cas, vous devez fusionner chaque instance qui est la cible de une relation non en cascade une à la fois.

Avant de quitter le sujet de la fusion, nous devons mentionner que le verrouillage et le contrôle de version joue un rôle essentiel pour garantir l'intégrité des données dans ces situations. Nous explorons ce sujet dans Chapitre 12.

Épisode 263

Travailler avec des entités détachées

Commençons par un scénario très courant avec les applications Web modernes. Un servlet appelle un bean session pour exécuter une requête et reçoit une collection d'entités dans revenir. Le servlet place ensuite ces entités dans la mappe de requête et transmet le demande à un JSP pour présentation. Ce modèle est appelé contrôleur de page [1], une variation de le contrôleur avant[2] modèle dans lequel il y a un seul contrôleur pour chaque vue au lieu de un contrôleur central pour toutes les vues. Dans le contexte du familier Model-View-Controller (MVC), le bean session fournit le modèle, la page JSP est la vue, et le servlet est le contrôleur.

Considérons d'abord le bean CDI géré qui produira les résultats qui seront rendu par la page JSP. Le Listing 6-24 montre l'implémentation du bean. Dans cet exemple, nous examinons uniquement la méthode findAll (), qui retourne toutes les instances Employee stocké dans la base de données.

Annonce 6-24. Le bean EmployeeService

```
@Dépendant
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
```

```
    privé EntityManager em;

    Liste publique findAll () {
        return em.createQuery ("SELECT e FROM Employee e")
                .getResultList ();
    }

    // ...
}
```

Référencement 6-25 montre le code source d'un simple servlet qui invoque findAll () méthode du bean EmployeeService pour récupérer toutes les entités Employee de la base de données. Il place ensuite les résultats dans la mappe de requêtes et les délègue à la JSP listEmployees.jsp page pour rendre le résultat.

[1] Martin Fowler, *Patterns of Enterprise Application Architecture* . Boston: Addison-Wesley, 2002.

[2] Deepak Alur, John Crupi et Dan Malks, *Core J2EE Patterns: Best Practices and Design Strategies, Deuxième édition* . Upper Saddle River, NJ: Prentice Hall PTR, 2003.

Épisode 264

Liste 6-25. Le servlet Afficher les employés

```
Public class EmployeeServlet étend HttpServlet {
    @Inject EmployeeService bean;

    protected void doGet (requête HttpServletRequest, HttpServletResponse
    réponse)
            lance ServletException, IOException {
        Liste emps = bean.findAll ();
        request.setAttribute ("employés", emps);
        getServletContext (). getRequestDispatcher ("/ listEmployees.jsp")
                .forward (demande, réponse);
    }
}
```

Enfin, liste 6-26 montre la dernière partie de notre architecture MVC, la page JSP rendre les résultats. Il utilise la bibliothèque de balises standard JavaServer Pages (JSTL) pour parcourir la collection d'instances Employé et afficher le nom de chaque employé ainsi que le nom du service auquel cet employé est affecté. La variable employés accessible par la balise <c: forEach /> est la liste des instances Employee qui a été placée dans la carte de demande par le servlet.

Liste 6-26. Page JSP pour afficher les informations sur les employés

```
<! DOCTYPE HTML PUBLIC "- // W3C // DTD HTML 4.01 Transitional // EN">
<% @ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c"%>
<html>
  <head>
    <title> Tous les employés </title>
  </head>
  <body>
    <table>
      <thead>
        <tr>
          <th> Nom </th>
          <th> Département </th>
        </tr>
```

```
        </thead>
```


Épisode 265


```
        <tbody>
          <c: forEach items = "$ {Employees}" var = "emp">
            <tr>
              <td> <c: out value = "$ {emp.name}" /> </td>
              <td> <c: out value = "$ {emp.department.name}" /> </td>
            </tr>
          </ c: forEach>
        </tbody>
      </table>
    </body>
</html>
```

La méthode findAll () du bean EmployeeService n'a pas de transaction. Parce que le servlet invoquant la méthode n'a pas lancé de transaction, findAll () est invoqué dans pas de contexte de transaction; ainsi, les résultats de la requête se détachent avant d'être retourné au servlet.

Cela pose un problème. Dans cet exemple, la relation de service de l'employé La classe a été configurée pour utiliser la récupération paresseuse. Comme vous l'avez appris précédemment dans la section au détachement, la seule chose portable à faire est de les laisser tranquilles. Dans cet exemple, cependant, nous ne voulons pas les laisser seuls. Afin d'afficher le nom du département pour l'employé, l'expression JSP accède à l'entité Department à partir de l'employé entité. Comme il s'agit d'une relation à chargement différé, les résultats sont imprévisibles. Ça pourrait travailler, mais encore une fois, il pourrait ne pas.

Ce scénario constitue la base de notre défi. Dans les sections suivantes, nous examinons à un certain nombre de stratégies pour préparer les entités nécessaires à la page JSP pour détachement ou éviter complètement le détachement.

Planification du détachement

Sachant que les résultats de la méthode findAll () seront utilisés pour afficher l'employé informations et que le nom du département sera requis dans le cadre de ce processus, nous devons nous assurer que la relation de service de l'entité Employé a été résolu avant que les entités ne se détachent. Il existe plusieurs stratégies qui peuvent être utilisé pour résoudre les associations chargées paresseusement en préparation du détachement. Nous discutons de deux d'entre eux ici, en se concentrant sur la façon de structurer le code d'application pour planifier le détachement. UNE la troisième stratégie, pour les requêtes JP QL appelées fetch join, est discutée au chapitre 8 , et une quatrième, l'utilisation de graphes d'entités est expliquée au chapitre 11 .

Épisode 266

Déclenchement du chargement différé

La première stratégie à considérer pour résoudre les associations de chargement différé consiste simplement à déclencher le comportement de chargement paresseux en accédant au champ ou à la relation. Cela semble un peu étrange code car les valeurs de retour des méthodes getter sont ignorées, mais néanmoins a l'effet désiré. Référencement6-27 montre une autre implémentation de findAll () méthode du bean de session EmployeeService. Dans ce cas, nous itérons sur l'employé

entités, déclenchant la relation départementale avant de renvoyer la liste d'origine à partir du méthode. Étant donné que findAll () est exécuté à l'intérieur d'une transaction, le getDepartment () l'appel se termine avec succès et l'instance d'entité Department est garantie disponible lorsque l'instance Employee est détachée.

Liste 6-27. Déclencher une relation de chargement différé

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    privé EntityManager em;

    Liste publique findAll () {
        List <Employee> emps = (List <Employee>)
            em.createQuery ("SELECT e FROM Employee e")
              .getResultList ();
        for (Employé emp: emps) {
            Département dept = emp.getDepartment ();
            if (dept! = null) {
                dept.getName ();
            }
        }
        return emps;
    }

    // ...
}
```

Une chose qui pourrait sembler étrange à partir de la liste 6-27 est que nous n'avons pas seulement invoqué getDepartment () sur l'instance Employee mais nous avons également appelé getName () sur l'instance Department. Si vous vous souvenez du chapitre 4 , l'entité est revenue d'un

Épisode 267

la relation de chargement paresseux peut en fait être un proxy qui attend qu'une méthode soit appelée sur le proxy avant la faute de l'entité. Nous devons invoquer une méthode sur l'entité pour garantir qu'il est effectivement extrait de la base de données. Si c'était une collection-relation valorisée, la méthode size () de la collection serait couramment utilisée pour forcer le chargement impatient.

Si des mappages de base à chargement différé ont été utilisés sur l'employé ou le service entités, ces attributs ne seraient pas garantis d'être présents après le détachement car bien. C'est une autre raison pour laquelle la configuration des mappages de base pour utiliser le chargement différé n'est pas conseillé. Les développeurs s'attendent souvent à ce qu'une relation ne soit pas très chargée mais peut être pris au dépourvu si un champ d'état de base tel que l'attribut name de l'employé l'instance est manquante.

Configuration du chargement hâtif

Lorsqu'une association est continuellement déclenchée pour des scénarios de détachement, à certains point il vaut la peine de revoir si l'association doit être chargée paresseusement dans le premier endroit. Passer soigneusement certaines relations à un chargement hâtif peut éviter beaucoup de cas dans le code qui tentent de déclencher le chargement différé.

Dans cet exemple, l'employé a une relation plusieurs-à-un avec le service. le le type de récupération par défaut pour une relation plusieurs-à-un est un chargement hâtif, mais la classe était modélisé en utilisant explicitement le chargement différé. En supprimant le type de récupération LAZY du relation de service ou en spécifiant explicitement le type de récupération EAGER, nous nous assurons que l'instance Department est toujours disponible pour l'instance Employee.

Les relations à valeur de collection se chargent par défaut, de sorte que le type d'extraction EAGER doit être explicitement appliqué à ces mappages si un chargement rapide est souhaité. Soyez judicieux dans configurer les relations à valeur de collection pour qu'elles soient chargées avec impatience, car cela peut entraîner un accès excessif à la base de données dans les cas où le détachement n'est pas un exigence.

Éviter le détachement

La seule solution complète à tout scénario de détachement est de ne pas se détacher du tout. Si votre code déclenche méthodiquement chaque relation paresseuse ou a marqué chaque association sur une entité à charger avec empressement en prévision du détachement, c'est probablement un signe qu'une approche alternative est nécessaire.

Épisode 268

Éviter le détachement se résume à deux approches. Soit vous ne travaillez pas avec entités dans votre page JSP, ou vous devez garder un contexte de persistance ouvert pendant la durée de le processus de rendu JSP afin que les relations de chargement différé puissent être résolues.

Ne pas utiliser d'entités signifie copier les données d'entité dans une structure de données différente n'ont pas le même comportement de chargement paresseux. Une approche consisterait à utiliser le transfert Motif de l' objet [3], mais cela semble très redondant étant donné la nature POJO des entités. UNE meilleure approche, dont nous discutons dans les chapitres 7 et8, consiste à utiliser des requêtes de projection pour récupérer uniquement l'état de l'entité qui sera affiché sur la page JSP au lieu de récupérer instances d'entité complètes.

Garder un contexte de persistance ouvert nécessite une planification supplémentaire mais permet Page JSP pour travailler avec les données d'entité à l'aide des propriétés JavaBean de la classe d'entité. Dans termes pratiques, garder un contexte de persistance ouvert signifie qu'il y a soit un transaction pour les entités récupérées à partir de contextes de persistance à portée de transaction ou qu'un Un contexte de persistance étendu ou géré par l'application est en cours d'utilisation. Ce n'est évidemment pas un option lorsque les entités doivent être sérialisées vers un niveau distinct ou un client distant, mais cela convient scénario d'application Web décrit précédemment. Nous couvrons chacune de ces stratégies ici.

Vue de transaction

Le contexte de persistance créé par un gestionnaire d'entités à portée transaction reste ouvert seulement tant que la transaction dans laquelle il a été créé n'est pas terminée. Par conséquent, dans l'ordre pour utiliser un gestionnaire d'entités à portée transactionnelle pour exécuter une requête et être en mesure de rendre les résultats de la requête lors de la résolution des relations de chargement différé, les deux opérations doivent être partie de la même transaction. Lorsqu'une transaction est lancée dans le niveau Web et inclut à la fois l'invocation du bean session et le rendu de la page JSP avant qu'il ne soit validé, nous appelons cela modèle une vue de transaction.

L'avantage de cette approche est que toutes les relations de chargement différé rencontrées lors du rendu de la vue sera résolu car les entités sont toujours gérées par un contexte de persistance. Pour implémenter ce modèle dans notre exemple de scénario, nous commençons une transaction gérée par bean avant que la méthode findAll () ne soit appelée et validée la transaction après que la page JSP a rendu les résultats. Référencement6-28 démontre cette approche. Notez que pour économiser de l'espace, nous avons omis la gestion des exceptions levées par les opérations UserTransaction. La méthode commit () seule jette pas moins de six exceptions vérifiées.

[3] Ibid.

Épisode 269

Liste 6-28. Combinaison d'une méthode Session Bean et JSP en une seule transaction

```
Public class EmployeeServlet étend HttpServlet {
    @Resource UserTransaction tx;
    @EJB EmployeeService bean;

    protected void doGet (requête HttpServletRequest, HttpServletResponse
    réponse)
            lance ServletException, IOException {
        // ...
        essayez {
            tx.begin ();
            Liste emps = bean.findAll ();
            request.setAttribute ("employés", emps);
            getServletContext (). getRequestDispatcher ("/ listEmployees.jsp")
                                        .forward (demande, réponse);
        } enfin {
            tx.commit
        }
        // ...
    }
}
```

Avec cette solution en place, les relations de chargement paresseux de l'entité Employé n'ont pas à être résolus avec impatience avant que la page JSP affiche les résultats. Le seul l'inconvénient de cette approche est que le servlet doit désormais gérer les transactions et récupérer des échecs de transaction. Beaucoup de logique doit également être dupliquée entre tous les servlet contrôleurs qui ont besoin de ce comportement.

Une façon de contourner cette duplication consiste à introduire une superclasse commune pour les servlets qui utilisent le modèle Transaction View qui encapsule la transaction comportement. Si, cependant, vous utilisez le modèle de contrôleur avant et les actions du contrôleur sont implémentées à l'aide du modèle Command [4] , cela peut devenir plus difficile à gérer, en particulier si le flux de pages est complexe et que plusieurs contrôleurs collaborent pour construire une vue composite. Ensuite, non seulement chaque contrôleur doit démarrer des transactions

[4] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns: Elements of Logiciel orienté objet réutilisable* . Boston: Addison-Wesley, 1995.

Épisode 270

mais il doit également être conscient de toutes les transactions qui ont été lancées plus tôt dans le rendu séquence.

Une autre solution possible, bien que non portable, consiste à déplacer la logique de transaction dans un filtre de servlet. Cela nous permet d'intercepter la requête HTTP avant le premier contrôleur servlet est accédé et encapsule la demande entière dans une transaction. Une telle utilisation à gros grains des transactions est quelque chose qui doit être géré avec soin, cependant. Si appliqué à toutes les requêtes HTTP de la même manière, cela peut également causer des problèmes pour les requêtes impliquant des mises à jour à la base de données. En supposant que ces opérations sont implémentées en tant que session beans, le L'attribut de transaction REQUIRES_NEW peut être requis pour isoler les mises à jour d'entité et gérer l'échec de la transaction sans affecter la transaction globale prioritaire.

Entity Manager par demande

Pour les applications qui n'encapsulent pas leurs opérations de requête derrière le bean session façades, une alternative au modèle Transaction View est de créer une nouvelle application gestionnaire d'entités géré pour exécuter les requêtes de rapport, en le fermant uniquement après la page JSP a été rendu. Parce que les entités renvoyées par la requête sur l'application-le gestionnaire d'entité géré restera géré jusqu'à ce que le gestionnaire d'entité soit fermé, il offre les mêmes avantages que le modèle Transaction View sans nécessiter un transaction.

Référencement 6-29 revisite à nouveau notre servlet EmployeeServlet, cette fois en créant un gestionnaire d'entités géré par l'application pour exécuter la requête. Les résultats sont placés dans la carte comme avant, et le gestionnaire d'entités est fermé une fois la page JSP terminée le rendu.

Liste 6-29. Utilisation d'un Entity Manager géré par une application pour la création de rapports

```
Public class EmployeeServlet étend HttpServlet {
    @PersistenceUnit (unitName = "EmployeeService")
    EntityManagerFactory emf;

    protected void doGet (requête HttpServletRequest,
                                    Réponse HttpServletResponse)
                lance ServletException, IOException {
        EntityManager em = emf.createEntityManager ();
```

Épisode 271


```
            essayez {
                List emps = em.createQuery ("SELECT e FROM Employee e")
                                    .getResultList ();
                request.setAttribute ("employés", emps);
                getServletContext (). getRequestDispatcher ("/ listEmployees.jsp")
                                        .forward (demande, réponse);
            } enfin {
                em.close ();
            }
        }
}
```

Malheureusement, nous avons maintenant une logique de requête intégrée dans notre implémentation de servlet. La requête n'est également plus réutilisable telle qu'elle était lorsqu'elle faisait partie d'une session sans état haricot. Il y a quelques autres options que nous pouvons explorer comme solution à ce problème. Au lieu d'exécuter la requête directement, nous pourrions créer une classe de service POJO qui utilise le gestionnaire d'entités géré par l'application créé par le servlet pour exécuter des requêtes. C'est similaire au premier exemple que nous avons créé au chapitre 2 . Nous avons l'avantage d'encapsuler le comportement des requêtes dans les méthodes métier tout en étant découplé d'un style de gestionnaire d'entités.

Nous pouvons également placer nos méthodes de requête sur un bean session avec état qui utilise un gestionnaire d'entités étendu. Lorsqu'un bean session avec état utilise une entité étendue manager, son contexte de persistance dure toute la durée de vie du bean session, qui se termine uniquement lorsque l'utilisateur appelle une méthode remove sur le bean. Si une requête est exécutée sur le contexte de persistance étendue d'un bean session avec état, les résultats de cette requête peuvent continuez à résoudre les relations de chargement différé tant que le bean est toujours disponible.

Explorons cette option et voyons à quoi elle ressemblerait au lieu de l'application-

gestionnaire d'entité géré que nous avons montré dans le Listing 6-29 . Le listing 6-30 introduit un stateful bean session équivalent au bean session sans état EmployeeService que nous avons utilisé jusqu'à présent. En plus d'utiliser le gestionnaire d'entités étendu, nous avons également défini le le type de transaction par défaut est NOT_SUPPORTED. Il n'y a pas besoin de transactions car les résultats de la requête ne seront jamais modifiés, uniquement affichés.

Épisode 272

Liste 6-30. Bean session avec état avec méthodes de requête

```
@Stateful
@TransactionAttribute (TransactionAttributeType.NOT_SUPPORTED)
Public class EmployeeQuery {
    @PersistenceContext (type = PersistenceContextType.EXTENDED,
                         unitName = "EmployeeService")
    EntityManager em;

    Liste publique findAll () {
        return em.createQuery ("SELECT e FROM Employee e")
                .getResultList ();
    }

    // ...

    @Retirer
    public void done () {
    }
}
```

L'utilisation de ce bean est très similaire à l'utilisation du gestionnaire d'entités géré par l'application. nous créer une instance du bean, exécuter la requête, puis supprimer le bean lorsque le Le rendu de la page JSP est terminé. Référencement6-31 montre cette approche.

Annonce 6-31. Utilisation d'un Extended Entity Manager pour la création de rapports

```
@EJB (nom = "queryBean", beanInterface = EmployeeQuery.class)
Public class EmployeeServlet étend HttpServlet

    protected void doGet (requête HttpServletRequest, HttpServletResponse
réponse)
            lance ServletException, IOException {
        EmployeeQuery bean = createQueryBean ();
        essayez {
            Liste emps = bean.findAll ();
            request.setAttribute ("employés", emps);
            getServletContext (). getRequestDispatcher ("/ listEmployees.jsp")
                                .forward (demande, réponse);
```


Épisode 273

```
            } enfin {
                bean.finished ();
            }
        }

        private EmployeeQuery createQueryBean () lance une exception ServletException {
            // recherche queryBean
            // ...
        }
}
```

À première vue, cela peut sembler une solution surdimensionnée. Nous en tirons profit de découplage des requêtes du servlet, mais nous avons introduit un nouveau bean session juste pour atteindre cet objectif. De plus, nous utilisons des beans session avec état avec des durées de vie. Cela ne va-t-il pas à l'encontre de la pratique acceptée d'utilisation d'une session avec état haricot?

Dans une certaine mesure, cela est vrai, mais le contexte de persistance étendue nous invite à expérimenter de nouvelles approches. En pratique, les beans session avec état n'ajoutent pas de quantité importante de frais généraux pour une opération, même lorsqu'elle est utilisée pendant de courtes durées. Comme vous verrez plus tard dans la section "Modifier la session", déplacer le bean session avec état vers le La session HTTP au lieu de la limiter à une seule requête ouvre également de nouvelles possibilités pour conception d'applications Web.

Fusionner les stratégies

La création ou la mise à jour des informations fait partie intégrante de la plupart des applications d'entreprise. Utilisateurs interagissent généralement avec une application via le Web, en utilisant des formulaires pour créer ou modifier des données comme demandé. La stratégie la plus courante pour gérer ces changements dans une application Java EE qui utilise JPA consiste à placer les résultats des modifications dans des instances d'entités détachées et fusionner les modifications en attente dans un contexte de persistance afin qu'elles puissent être écrites dans le base de données.

Revoyons à nouveau notre scénario d'application Web simple. Cette fois, au lieu de simplement affichage des informations sur les employés, l'utilisateur peut sélectionner un employé et mettre à jour des informations sur cet employé. Les entités sont interrogées pour une présentation sous une forme en une demande puis mis à jour dans une deuxième demande lorsque l'utilisateur soumet le formulaire avec modifications entrées.

Épisode 274

En utilisant un modèle de façade de session, cette opération est simple. L'entité modifiée est mis à jour et transféré à un bean session sans état pour être fusionné. La seule complexité est de s'assurer que les relations fusionnent correctement en identifiant les cas où Le réglage de la cascade MERGE est requis.

Similaire à la question de savoir si nous pouvons éviter de détacher des entités pour compenser pour les problèmes de chargement paresseux, la nature à long terme des applications gérées et étendues les contextes de persistance suggèrent qu'il pourrait également y avoir un moyen d'appliquer une technique similaire à cette situation. Au lieu d'interroger des entités dans une seule requête HTTP et de lancer le les instances d'entité après le rendu de la vue, nous souhaitons conserver ces entités autour dans un état géré afin qu'ils puissent être mis à jour dans une requête HTTP ultérieure et persistait simplement en commençant et en validant une nouvelle transaction.

Dans les sections suivantes, nous revisitons l'approche traditionnelle de la façade de session pour fusionner puis examiner les nouvelles techniques possibles avec le gestionnaire d'entités étendu qui gardera les entités gérées pendant toute la durée de la session d'édition d'un utilisateur.

Façade de session

Pour utiliser un modèle de façade de session pour capturer les modifications apportées aux entités, nous fournissons une entreprise méthode qui fusionnera les modifications apportées à une instance d'entité détachée. Dans notre exemple scénario, cela signifie accepter une instance Employee et la fusionner dans une transaction-contexte de persistance de portée. Le listing 6-32 montre une implémentation de cette technique dans notre bean de session EmployeeService.

Listing 6-32. Méthode commerciale pour mettre à jour les informations sur les employés

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    privé EntityManager em;

    public void updateEmployee (Employee emp) {
        if (em.find (Employee.class, emp.getId ()) == null) {
            throw new IllegalArgumentException ("Identifiant d'employé inconnu:" +
                                                emp.getId ());
        }
```

Épisode 275


```
        em.merge (emp);
    }

    // ...
}
```

La méthode updateEmployee () du Listing 6-32 est simple. Compte tenu du instance Employee détachée, il tente d'abord de vérifier si un identifiant correspondant existe déjà. Si aucun employé correspondant n'est trouvé, une exception est levée car nous ne souhaite pas autoriser la création de nouveaux enregistrements d'employé. Ensuite, nous utilisons la fusion () opération pour copier les modifications dans le contexte de persistance, qui sont ensuite enregistrées lorsque la transaction est validée.

L'utilisation de la façade à partir d'un servlet est une approche en deux étapes. Pendant le HTTP initial demande de commencer une session d'édition, l'instance Employee est interrogée (généralement à l'aide d'un méthode distincte sur la même façade) et utilisé pour créer un formulaire Web sur lequel l'utilisateur peut apporter les modifications souhaitées. L'instance détachée est ensuite stockée dans la session HTTP afin il peut être mis à jour lorsque l'utilisateur soumet le formulaire à partir du navigateur. Nous devons garder le une instance détachée afin de préserver les relations ou tout autre état qui restent inchangés par la modification. Créer une nouvelle instance Employee et fournir uniquement les valeurs partielles peuvent avoir de nombreux effets secondaires négatifs lorsque l'instance est fusionnée.

Référencement 6-33 montre un servlet EmployeeUpdateServlet qui collecte l'ID, le nom et informations de salaire à partir des paramètres de demande et invoque la méthode du bean session pour effectuer la mise à jour. L'instance Employee précédemment détachée est extraite du La session HTTP, puis les modifications indiquées par les paramètres de la requête y sont définies. Nous avons omis la validation des paramètres de demande pour économiser de l'espace, mais idéalement, cela doit se produire avant que la méthode métier sur le bean session ne soit appelée.

Annonce 6-33. Utilisation d'un bean session pour effectuer des mises à jour d'entité

```
Public class EmployeeUpdateServlet étend HttpServlet {
    @EJB EmployeeService bean;
```

```
protected void doPost (requête HttpServletRequest,
                       Réponse HttpServletResponse)
            lance ServletException, IOException {
        int id = Integer.parseInt (request.getParameter ("id"));
        String name = request.getParameter ("name");
        salaire long = Long.parseLong (request.getParameter ("salaire"));
```


Épisode 276


```
            Session HttpSession = request.getSession ();
            Employé emp = (Employé) session.getAttribute ("employee.edit");
            emp.setId (id);
            emp.setName (nom);
            emp.setSalary (salaire);
            bean.updateEmployee (emp);
            // ...
        }
}
```

Si la quantité d'informations mises à jour est très petite, nous pouvons éviter le détachement
object et merge () entièrement en localisant la version gérée et manuellement
copier les modifications dedans. Prenons l'exemple suivant:

```
public void updateEmployee (int id, nom de chaîne, salaire long) {
    Employé emp = em.find (Employee.class, id);
    if (emp == null) {
        throw new IllegalArgumentException ("Identifiant d'employé inconnu:" + id);
    }
    emp.setEmpName (nom);
    emp.setSalary (salaire);
}
```

La beauté de cette approche est sa simplicité, mais c'est aussi sa principale limite.
Les applications Web typiques offrent aujourd'hui la possibilité de mettre à jour de grandes quantités d'informations
en une seule opération. Pour accommoder ces situations avec ce modèle, il y aurait
doivent être des méthodes commerciales prenant un grand nombre de paramètres ou de nombreux
méthodes commerciales qui devraient être appelées en séquence pour mettre à jour complètement
toutes les informations nécessaires. Et, bien sûr, une fois que vous avez plus d'une méthode
impliqué, il peut être nécessaire de maintenir une transaction sur toutes les méthodes de mise à jour afin
que les modifications sont validées comme une seule unité.

Par conséquent, malgré la disponibilité de cette approche, le niveau Web
collecte les modifications en entités détachées ou transfère des objets et passe l'état modifié
retour aux beans session à fusionner et à écrire dans la base de données.

Épisode 277

Modifier la session

Avec l'introduction du gestionnaire d'entités étendu, nous pouvons adopter une approche différente
pour créer des applications Web qui mettent à jour les entités. Comme nous l'avons vu dans ce chapitre,
les entités associées à un gestionnaire d'entités étendu restent gérées tant que le
Le bean session avec état contenant le gestionnaire d'entités étendu n'est pas supprimé. En plaçant
un bean session avec état dans un emplacement central tel que la session HTTP, nous pouvons opérer
sur les entités gérées par le gestionnaire d'entités étendu sans avoir à fusionner dans l'ordre
pour persister les changements. Nous appelons cela le modèle de session d'édition pour refléter le fait que le
L'objectif principal de ce modèle est d'encapsuler des cas d'utilisation d'édition à l'aide d'une session avec état
des haricots.

Référencement 6-34 introduit un bean session avec état qui représente l'édition d'un employé
session. Contrairement au bean session EmployeeService qui contient un certain nombre de
méthodes métier, ce style de bean session avec état est destiné à une seule application
cas d'utilisation. En plus d'utiliser le gestionnaire d'entités étendu, nous avons également défini la valeur par défaut
le type de transaction doit être NOT_SUPPORTED à l'exception de la méthode save (). Il y a
pas besoin de transactions pour les méthodes qui accèdent simplement à l'instance Employee car
ces méthodes ne fonctionnent qu'en mémoire. Ce n'est que lorsque nous voulons persister les changements
la base de données dont nous avons besoin d'une transaction, et cela ne se produit que dans la méthode save ().

Annonce 6-34. Bean session avec état pour gérer une session d'édition d'employé

```
@Stateful
@TransactionAttribute (TransactionAttributeType.NOT_SUPPORTED)
Public class EmployeeEdit {
    @PersistenceContext (type = PersistenceContextType.EXTENDED,
                                    unitName = "EmployeeService")
    EntityManager em;
    Employé emp;

    public void begin (int id) {
        emp = em.find (Employee.class, id);
        if (emp == null) {
            throw new IllegalArgumentException ("Identifiant d'employé inconnu:
            "+ id);
        }
    }
```

Épisode 278


```
    employé public getEmployee () {
        return emp;
    }

    @Retirer
    @TransactionAttribute (TransactionAttributeType.REQUIRES_NEW)
    public void save () {}

    @Retirer
    public void cancel () {}
}
```

Commençons par mettre les opérations du bean EmployeeEdit en contexte. Quand le
La requête HTTP arrive et démarre la session d'édition, nous créons un nouvel EmployeeEdit
bean session avec état et appelez begin () en utilisant l'ID de l'instance Employee qui
sera édité. Le bean session charge ensuite l'instance Employee et la met en cache sur le
haricot. Le bean est alors lié à la session HTTP afin de pouvoir y accéder à nouveau dans un
demande ultérieure une fois que l'utilisateur a modifié les informations de l'employé. Référencement6-35
montre le servlet EmployeeEditServlet qui gère la requête HTTP pour commencer un nouveau

session d'édition.

Annonce 6-35. Début d'une session de modification des employés

```
@EJB (nom = "EmployeeEdit", beanInterface = EmployeeEdit.class)
Public class EmployeeEditServlet étend HttpServlet {

    protected void doPost (requête HttpServletRequest, HttpServletResponse
    réponse)
                lance ServletException, IOException {
        int id = Integer.parseInt (request.getParameter ("id"));
        EmployeeEdit bean = getBean ();
        bean.begin (id);
        Session HttpSession = request.getSession ();
        session.setAttribute ("employé.edit", bean);
        request.setAttribute ("employé", bean.getEmployee ());
        getServletContext (). getRequestDispatcher ("/ editEmployee.jsp")
                                    .forward (demande, réponse);
    }
```


Épisode 279


```
    public EmployeeEdit getBean () lance une exception ServletException {
        // recherche bean EmployeeEdit
        // ...
    }
}
```

Regardons maintenant l'autre moitié de la session d'édition, dans laquelle nous souhaitons commettre le changements. Lorsque l'utilisateur soumet le formulaire contenant les modifications nécessaires de l'employé, EmployeeUpdateServlet est appelé. Il commence par récupérer le bean EmployeeEdit depuis la session HTTP. Les paramètres de demande avec les valeurs modifiées sont ensuite copiés dans l'instance Employee obtenue en appelant getEmployee () sur EmployeeEdit haricot. Si tout est en ordre, la méthode save () est appelée pour écrire les modifications dans le base de données. Référencement6-36 montre l'implémentation EmployeeUpdateServlet. Notez que nous besoin de supprimer le bean de la session HTTP une fois la session d'édition terminée.

Annonce 6-36. Terminer une session d'édition des employés

```
Public class EmployeeUpdateServlet étend HttpServlet {

    protected void doPost (requête HttpServletRequest, HttpServletResponse
    réponse)
                lance ServletException, IOException {
        String name = request.getParameter ("name");
        salaire long = Long.parseLong (request.getParameter ("salaire"));
        Session HttpSession = request.getSession ();
        EmployeeEdit bean = (EmployeeEdit) session.getAttribute ("employé.
        Éditer");
        session.removeAttribute ("employé.edit");
        Employé emp = bean.getEmployee ();
        emp.setName (nom);
        emp.setSalary (salaire);
        bean.save ();
        // ...
    }
}
```

Épisode 280

Le modèle d'utilisation des beans session avec état et des gestionnaires d'entités étendus sur le Web le niveau est le suivant:

1. Pour chaque cas d'utilisation d'application qui modifie les données d'entité, nous créons un bean session avec état avec un contexte de persistance étendu. Cette bean conservera toutes les instances d'entité nécessaires pour rendre le les changements souhaités.
2. La requête HTTP qui lance le cas d'utilisation d'édition crée un instance du bean session avec état et la lie au HTTP session. Les entités sont récupérées à ce stade et utilisées pour remplissez le formulaire Web pour le modifier.
3. La requête HTTP qui termine le cas d'utilisation d'édition obtient l'instance de bean session avec état précédemment liée et écrit les données modifiées du formulaire Web dans les entités stockées sur le haricot. Une méthode est alors invoquée sur le bean pour valider le modifications de la base de données.

Dans notre scénario d'édition simple, cela peut sembler un peu excessif, mais cela peut échelle pour accueillir des sessions d'édition de toute complexité. Département, projet et d'autres informations peuvent toutes être modifiées en une ou même plusieurs sessions avec les résultats accumulé sur le bean session avec état jusqu'à ce que l'application soit prête à conserver résultats.

Un autre avantage est que les frameworks d'applications Web tels que JSF peuvent accéder directement le bean lié dans la session HTTP à partir des pages JSP. L'entité est accessible à la fois pour afficher le formulaire à modifier et comme cible du formulaire lorsque l'utilisateur soumet Les resultats. Dans ce scénario, le développeur doit uniquement s'assurer que la sauvegarde et Les méthodes d'annulation sont appelées au point correct dans le flux de page de l'application.

Il y a quelques autres points que nous devons mentionner à propos de cette approche. Une fois lié à la session HTTP, le bean session y restera jusqu'à ce qu'il soit explicitement supprimé ou jusqu'à l'expiration de la session HTTP. Il est donc important de s'assurer que le bean est supprimé une fois la session d'édition terminée, que les modifications sera sauvegardé ou abandonné. L'interface de rappel HttpSessionBindingListener peut être utilisé par les applications pour suivre le moment où la session HTTP est détruite et nettoyer les beans session correspondants de manière appropriée.

Épisode 281

La session HTTP n'est pas thread-safe, et le bean session avec état non plus les références. Dans certaines circonstances, il peut être possible pour plusieurs requêtes HTTP du même utilisateur pour accéder simultanément à la session HTTP. C'est surtout un problème lorsque les demandes prennent du temps à traiter et qu'un utilisateur impatient actualise la page ou abandonne sa session d'édition pour une autre partie de l'application Web. Dans ces

circonstances, l'application web devra soit faire face à d'éventuelles exceptions
se produisant si le bean session avec état est accédé par plus d'un thread ou proxy
bean session avec état avec un wrapper synchronisé.

Conversation

La dernière stratégie est une sorte d'extension / amélioration de l'approche de session d'édition, mais
qui engage le contexte de persistance non synchronisé décrit plus haut dans le chapitre.
CDI peut également être utilisé pour gérer l'instance du bean dans notre étendue de session.

Comme nous l'avons dit au chapitre 1, JPA version 2.2 permet aux convertisseurs d'attributs de prendre en charge
Injection de CDI dans les classes AttributeConverter. Avec cette nouvelle fonctionnalité, la persistance
le fournisseur peut prendre en charge l'injection de CDI dans les convertisseurs d'attributs dans d'autres environnements
dont le BeanManager est disponible.

Le fournisseur de persistance est responsable de:

- Utilisation du CDI SPI pour créer des instances de la classe de convertisseur d'attributs
- Effectuer une injection sur de telles instances pour appeler leur
  Méthodes PostConstruct et PreDestroy.
- Élimination des instances de convertisseur d'attributs.

Notez que le fournisseur de persistance est uniquement requis pour prendre en charge l'injection CDI dans
convertisseurs d'attributs dans les environnements de conteneurs Java EE. Si le CDI n'est pas activé,
le fournisseur de persistance ne doit pas appeler de convertisseurs d'attributs qui dépendent de CDI
injection.

> Remarque CDI et JSF prennent en charge une portée de conversation dédiée, qui permet de naviguer entre plusieurs vues et pages client. Nous renvoyons les lecteurs à livres dédiés à CDI et JSF pour en savoir plus sur la portée de la conversation et comment JSF et CDI interagissent les uns avec les autres lors de l'utilisation de cette étendue.

Épisode 282

Le modèle de conversation sera utile non seulement lorsque vous souhaitez modifier
plusieurs objets, voire plusieurs types d'objets, mais lorsque vous ne le souhaitez pas nécessairement
le bean session doit mettre en cache ou garder une trace de tout ce qui a été
édité tout au long de la conversation. Toutes les entités qui ont été modifiées par le client sont
géré directement par le contexte de persistance JPA.

Dans la liste 6-37 , le bean session avec état fournit une méthode de chargement pour que le servlet
charger un employé et donner à l'utilisateur la possibilité d'apporter des modifications à cet employé.
L'utilisateur peut ensuite enregistrer les modifications ou annuler ces modifications individuelles des employés.
Ce processus peut être répété un nombre arbitraire de fois jusqu'à ce que l'utilisateur décide de
soit accepter les changements pour tous les employés, soit les abandonner. S'il est accepté, le
les modifications sont jointes à la transaction démarrée dans le cadre de processAllChanges ()
méthode et validée lorsque la méthode et la transaction sont terminées.

Annonce 6-37. Bean session avec état avec contexte de persistance non synchronisé

```
@Stateful
@SessionScoped
public class EmployeeService {
    @PersistenceContext (type = PersistenceContextType.EXTENDED,
                                synchronisation = SynchronizationType.UNSYNCHRONIZED,
                                unitName = "EmployeeService")
    EntityManager em;
```

```
    Employé actuelEmployé;

    public Employee getCurrentEmployee () {retourne currentEmployee; }

    public Employee loadEmployee (int id) {
        Employé emp = em.find (Employee.class, id);
        currentEmployee = emp;
        return emp;
    }

    public void saveChangesToEmployee () {}

    public void cancel () {
        em.detach (currentEmployee);
    }
```


Épisode 283


```
    public void processAllChanges () {
        em.joinTransaction ();
    }

    public void abandonAllChanges () {
        em.clear ();
    }
}
```

En regardant la méthode saveChangesToEmployee (), vous pouvez penser que du code manque car la méthode semble ne rien faire. La raison pour laquelle il n'y a pas de code dans la méthode est parce qu'il n'a pas besoin de faire quoi que ce soit. Le loadEmployee () renvoie l'employé géré à partir du contexte de persistance afin que l'employé en cours de modification au niveau de la couche de présentation est déjà suivi par la persistance fournisseur. En revanche, si l'utilisateur annule les modifications, nous détacherons simplement l'objet du contexte de persistance pour garantir que ces modifications ne seront pas validées.

Lorsque l'utilisateur décide de mettre fin à la conversation en acceptant toutes les modifications, le La méthode processAllChanges () effectue la jointure de la transaction. Si l'utilisateur décide de mettre fin en supprimant toutes les modifications, la méthode abandonAllChanges () efface toutes les objets du contexte de persistance et la conversation peut recommencer.

Avant de quitter cet exemple, nous faisons un dernier commentaire sur la portée de la session que nous avons utilisé. Nous avons utilisé CDI et avons étendu le bean EmployeeService à la session afin il sera disponible pour le thread à portée de session demandeur. En conséquence, nous aurons entités restantes dans le contexte de persistance lorsqu'une conversation est validée, car le contexte n'est effacé que lorsque les modifications sont abandonnées. C'est correct tant que la séance ne se poursuit pas excessivement. Si la session dure très longtemps ou si les entités dans différentes conversations ne se chevauchent jamais, le contexte de persistance devrait probablement être effacé. Cependant, si vous êtes tenté d'ajouter simplement un appel à em.clear () dans le processAllChanges () après l'appel joinTransaction (), vous seriez surprise désagréable. Si les modifications ont été effacées avant le processAllChanges () est terminée, puis lorsque la transaction gérée par le conteneur est validée, à la à la fin de cette méthode, il n'y aura aucun changement dans le contexte de persistance à valider! UNE la solution correcte serait d'effacer le contexte de persistance au début du prochain conversation à la place.

Épisode 284

Résumé

Dans ce chapitre, nous avons présenté un traitement approfondi du gestionnaire d'entité et de ses interactions avec des entités, des contextes de persistance et des transactions. Comme vous l'avez vu, le Le gestionnaire d'entités peut être utilisé de différentes manières pour s'adapter à une grande variété de les exigences de l'application.

Nous avons commencé par réintroduire la terminologie de base de JPA et exploré la persistance le contexte. Nous avons ensuite couvert les différents types de gestionnaires d'entités: transactionnels, étendu et géré par l'application. Nous avons examiné comment acquérir et utiliser chaque type et les types de problèmes qu'ils sont censés résoudre.

Dans la section de gestion des transactions, nous avons examiné chacun des responsables d'entités types et comment ils sont liés aux transactions JTA gérées par le conteneur et à la ressource locale transactions du pilote JDBC. Nous avons illustré pourquoi les transactions jouent un rôle important dans tous les aspects du développement d'applications d'entreprise avec JPA. Nous avons montré un genre spécial du contexte de persistance qui reste non synchronisé avec la transaction JTA jusqu'à ce qu'elle soit joint par programme, et comment il peut apporter de la flexibilité pour une interaction plus longue séquences.

Ensuite, nous avons revisité les opérations de base du gestionnaire d'entités, cette fois armé avec une compréhension complète des différents types de gestionnaires d'entités et de transactions stratégies de gestion. Nous avons introduit la notion de cascade et examiné l'impact des relations sur la persistance.

Dans notre discussion sur le détachement, nous avons présenté le problème et l'avons examiné à la fois du point de vue des entités mobiles vers les niveaux distants et le défi de la fusion l'entité hors ligne revient dans un contexte de persistance. Nous avons présenté plusieurs stratégies pour minimiser l'impact du détachement et de la fusion sur la conception des applications en adoptant modèles de conception spécifiques à JPA.

Dans le chapitre suivant, nous nous intéresserons aux fonctionnalités de requête de JPA, en montrant comment pour créer, exécuter et utiliser les résultats des opérations de requête.

Épisode 285

CHAPITRE 7

Utilisation des requêtes

Pour la plupart des applications d'entreprise, l'extraction des données de la base de données est au moins aussi importante

comme la possibilité d'insérer de nouvelles données. De la recherche au tri, en passant par l'analyse et les activités
intelligence, déplacement efficace des données de la base de données vers l'application et présentation
cela à l'utilisateur fait partie intégrante du développement de l'entreprise. Cela nécessite la capacité
pour émettre des requêtes groupées sur la base de données et interpréter les résultats pour l'application.
Bien que les langages de haut niveau et les cadres d'expression aient dans de nombreux cas
a tenté d'isoler les développeurs de la tâche de traiter les requêtes de base de données au
niveau SQL, il est probablement juste de dire que la plupart des développeurs d'entreprise ont
au moins un dialecte SQL à un moment donné de leur carrière.

Le mappage objet-relationnel ajoute un autre niveau de complexité à cette tâche. La plupart
heure, le développeur voudra que les résultats soient convertis en entités afin que les résultats de la requête
peut être utilisé directement par la logique d'application. De même, si le modèle de domaine a été
abstraite du modèle physique via un mappage objet-relationnel, il est logique de
requêtes abstraites loin de SQL, qui n'est pas seulement lié au modèle physique mais aussi
difficile à porter entre les vendeurs. Heureusement, comme vous le verrez, JPA peut gérer divers
ensemble d'exigences de requête.

JPA prend en charge deux méthodes pour exprimer des requêtes pour récupérer des entités et d'autres
données persistantes de la base de données: langages de requête et API Criteria. Le primaire
Le langage de requête est Java Persistence Query Language (JP QL), un langage indépendant de la base de données
langage de requête qui fonctionne sur le modèle d'entité logique par opposition au modèle physique
modèle de données. Les requêtes peuvent également être exprimées en SQL pour tirer parti du sous-jacent
base de données. Nous explorons l'utilisation des requêtes SQL avec JPA au chapitre 11 . L'API Criteria
fournit une méthode alternative pour construire des requêtes basées sur des objets Java au lieu de
chaînes de requête. Chapitre9 couvre l'API Criteria en détail.

Nous commençons notre discussion sur les requêtes par une introduction à JP QL, suivie d'un
exploration des fonctionnalités de requête fournies par les interfaces EntityManager et Query.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_7

Épisode 286

Langage de requête de persistance Java

Avant de parler de JP QL, nous devons d'abord nous pencher sur ses racines. La requête Enterprise JavaBeans
Le langage (EJB QL) a été introduit dans la spécification EJB 2.0 pour permettre aux développeurs de
écrire un finder portable et sélectionner des méthodes pour les beans entité gérés par des conteneurs. Basé sur
un petit sous-ensemble de SQL, il a introduit un moyen de naviguer à travers les relations d'entité à la fois
pour sélectionner des données et filtrer les résultats. Malheureusement, il a imposé des limites strictes à la
structure de la requête, limitant les résultats à une seule entité ou à un champ persistant de
une entité. Les jointures internes entre entités étaient possibles, mais utilisaient une notation étrange. le
la version initiale ne prenait même pas en charge le tri.

La spécification EJB 2.1 a légèrement modifié EJB QL, ajoutant la prise en charge du tri et
introduit des fonctions d'agrégation de base; mais encore une fois la limitation d'un seul type de résultat
entravé l'utilisation d'agrégats. Vous pouviez filtrer les données, mais il n'y avait pas d'équivalent à
Expressions SQL GROUP BY et HAVING.

JP QL étend considérablement EJB QL, éliminant de nombreuses faiblesses du précédent
versions tout en préservant la compatibilité descendante. La liste suivante décrit certains des
les fonctionnalités disponibles au-dessus et au-delà de EJB QL:

- Types de résultats à valeur unique et multiple
- Fonctions d'agrégation, avec des clauses de tri et de regroupement
- Une syntaxe de jointure plus naturelle, avec prise en charge à la fois
  jointures externes
- Expressions conditionnelles impliquant des sous-requêtes
- Mettre à jour et supprimer des requêtes pour les modifications de données en masse

• Projection des résultats dans des classes non persistantes

Les prochaines sections fournissent une introduction rapide à JP QL destinée aux lecteurs familiers avec SQL ou EJB QL. Un tutoriel complet et une référence pour JP QL se trouvent dans le chapitre8 .

Commencer

La requête JP QL la plus simple sélectionne toutes les instances d'un seul type d'entité. Prendre en compte requête suivante:

```
SÉLECTIONNER e
DE Employé e
```


Épisode 287

Si cela ressemble à SQL, il devrait. JP QL utilise la syntaxe SQL lorsque cela est possible dans afin de donner aux développeurs expérimentés avec SQL une longueur d'avance dans l'écriture de requêtes. le la principale différence entre SQL et JP QL pour cette requête est qu'au lieu de sélectionner une table, une entité du modèle de domaine d'application a été spécifiée à la place. le La clause SELECT de la requête est également légèrement différente, ne répertoriant que l'alias Employee e. Cela indique que le type de résultat de la requête est l'entité Employee, donc l'exécution de cette Cette instruction aboutira à une liste de zéro ou plusieurs instances Employee.

En commençant par un alias, nous pouvons naviguer dans les relations d'entité à l'aide du point (.) opérateur. Par exemple, si nous voulons uniquement les noms des employés, la requête suivante suffira:

```
SELECT e.name
DE Employé e
```

Chaque partie de l'expression correspond à un champ persistant de l'entité qui est un type simple ou intégrable, ou une association menant à une autre entité ou collection de entités. Étant donné que l'entité Employee a un champ persistant nommé name de type String, cette requête aboutira à une liste de zéro ou plusieurs objets String.

Nous pouvons également sélectionner une entité que nous n'avons même pas répertoriée dans la clause FROM. Prendre en compte exemple suivant:

```
SELECT e.department
DE Employé e
```

Une employée a une relation plusieurs-à-un avec son service nommé department, le type de résultat de la requête est donc l'entité Department.

Filtrage des résultats

Tout comme SQL, JP QL prend en charge la clause WHERE pour définir des conditions sur les données revenu. La majorité des opérateurs couramment disponibles en SQL sont disponibles en JP QL, y compris les opérateurs de comparaison de base; Expressions IN, LIKE et BETWEEN; nombreux expressions de fonction (telles que SUBSTRING et LENGTH); et sous-requêtes. La clé la différence pour JP QL est que des expressions d'entité et non des références de colonne sont utilisées. le Voici un exemple de filtrage à l'aide d'expressions d'entité dans la clause WHERE:

```
SÉLECTIONNER e
DE Employé e
```

Épisode 288


```
WHERE e.department.name = 'NA42' ET
        e.address.state IN ('NY', 'CA')
```

Projection des résultats

Pour les applications qui doivent produire des rapports, un scénario courant consiste à sélectionner nombre d'instances d'entité, mais n'utilisant qu'une partie de ces données. Selon comment une entité est mappée à la base de données, cela peut être une opération coûteuse si une grande partie du les données d'entité sont supprimées. Il serait utile de ne renvoyer qu'un sous-ensemble des propriétés de une entité. La requête suivante montre la sélection uniquement du nom et du salaire de chaque Instance d'employé:

```
SELECT e.name, e.salary
DE Employé e
```

Jointures entre entités

Le type de résultat d'une requête de sélection ne peut pas être une collection; il doit s'agir d'un objet à valeur unique comme une instance d'entité ou un type de champ persistant. Les expressions telles que les e.phones sont illégale dans la clause SELECT car elles entraîneraient des instances de Collection (chacune l'occurrence de e.phones est une collection, pas une instance). Par conséquent, tout comme avec SQL et tables, si nous voulons naviguer le long d'une association de collection et en renvoyer des éléments collection, nous devons joindre les deux entités ensemble. Dans le code suivant, une jointure entre Les entités Employee et Phone sont implicitement appliquées afin de récupérer tout le téléphone portable numéros pour un département spécifique:

```
SELECT p. Numéro
FROM Employé e, Téléphone p
O e = p. Employé ET
        e.department.name = 'NA42' ET
        p.type = 'Cellule'
```

Dans JP QL, les jointures peuvent également être exprimées dans la clause FROM à l'aide de l'opérateur JOIN. le l'avantage de cet opérateur est que la jointure peut être exprimée en termes d'association lui-même, et le moteur de requête fournira automatiquement les critères de jointure nécessaires lorsque

Épisode 289

il génère le SQL. La requête précédente peut être réécrite pour utiliser l'opérateur JOIN. Juste comme précédemment, l'alias p est de type Phone, mais cette fois il se réfère à chacun des téléphones du Collection e.phones:

```
SELECT p. Numéro
FROM Employé e JOIN e.phones p
WHERE e.department.name = 'NA42' ET
        p.type = 'Cellule'
```

JP QL prend en charge plusieurs types de jointures, y compris les jointures internes et externes, ainsi que une technique appelée *extraction de jointures* pour charger rapidement les données associées au type de résultat d'une requête mais pas directement renvoyée. Voir la section «Jointures» dans le chapitre8 pour plus information.

Requêtes agrégées

La syntaxe des requêtes agrégées dans JP QL est très similaire à celle de SQL. Ils sont cinq fonctions d'agrégation prises en charge (AVG, COUNT, MIN, MAX et SUM) et les résultats peuvent être regroupés dans la clause GROUP BY et filtrés à l'aide de la clause HAVING. Encore une fois, le La différence est l'utilisation d'expressions d'entité lors de la spécification des données à agréger. Un la requête agrégée JP QL peut utiliser de nombreuses fonctions d'agrégation dans la même requête:

```
SELECT d, COUNT (e), MAX (e.salary), AVG (e.salary)
DU Département d REJOINDRE d. Employés e
GROUPE PAR D
COMPTE (e)> = 5
```

Paramètres de requête

JP QL prend en charge deux types de syntaxe de liaison de paramètres. Le premier est la liaison de position, où les paramètres sont indiqués dans la chaîne de requête par un point d'interrogation suivi du numéro de paramètre. Lorsque la requête est exécutée, le développeur spécifie le paramètre numéro à remplacer. La syntaxe des paramètres de position est similaire à celle de JDBC prend actuellement en charge.

Épisode 290


```
SÉLECTIONNER e
DE Employé e
O e.department =? 1 ET
        e.salaire>? 2
```

Les paramètres nommés peuvent également être utilisés et sont indiqués dans la chaîne de requête par un signe deux-points suivi du nom du paramètre. Lorsque la requête est exécutée, le développeur spécifie le nom du paramètre à remplacer. Ce type de paramètre permet plus spécificateurs de paramètres descriptifs, comme ceci:

```
SÉLECTIONNER e
DE Employé e
WHERE e.department =: dept AND
        e.salary>: base
```

Définition des requêtes

JPA fournit les interfaces Query et TypedQuery pour configurer et exécuter des requêtes. le L'interface de requête est utilisée dans les cas où le type de résultat est Object ou dans les requêtes dynamiques lorsque le type de résultat peut ne pas être connu à l'avance. L'interface TypedQuery est le préféré et peut être utilisé chaque fois que le type de résultat est connu. Comme TypedQuery étend la requête, une requête fortement typée peut toujours être traitée comme une version non typée, mais pas l'inverse. Une implémentation de l'interface appropriée pour une requête donnée est obtenu via l'une des méthodes de fabrique de l'interface EntityManager. le le choix de la méthode d'usine dépend du type de requête (JP QL, SQL ou objet de critères), si la requête a été prédéfinie et si des résultats fortement typés sont souhaités.
Pour l'instant, nous limitons notre discussion aux requêtes JP QL. La définition de requête SQL est abordée dans Chapitre 11, et les requêtes de critères sont traitées au chapitre 9.

Il existe trois approches pour définir une requête JP QL. Une requête peut être spécifié dynamiquement au moment de l'exécution, configuré dans les métadonnées d'unité de persistance (annotation

ou XML) et référencé par nom, ou spécifié et enregistré dynamiquement pour être plus tard
référencé par son nom. Les requêtes JP QL dynamiques ne sont rien de plus que des chaînes, et
peut donc être défini à la volée en fonction des besoins. Requêtes nommées, d'autre part
main, sont statiques et immuables, mais sont plus efficaces à exécuter car les
le fournisseur de persistance peut traduire la chaîne JP QL en SQL une fois lorsque l'application
démarre au lieu de chaque fois que la requête est exécutée. Définition dynamique d'une requête et

Épisode 291

puis le nommer permet à une requête dynamique d'être réutilisée plusieurs fois tout au long de la vie de
l'application mais n'encourent qu'une seule fois le coût de traitement dynamique.

Les sections suivantes comparent les approches et discutent quand il faut
utilisé à la place des autres.

Définition de requête dynamique

Une requête peut être définie dynamiquement en passant la chaîne de requête JP QL et le résultat attendu
tapez à la méthode createQuery () de l'interface EntityManager. Le type de résultat peut être
omis pour créer une requête non typée. Nous discutons de cette approche dans la section «Utilisation des requêtes
Section Résultats ». Il n'y a aucune restriction sur la définition de la requête. Tous les types de requêtes JP QL
sont pris en charge, ainsi que l'utilisation de paramètres. La possibilité de créer une chaîne à l'exécution
et l'utiliser pour une définition de requête est utile, en particulier pour les applications où l'utilisateur peut
spécifiez des critères complexes et la forme exacte de la requête ne peut pas être connue à l'avance.
Comme indiqué précédemment, en plus des requêtes de chaînes dynamiques, JPA prend également en charge une API Criteria pour
créer des requêtes dynamiques à l'aide d'objets Java. Nous discutons de cette approche au chapitre 9.

Un problème à prendre en compte avec les requêtes dynamiques de chaîne, cependant, est le coût de la traduction
la chaîne JP QL à SQL pour exécution. Un moteur de requête typique devra analyser le JP QL
chaîne dans une arborescence de syntaxe, récupère les métadonnées de mappage relationnel objet pour chaque entité dans
chaque expression, puis générez le SQL équivalent. Pour les applications qui émettent de nombreux
requêtes, le coût des performances du traitement dynamique des requêtes peut devenir un problème.

De nombreux moteurs de requête mettront en cache le SQL traduit pour une utilisation ultérieure, mais cela peut facilement être
annulé si l'application n'utilise pas la liaison de paramètres et concatène le paramètre
valeurs directement dans les chaînes de requête. Cela a pour effet de générer un nouveau et unique
query chaque fois qu'une requête nécessitant des paramètres est construite.

Considérez la méthode bean [1] illustrée dans le Listing 7-1 qui recherche des informations sur le salaire
étant donné le nom d'un service et le nom d'un employé. Il y a deux problèmes
avec cet exemple, un lié aux performances et un lié à la sécurité. Parce que
les noms sont concaténés dans la chaîne au lieu d'utiliser la liaison de paramètres, c'est
créer efficacement une nouvelle et unique requête à chaque fois. Cent appels à cette méthode
pourrait potentiellement générer une centaine de chaînes de requête différentes. Cela nécessite non seulement
une analyse excessive de JP QL mais rend également presque certainement difficile la persistance
provider s'il tente de créer un cache de requêtes converties.

[1] Comme dans la plupart des exemples du livre, le bean peut être un bean session, un bean CDI ou tout autre type
de conteneur bean prenant en charge l'injection de gestionnaire d'entités.

Épisode 292

Liste 7-1. Définition dynamique d'une requête

```
classe publique QueryService {
    @PersistenceContext (unitName = "DynamicQueries")
    EntityManager em;

    public long queryEmpSalary (String deptName, String empName) {
        Requête de chaîne = "SELECT e.salary" +
                            "FROM Employé e" +
                            "WHERE e.department.name = '" + deptName +
                            "'ET" +
                            "e.name = '" + empName + "'";
        return em.createQuery (requête, Long.class) .getSingleResult ();
    }
}
```

Le deuxième problème avec cet exemple est qu'il est vulnérable aux attaques par injection, où un utilisateur malveillant pourrait transmettre une valeur qui modifie la requête à son avantage. Prenons un cas où l'argument du département a été corrigé par l'application mais l'utilisateur a pu spécifier le nom de l'employé (le responsable du service demande les salaires de ses employés, par exemple). Si l'argument de nom était en fait le texte '_UNKNOWN' OU e.name = 'Roberts', la requête réelle analysée par la requête moteur serait comme suit:

```
SELECT e.salary
DE Employé e
WHERE e.department.name = 'NA65' ET
        e.name = '_UNKNOWN' OU
        e.name = 'Roberts'
```

En introduisant la condition OU, l'utilisateur s'est effectivement donné accès au la valeur salariale de tout employé de l'entreprise car la condition ET d'origine a un priorité plus élevée que OR, et le faux nom d'employé n'appartiendra probablement pas à un employé de ce service.

Épisode 293

Ce type de problème peut sembler peu probable, mais dans la pratique, de nombreuses applications Web prendre le texte soumis sur une requête GET ou POST et construire aveuglément des requêtes de ce type sans considérer les effets secondaires. Une ou deux tentatives qui aboutissent à une pile d'analyseurs trace affichée sur la page Web, et l'attaquant apprendra tout ce qu'il doit savoir sur la façon de modifier la requête à son avantage.

Référencement 7-2 montre la même méthode que dans la liste7-1, sauf qu'il utilise named paramètres à la place. Cela réduit non seulement le nombre de requêtes uniques analysées par le moteur de requête, mais il élimine également le risque de modification de la requête.

Liste 7-2. Utilisation de paramètres avec une requête dynamique

```
classe publique QueryService {
    Chaîne finale statique privée QUERY =
        "SELECT e.salary" +
        "FROM Employé e" +
        "WHERE e.department.name =: deptName AND" +
        "e.name =: empName";
```

```
    @PersistenceContext (unitName = "QueriesUnit")
    EntityManager em;

    public long queryEmpSalary (String deptName, String empName) {
        retourne em.createQuery (QUERY, Long.class)
                .setParameter ("deptName", deptName)
                .setParameter ("empName", empName)
                .getSingleResult ();
    }
}
```

L'approche de liaison de paramètres présentée dans la liste 7-2 vainc la menace de sécurité décrit précédemment car la chaîne de requête d'origine n'est jamais modifiée. Les paramètres sont marshalés à l'aide de l'API JDBC et gérés directement par la base de données. Le texte d'un la chaîne de paramètres est effectivement citée par la base de données, de sorte que l'attaque malveillante finissent en fait par produire la requête suivante:

```
SELECT e.salary
DE Employé e
WHERE e.department.name = 'NA65' ET
        e.name = '_UNKNOWN' 'OU e.name =' 'Roberts'
```


Épisode 294

Les guillemets simples utilisés dans le paramètre de requête ici ont été échappés par préfixage avec un guillemet simple supplémentaire. Cela leur enlève toute signification particulière, et la séquence entière est traitée comme une seule valeur de chaîne.

Nous recommandons les requêtes nommées définies statiquement en général, en particulier pour requêtes qui sont exécutées fréquemment. Si les requêtes dynamiques sont une nécessité, veillez à utiliser liaison de paramètres au lieu de concaténer les valeurs de paramètres dans des chaînes de requête dans l'ordre pour minimiser le nombre de chaînes de requête distinctes analysées par le moteur de requête.

Définition de requête nommée

Les requêtes nommées sont un outil puissant pour organiser les définitions de requêtes et améliorer les performances des applications. Une requête nommée est définie à l'aide de l'annotation @NamedQuery, qui peut être placé sur la définition de classe pour toute entité. L'annotation définit le nom de la requête, ainsi que le texte de la requête. Référencement7-3 montre comment la chaîne de requête utilisée dans le Listing 7-2 serait déclaré comme une requête nommée.

Liste 7-3. Définition d'une requête nommée

```
@NamedQuery (name = "findSalaryForNameAndDepartment",
                query = "SELECT e.salary" +
                        "FROM Employé e" +
                        "WHERE e.department.name =: deptName AND" +
                        "e.name =: empName")
```

Les requêtes nommées sont généralement placées sur la classe d'entité qui correspond le plus directement au résultat de la requête, donc l'entité Employee serait un bon emplacement pour ce nommé requete. Notez l'utilisation de la concaténation de chaînes dans la définition d'annotation. Mise en page vos requêtes contribuent visuellement à la lisibilité de la définition de requête. La poubelle normalement associé à une concaténation de chaînes répétée ne s'appliquera pas ici car le l'annotation ne sera traitée qu'une seule fois au démarrage et sera exécutée au moment de l'exécution sous forme de requête.

Le nom de la requête porte sur l'unité de persistance entière et doit être unique dans ce cadre. Il s'agit d'une restriction importante à garder à l'esprit, car elle est couramment utilisée les noms de requête tels que "findAll" devront être qualifiés pour chaque entité. Un commun

Épisode 295

la pratique consiste à faire précéder le nom de la requête du nom de l'entité. Par exemple, le "findAll" la requête pour l'entité Employee serait nommée "Employee.findAll". C'est indéfini que se passe-t-il si deux requêtes dans la même unité de persistance ont le même nom, mais il est probable que le déploiement de l'application échouera ou que l'un écrasera le autre, conduisant à des résultats imprévisibles à l'exécution.

Si plus d'une requête nommée doit être définie sur une classe, elles doivent être placées à l'intérieur d'une annotation @NamedQueries, qui accepte un tableau d'un ou plusieurs @NamedQuery annotations. Le listing 7-4 montre la définition de plusieurs requêtes liées à l'employé entité. Les requêtes peuvent également être définies (ou redéfinies) à l'aide de XML. Cette technique est discuté au chapitre 13.

Liste 7-4. Plusieurs requêtes nommées pour une entité

```
@NamedQueries ({
    @NamedQuery (nom = "Employee.findAll",
                 query = "SELECT e FROM Employé e"),
    @NamedQuery (nom = "Employee.findByPrimaryKey",
                 query = "SELECT e FROM Employee e WHERE e.id =: id"),
    @NamedQuery (name = "Employee.findByName",
                 query = "SELECT e FROM Employee e WHERE e.name =: name")
})
```

La chaîne de requête étant définie dans l'annotation, elle ne peut pas être modifiée par l'application au moment de l'exécution. Cela contribue à la performance de l'application et aide à prévenir le type de problèmes de sécurité abordés dans la section précédente. En raison de la nature statique de la chaîne de requête, tout critère supplémentaire requis pour le La requête doit être spécifiée à l'aide des paramètres de requête. Référencement7-5 démontre l'utilisation du createNamedQuery () appelle l'interface EntityManager pour créer et exécuter un requête qui nécessite un paramètre de requête.

Liste 7-5. Exécution d'une requête nommée

```
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;
```


Épisode 296


```
    public Employee findEmployeeByName (String name) {
        return em.createNamedQuery ("Employee.findByName",
                                     Employé.classe)
                 .setParameter ("nom", nom)
```

```
                .getSingleResult ();
    }

    // ...
}
```

Les paramètres nommés sont le choix le plus pratique pour les requêtes nommées car ils auto-documentent efficacement le code d'application qui appelle les requêtes. Positionnel Cependant, les paramètres sont toujours pris en charge et peuvent être utilisés à la place.

Requêtes nommées dynamiques

Une approche hybride consiste à créer dynamiquement une requête, puis à l'enregistrer en tant que requête nommée dans l'usine du gestionnaire d'entités. À ce stade, il devient comme n'importe quelle autre requête nommée qui peuvent avoir été déclarées statiquement dans les métadonnées. Bien que cela puisse sembler un bon compromis, il s'avère utile dans quelques cas précis seulement. L'avantage principal il propose s'il y a des requêtes qui ne sont pas connues avant l'exécution, mais qui sont ensuite réémises à plusieurs reprises. Une fois que la requête dynamique devient une requête nommée, elle ne supportera que le coût de traitement une fois. Il est spécifique à la mise en œuvre si ce coût est payé lorsque la requête est enregistrée en tant que requête nommée ou différée jusqu'à la première exécution.

Une requête dynamique peut être transformée en requête nommée à l'aide de la EntityManagerFactory méthode addNamedQuery (). Référencement7-6 montre comment cela est fait.

Annonce 7-6. Ajout dynamique d'une requête nommée

```
classe publique QueryService {
    Chaîne finale statique privée QUERY =
        "SELECT e.salary" +
        "FROM Employé e" +
        "WHERE e.department.name =: deptName AND" +
        "e.name =: empName";
```



```
    @PersistenceContext (unitName = "QueriesUnit")
    EntityManager em;

    @PersistenceUnit (unitName = "QueriesUnit")
    EntityManagerFactory emf;

    @PostConstruct
    public void init () {
        TypeQuery <Long> q = em.createQuery (QUERY, Long.class);
        emf.addNamedQuery ("findSalaryForNameAndDepartment", q);
    }

    public long queryEmpSalary (String deptName, String empName) {
        return em.createNamedQuery ("findSalaryForNameAndDepartment", Long.
        classe)
                .setParameter ("deptName", deptName)
                .setParameter ("empName", empName)
                .getSingleResult ();
    }

    // ...
}
```

La méthode d'initialisation du bean, annotée avec @PostConstruct, crée le
requête dynamique et l'ajoute à l'ensemble des requêtes nommées. Comme mentionné précédemment, nommé
les requêtes sont étendues à l'unité de persistance entière, il est donc logique qu'elles soient ajoutées à
le niveau de la fabrique de gestionnaires d'entités. Cela nécessite également une certaine prudence dans le choix de
noms, car l'ajout d'une requête portant le même nom qu'une requête existante serait simplement
provoquer l'écrasement de l'existant. Parce que nous avions besoin du gestionnaire d'entités
factory dans les deux méthodes, nous l'avons juste injecté dans l'instance du bean en utilisant
@PersistenceUnit. On aurait pu tout aussi bien y accéder depuis le gestionnaire d'entités
en utilisant getEntityManagerFactory ().

Notez que la méthode addNamedQuery () a été ajoutée dans Jpa 2.1. s'il venait à
vous que l'exemple est un peu moins qu'utile, vous commencez à comprendre la raison
pourquoi l'utilisation de cette approche mixte est moins souvent nécessaire. si vous pouvez mettre la requête
description dans une chaîne statique au moment du développement, il est plus approprié de simplement
utilisez une annotation pour définir une requête nommée statique. les cas où cela pourrait être
Les avantages de créer une requête nommée à partir d'une requête dynamique sont les suivants:

- l'application a accès à certains critères lors de l'exécution
  contribue à une requête qui est prédéterminée pour être
  réalisé.
- une requête nommée est déjà définie mais en raison de certains aspects de
  l'environnement d'exécution dans lequel vous souhaitez remplacer la requête nommée
  avec un autre sans utiliser de descripteur XML supplémentaire.
- il existe une préférence pour définir toutes les requêtes dans le code au démarrage
  temps.

Conseil Lorsqu'une requête dynamique est ajoutée en tant que requête nommée, tous les paramètres [2]
sur la requête au moment où elle a été ajoutée sera enregistrée avec elle. chaque fois que le
la requête nommée est exécutée, les paramètres enregistrés s'appliqueront, sauf s'ils sont remplacés
au moment de l'exécution. cependant, comme pour toutes les autres requêtes nommées, tout paramètre effectué
à la requête nommée (c'est-à-dire la requête renvoyée par createNamedQuery ()) à
le temps d'exécution sera temporaire et ne s'appliquera qu'à cette exécution individuelle.

Types de paramètres

Comme mentionné précédemment, JPA prend en charge les paramètres nommés et positionnels pour JP QL
requêtes. Les méthodes de fabrique de requêtes du gestionnaire d'entités renvoient une implémentation
de l'interface de requête. Les valeurs des paramètres sont ensuite définies sur cet objet à l'aide du
méthodes setParameter () de l'interface Query.

[2] Les valeurs de paramètres qui ont peut-être déjà été liées à l'aide de setParameter () ne sont pas enregistrées sous
partie de la requête nommée.

Épisode 299

Il existe trois variantes de cette méthode pour les paramètres nommés et positionnels
paramètres. Le premier argument est toujours le nom ou le numéro du paramètre. La deuxième
argument est l'objet à lier au paramètre nommé. Date et calendrier
Les paramètres nécessitent également un troisième argument qui spécifie si le type passé à JDBC
est une valeur java.sql.Date, java.sql.Time ou java.sql.TimeStamp.

Considérez la définition de requête nommée suivante, qui nécessite deux
paramètres:

```
@NamedQuery (nom = "findEmployeesAboveSal",
                query = "SELECT e" +
                        "FROM Employé e" +
                        "WHERE e.department =: dept AND" +
                        "e.salary>: sal")
```

Cette requête met en évidence l'une des fonctionnalités intéressantes de JP QL en ce sens que les types d'entités peuvent être
utilisé comme paramètres. Lorsque la requête est traduite en SQL, la clé primaire nécessaire
les colonnes seront insérées dans l'expression conditionnelle et associées à la clé primaire
valeurs du paramètre. Il n'est pas nécessaire de savoir comment la clé primaire est mappée dans
afin d'écrire la requête. La liaison des paramètres de cette requête est un simple cas de passage
dans l'instance d'entité Department requise ainsi qu'un long représentant le minimum
valeur de salaire pour la requête. Référencement7-7 montre comment lier l'entité et la primitive
paramètres requis par cette requête.

Annonce 7-7. Lier des paramètres nommés

```
Liste publique <Employee> findEmployeesAboveSal (Department dept, long minSal) {
    return em.createNamedQuery ("findEmployeesAboveSal", Employee.class)
                                .setParameter ("dept", dept)
                                .setParameter ("sal", minSal)
                                .getResultList ();
}
```

Les paramètres de date et de calendrier sont un cas particulier car ils représentent les deux dates
et les temps. Au chapitre4, nous avons discuté de la cartographie des types temporels en utilisant le @Temporal
annotation et l'énumération TemporalType. Cette énumération indique si
le champ persistant est une date, une heure ou un horodatage. Lorsqu'une requête utilise une date ou un calendrier
paramètre, il doit sélectionner le type temporel approprié pour le paramètre. Référencement7-8
illustre les paramètres de liaison où la valeur doit être traitée comme une date.

Épisode 300

Annonce 7-8. Paramètres de date de liaison

```
Liste publique <Employee> findEmployeesHiredDuringPeriod (Date de début, Date de fin)
{
    retourne em.createQuery ("SELECT e" +
                                "FROM Employé e" +
                                "O e.startDate ENTRE? 1
                                ET? 2 ",
```

```
                                        Employé.classe)
                                .setParameter (1, début, TemporalType.DATE)
                                .setParameter (2, fin, TemporalType.DATE)
                                .getResultList ();
}
```

Un point à garder à l'esprit avec les paramètres de requête est que le même paramètre peut être utilisé plusieurs fois dans la chaîne de requête, mais ne doit être lié qu'une seule fois à l'aide du méthode setParameter (). Par exemple, considérez la définition de requête nommée suivante, où le paramètre "dept" est utilisé deux fois dans la clause WHERE:

```
@NamedQuery (name = "findHighestPaidByDepartment",
                query = "SELECT e" +
                        "FROM Employé e" +
                        "WHERE e.department =: dept AND" +
                        "e.salary = (SELECT MAX (e.salary)" +
                        "                       FROM Employé e "+
                        "                       WHERE e.department =: dept) ")
```

Pour exécuter cette requête, le paramètre "dept" ne doit être défini qu'une seule fois avec setParameter (), comme dans l'exemple suivant:

```
public Employee findHighestPaidByDepartment (Département département) {
    return em.createNamedQuery ("findHighestPaidByDepartment",
                                        Employé.classe)
                                .setParameter ("dept", dept)
                                .getSingleResult ();
}
```


Épisode 301

Exécuter des requêtes

Les interfaces Query et TypedQuery fournissent chacune trois façons différentes d'exécuter un requête, selon que la requête renvoie ou non des résultats et du nombre de résultats devrait être prévu. Pour les requêtes qui renvoient des valeurs, le développeur peut choisir de appelez soit getSingleResult () si la requête est censée renvoyer un seul résultat, soit getResultList () si plus d'un résultat peut être renvoyé. La méthode executeUpdate () est utilisé pour appeler une mise à jour en bloc et supprimer des requêtes. Nous discuterons de cette méthode plus tard dans le Section «Mise à jour et suppression groupées». Notez que les deux interfaces de requête définissent la même ensemble de méthodes et ne diffèrent que par leurs types de retour. Nous couvrons ce problème dans la section suivante.

La forme la plus simple d'exécution de requête est via la méthode getResultList (). Il retourne une collection contenant les résultats de la requête. Si la requête n'a renvoyé aucune donnée, le la collection est vide. Le type de retour est spécifié comme une liste au lieu d'une collection dans order pour prendre en charge les requêtes qui spécifient un ordre de tri. Si la requête utilise la clause ORDER BY pour spécifier un ordre de tri, les résultats seront placés dans la liste de résultats dans le même ordre. Référencement 7-9 montre comment une requête peut être utilisée pour générer un menu pour une commande-application en ligne qui affiche le nom de chaque employé travaillant sur un projet ainsi que le nom du service auquel l'employé est affecté. Les résultats sont triés par le nom de l'employé. Les requêtes ne sont pas ordonnées par défaut.

Liste 7-9. Itération sur des résultats triés

```
public void displayProjectEmployees (String projectName) {
    List <Employee> result = em.createQuery (
                                "SELECT e" +
```

```
                                    "FROM Project p JOIN p.employees e" +
                                    "WHERE p.name =? 1" +
                                    "ORDER BY e.name",
                                    Employé.classe)
                            .setParameter (1, projectName)
                            .getResultList ();
        nombre int = 0;
        for (Employé e: résultat) {
            System.out.println (++ count + ":" + e.getName () + "," +
                                    e.getDepartment (). getName ());
        }
    }
```

Épisode 302

La méthode getSingleResult () est fournie comme une commodité pour les requêtes qui renvoie une seule valeur. Au lieu d'itérer jusqu'au premier résultat d'une collection, le l'objet est directement renvoyé. Il est important de noter, cependant, que getSingleResult () se comporte différemment de getResultList () dans la façon dont il gère les résultats inattendus. Alors que getResultList () renvoie une collection vide quand aucun résultat n'est disponible, getSingleResult () lève une exception NoResultException. Par conséquent, s'il y a un chance que le résultat souhaité ne soit pas trouvé, alors cette exception doit être traitée.

Si plusieurs résultats sont disponibles après l'exécution de la requête au lieu du seul résultat attendu, getSingleResult () lancera une NonUniqueResultException exception. Encore une fois, cela peut être problématique pour le code d'application si les critères de requête peut entraîner le renvoi de plus d'une ligne dans certaines circonstances. Bien que getSingleResult () est pratique à utiliser, assurez-vous que la requête et ses résultats possibles sont bien compris; sinon le code de l'application devra peut-être faire face à un exception d'exécution. Contrairement aux autres exceptions lancées par les opérations du gestionnaire d'entités, ces les exceptions n'obligeront pas le fournisseur à annuler la transaction en cours, s'il y en a une.

Toute requête SELECT qui renvoie des données via getResultList () et Les méthodes getSingleResult () peuvent également spécifier des contraintes de verrouillage pour la base de données lignes impactées par la requête. Cette fonction est exposée via les interfaces de requête via le méthode setLockMode (). Nous reportons la discussion de la sémantique de verrouillage pour les requêtes jusqu'à la discussion complète du verrouillage dans le chapitre 12.

Les objets Query et TypedQuery peuvent être réutilisés aussi souvent que nécessaire à condition qu'ils soient identiques Le contexte de persistance utilisé pour créer la requête est toujours actif. Pour transaction- gestionnaires d'entités étendues, cela limite la durée de vie de l'objet Query ou TypedQuery à la vie de la transaction. D'autres types de gestionnaires d'entités peuvent les réutiliser jusqu'à ce que l'entité le gestionnaire est fermé ou supprimé.

Référencement 7-10 illustre la mise en cache d'une instance d'objet TypedQuery sur la classe bean d'un bean session avec état qui utilise un contexte de persistance étendu. Chaque fois que le haricot a besoin de trouver la liste des employés qui ne sont actuellement affectés à aucun projet, il réutilise le même objet unassignedQuery qui a été initialisé pendant PostConstruct.

Annonce 7-10. Réutilisation d'un objet de requête

```
@Stateful
La classe publique ProjectManagerBean implémente ProjectManager {
    @PersistenceContext (unitName = "EmployeeService",
                         type = PersistenceContextType.EXTENDED)
    EntityManager em;
```

Épisode 303


```
    TypedQuery <Employee> unassignedQuery;

    @PostConstruct
    public void init () {
        unassignedQuery =
            em.createQuery ("SELECT e" +
                            "FROM Employé e" +
                            "O e.projects EST VIDE",
                            Classe.employé);
    }

    Liste publique <Employee> findEmployeesWithoutProjects () {
        return unassignedQuery.getResultList ();
    }

    // ...
}
```

Utilisation des résultats de requête

Le type de résultat d'une requête est déterminé par les expressions répertoriées dans le SELECT clause de la requête. Si le type de résultat d'une requête est l'entité Employee, alors l'exécution getResultList () aboutira à une collection de zéro ou plusieurs instances d'entité Employee.
Il existe une grande variété de résultats possibles, en fonction de la composition de la requête. le Voici quelques-uns des types qui peuvent résulter des requêtes JP QL:

- Types de base, tels que String, les types primitifs et les types JDBC
- Types d'entités
- Un tableau d'objets
- Types définis par l'utilisateur créés à partir d'une expression de constructeur

Pour les développeurs habitués à JDBC, le point le plus important à retenir lors de l'utilisation les interfaces Query et TypedQuery est que les résultats ne sont pas encapsulés dans un ResultSet JDBC. La collection ou résultat unique correspond directement au type de résultat de la requête.

Épisode 304

Chaque fois qu'une instance d'entité est renvoyée, elle devient gérée par l'actif contexte de persistance. Si cette instance d'entité est modifiée et que le contexte de persistance fait partie d'une transaction, les modifications seront conservées dans la base de données. La seule exception à cela règle est l'utilisation de gestionnaires d'entités à portée de transaction en dehors d'une transaction. Toute requête exécuté dans cette situation renvoie des instances d'entités détachées au lieu d'entités gérées instances. Pour apporter des modifications à ces entités détachées, elles doivent d'abord être fusionnées dans un contexte de persistance avant de pouvoir être synchronisés avec la base de données.

Une conséquence de la gestion à long terme des entités avec application les contextes de persistance gérés et étendus sont que l'exécution de requêtes volumineuses le contexte de persistance à croître car il stocke toutes les instances d'entité gérées qui sont revenu. Si beaucoup de ces contextes de persistance s'accrochent à un grand nombre de entités gérées pendant de longues périodes, l'utilisation de la mémoire peut alors devenir un problème.

La méthode clear () de l'interface EntityManager peut être utilisée pour effacer l'application-
contextes de persistance gérés et étendus, supprimant les entités gérées inutiles.

Résultats de la requête de flux

Comme nous l'avons dit dans le premier chapitre de ce livre, l'un des changements de JPA 2.2 est la possibilité de
diffuser le résultat d'une exécution de requête.

Voici comment la méthode stream getResultStream () ajoutée à la requête et
Les interfaces TypedQuery recherchent dans JAP 2.2 lors de la création d'une classe pour récupérer les informations de salaire en tant que
résultat du flux.

À partir du Listing 7-9 , nous voulons démontrer l'utilisation de getResultStream ()
méthode pour obtenir les valeurs de salaire dans un format de flux.

```
public void displayProjectEmployees (String projectName) {
    salaire final AtomicInteger = 0;
    List <Employee> result = em.createQuery (
                                    "SELECT e" +
                                    "FROM Project p JOIN p.employees e" +
                                    "WHERE p.name =? 1" +
                                    "ORDER BY e.name",
                                    Employé.classe)
                            .setParameter (1, projectName)
```


Épisode 305


```
    Stream <emp> empStream = result.getResultStream ();
    empStream.forEach (e -> salaire.set (salaire.get () + e.getSalary ()));
    return salaire.get ();}
```

Notez que cela peut être très utile lorsque nous devons traiter un énorme ensemble de résultats.

Résultats non typés

Jusqu'à présent, dans ce chapitre, nous avons présenté les versions fortement typées du
méthodes de création de requêtes. Nous avons fourni le type de résultat attendu et donc
a reçu une instance de TypedQuery liée au type attendu. En qualifiant le
type de résultat de cette manière, les méthodes getResultList () et getSingleResult () retournent
les types corrects sans avoir besoin de couler.

Dans le cas où le type de résultat est Object, ou si la requête JP QL sélectionne plusieurs
objets, vous pouvez utiliser les versions non typées des méthodes de création de requête. Omettre le
result type produit une instance Query au lieu d'une instance TypedQuery, qui définit
getResultList () pour renvoyer une liste indépendante et getSingleResult () pour renvoyer Object.
Pour un exemple d'utilisation de résultats non typés, consultez les listes de codes dans la section «Résultat spécial
Section "Types".

Optimiser les requêtes en lecture seule

Lorsque les résultats de la requête ne seront pas modifiés, les requêtes utilisant une entité à portée de transaction
les gestionnaires en dehors d'une transaction peuvent être plus efficaces que les requêtes exécutées dans
une transaction lorsque le type de résultat est une entité. Lorsque les résultats de la requête sont préparés dans
une transaction, le fournisseur de persistance doit prendre des mesures pour convertir les résultats en
entités gérées. Cela implique généralement de prendre un instantané des données pour chaque entité dans
afin d'avoir une référence à comparer lorsque la transaction est validée. Si la
les entités gérées ne sont jamais modifiées, l'effort de convertir les résultats en

entités est gaspillée.

En dehors d'une transaction, dans certaines circonstances, le fournisseur de persistance peut être capable d'optimiser le cas où les résultats seront détachés immédiatement. Donc cela peut éviter la surcharge liée à la création des versions gérées. Notez que cette technique ne fonctionne pas sur les gestionnaires d'entités gérés par des applications ou étendues car leur le contexte de persistance survit à la transaction. Tout résultat de requête de ce type de le contexte de persistance peut être modifié pour une synchronisation ultérieure avec la base de données même si il n'y a pas de transaction.

Épisode 306

Lors de l'encapsulation d'opérations de requête derrière un bean avec gestion du conteneur transactions, le moyen le plus simple d'exécuter des requêtes non transactionnelles est d'utiliser le Attribut de transaction NOT_SUPPORTED pour la méthode du bean session. Cela causera toute transaction active à être suspendue, forçant le détachement des résultats de la requête et permettant cette optimisation. Le Listing 7-11 montre un exemple de cette technique utilisant un bean session sans état.

Annonce 7-11. Exécution d'une requête en dehors d'une transaction

```
@Apatride
classe publique QueryService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    @TransactionAttribute (TransactionAttributeType.NOT_SUPPORTED)
    Liste publique <Department> findAllDepartmentsDetached () {
        return em.createQuery ("SELECT d FROM Department d",
                                Département.classe)
                .getResultList ();
    }

    // ...
}
```

Notez que cette optimisation est entièrement spécifique au fournisseur. certains fournisseurs peuvent choisissez plutôt de créer un contexte de persistance temporaire pour la requête et lancez simplement après en avoir extrait les résultats, faisant de cette optimisation suggérée plutôt étranger. Vérifiez votre fournisseur avant de prendre une décision de codage.

Types de résultats spéciaux

Chaque fois qu'une requête implique plus d'une expression dans la clause SELECT, le résultat de la requête sera une liste de tableaux d'objets. Les exemples courants incluent la projection d'entité champs et requêtes agrégées où des expressions de regroupement ou plusieurs fonctions sont utilisées. Référencement 7-12 revisite le générateur de menu du Listing 7-9 en utilisant une requête de projection à la place

Épisode 307

de renvoyer des instances complètes d'entité Employee. Chaque élément de la liste est converti en un tableau de Objet qui est ensuite utilisé pour extraire les informations sur l'employé et le nom du service. nous utilisez une requête non typée car le résultat contient plusieurs éléments.

Annonce 7-12. Gestion de plusieurs types de résultats

```
public void displayProjectEmployees (String projectName) {
        Résultat de la liste = em.createQuery (
                                                "SELECT e.name, e.department.name" +
                                                "FROM Project p JOIN p.employees e" +
                                                "WHERE p.name =? 1" +
                                                "ORDER BY e.name")
                                        .setParameter (1, projectName)
                                        .getResultList ();
        nombre int = 0;
        for (Iterator i = result.iterator (); i.hasNext ();) {
                Object [] values = (Object []) i.next ();
                System.out.println (++ count + ":" +
                                                valeurs [0] + "," + valeurs [1]);
        }
}
```

Les expressions de constructeur fournissent aux développeurs un moyen de mapper un tableau d'objets types de résultats en objets personnalisés. En règle générale, il est utilisé pour convertir les résultats en JavaBean-classes de style qui fournissent des getters pour les différentes valeurs renvoyées. Cela rend les résultats plus facile à travailler et permet d'utiliser les résultats directement dans un environnement tels que JavaServer Faces sans traduction supplémentaire.

Une expression de constructeur est définie dans JP QL à l'aide de l'opérateur NEW dans SELECT clause. L'argument de l'opérateur NEW est le nom complet de la classe qui être instancié pour contenir les résultats de chaque ligne de données renvoyée. La seule exigence pour cette classe, c'est qu'elle a un constructeur avec des arguments correspondant au type exact et l'ordre qui sera spécifié dans la requête. Référencement7-13 montre une classe EmpMenu définie dans l'exemple de package qui pourrait être utilisé pour contenir les résultats de la requête qui était exécuté dans la liste 7-12 .

Épisode 308

Annonce 7-13. Définition d'une classe à utiliser dans une expression de constructeur

```
exemple de package;

public class EmpMenu {
        private String employeeName;
        private String departmentName;

        public EmpMenu (String employeeName, String departmentName) {
                this.employeeName = employeeName;
                this.departmentName = departmentName;
        }

        public String getEmployeeName () {return employeeName; }
        public String getDepartmentName () {return departmentName; }
}
```

Référencement 7-14 montre le même exemple que Listing7-12 utilisation du EmpMenu pleinement qualifié

nom de classe dans une expression de constructeur. Au lieu de travailler avec des index de tableau, chacun result est une instance de la classe EmpMenu et est utilisé comme un objet Java standard. nous pouvons utilisez également à nouveau des requêtes typées car il n'y a qu'une seule expression dans la clause SELECT.

Liste 7-14. Utilisation d'expressions de constructeur

```
public void displayProjectEmployees (String projectName) {
    Liste <EmpMenu> résultat =
        em.createQuery ("SELECT NEW example.EmpMenu (" +
                                                    "e.name, e.department.name)" +
                        "FROM Project p JOIN p.employees e" +
                        "WHERE p.name =? 1" +
                        "ORDER BY e.name",
                        EmpMenu.class)
            .setParameter (1, projectName)
            .getResultList ();
    nombre int = 0;
```



```
    for (Menu EmpMenu: résultat) {
        System.out.println (++ count + ":" +
                            menu.getEmployeeName () + "," +
                            menu.getDepartmentName ());
    }
}
```

Requête de pagination

Les jeux de résultats volumineux issus de requêtes posent souvent problème pour de nombreuses applications. Dans les cas où il serait écrasant d'afficher l'ensemble des résultats, ou si le support d'application rend l'affichage de nombreuses lignes inefficace (applications web notamment), applications doit pouvoir afficher les plages d'un ensemble de résultats et permettre aux utilisateurs de contrôler la plage de données qu'ils consultent. La forme la plus courante de cette technique est pour présenter à l'utilisateur un tableau de taille fixe qui agit comme une fenêtre glissante sur le résultat ensemble. Chaque incrément de résultats affichés est appelé une page et le processus de navigation à travers les résultats est appelée pagination.

La pagination efficace des ensembles de résultats a longtemps été un défi pour les deux applications développeurs et fournisseurs de bases de données. Avant la prise en charge au niveau de la base de données, un La technique courante consistait à récupérer d'abord toutes les clés primaires de l'ensemble de résultats et puis émettez des requêtes séparées pour les résultats complets en utilisant des plages de valeurs de clé primaire. Plus tard, les fournisseurs de bases de données ont ajouté le concept de numéro de ligne logique aux résultats des requêtes, garantissant que tant que le résultat était commandé, le numéro de ligne pouvait être invoqué pour récupérer des parties de l'ensemble de résultats. Plus récemment, la spécification JDBC a pris cette encore plus loin avec le concept d'ensembles de résultats déroulants, qui peuvent être parcourus vers l'avant et en arrière au besoin.

Les interfaces Query et TypedQuery prennent en charge la pagination via le méthodes setFirstResult () et setMaxResults (). Ces méthodes spécifient le premier résultat à recevoir (numéroté à partir de zéro) et le nombre maximum de résultats à retour par rapport à ce point. Les valeurs définies pour ces méthodes peuvent également être récupérées via les méthodes getFirstResult () et getMaxResults (). Un fournisseur de persistance peut choisir de mettre en œuvre la prise en charge de cette fonctionnalité de différentes manières car non

toutes les bases de données bénéficient de la même approche. C'est une bonne idée de se familiariser avec la façon dont votre fournisseur aborde la pagination et le niveau de support existant dans la cible plateforme de base de données pour votre application.

Épisode 310

Attention, les méthodes setFirstResult () et setMaxResults () ne doivent pas être utilisé avec les requêtes qui se rejoignent entre les relations de collection (un-à-plusieurs et plusieurs à plusieurs) car ces requêtes peuvent renvoyer des valeurs en double. le duplicata les valeurs de l'ensemble de résultats rendent impossible l'utilisation d'une position de résultat logique.

Pour mieux illustrer la prise en charge de la pagination, considérez le bean avec état indiqué dans la liste 7-15 . Une fois créé, il est initialisé avec le nom d'une requête pour compter le total des résultats et le nom d'une requête pour générer le rapport. Lorsque les résultats sont demandés, il utilise la taille de la page et le numéro de la page actuelle pour calculer les paramètres corrects pour setFirstResult () et les méthodes setMaxResults (). Le nombre total de résultats possibles est calculé par exécution de la requête de comptage. En utilisant les suivants (), previous () et getCurrentResults () méthodes, le code de présentation peut parcourir les résultats selon les besoins. Si ce haricot était lié à une session HTTP, il pourrait être directement utilisé par une page JSP ou JavaServer Faces présenter les résultats dans un tableau de données. La classe du Listing 7-15 est un modèle général pour un bean qui contient un état intermédiaire pour une requête d'application à partir de laquelle les résultats sont traitées en segments. Un bean session avec état est utilisé.

Liste 7-15. Pager de rapport avec état

```
@Stateful
classe publique ResultPager {
    @PersistenceContext (unitName = "QueryPaging")
    privé EntityManager em;

    private String reportQueryName;
    private long currentPage;
    private long maxResults;
    pageSize privée longue;

    public long getPageSize () {
        return pageSize;
    }

    public long getMaxPages () {
        return maxResults / pageSize;
    }
```


Épisode 311


```
    public void init (long pageSize, String countQueryName,
                      String reportQueryName) {
        this.pageSize = pageSize;
```

```
        this.reportQueryName = reportQueryName;
        maxResults = em.createNamedQuery (countQueryName, Long.class)
                                    .getSingleResult ();
        currentPage = 0;
    }

    Liste publique getCurrentResults () {
        renvoie em.createNamedQuery (reportQueryName)
                .setFirstResult (currentPage * pageSize)
                .setMaxResults (pageSize)
                .getResultList ();
    }

    public void next () {
        currentPage ++;
    }

    public void précédent () {
        page actuelle--;
        if (currentPage <0) {
            currentPage = 0;
        }
    }

    public long getCurrentPage () {
        return currentPage;
    }

    public void setCurrentPage (long currentPage) {
        this.currentPage = currentPage;
    }

    @Retirer
    public void done () {}
}
```


Épisode 312

Requêtes et modifications non validées

L'exécution de requêtes sur des entités qui ont été créées ou modifiées dans une transaction est un sujet qui nécessite une attention particulière. Comme nous l'avons vu au chapitre 6 , la persistance le fournisseur tentera de minimiser le nombre de fois où le contexte de persistance doit être vidé dans une transaction. Idéalement, cela ne se produira qu'une seule fois, lorsque la transaction s'engage. Pendant que la transaction est ouverte et que des modifications sont apportées, le fournisseur compte sur sa propre synchronisation de cache interne pour garantir que la bonne version de chaque entité est utilisé dans les opérations du gestionnaire d'entités. Tout au plus, le fournisseur devra peut-être lire de nouvelles données de la base de données afin de répondre à une demande. Toutes les opérations d'entité autres que les requêtes peut être satisfait sans vider le contexte de persistance dans la base de données.

Les requêtes sont un cas particulier car elles sont exécutées directement en tant que SQL sur le base de données. Étant donné que la base de données exécute la requête et non le fournisseur de persistance, le Le contexte de persistance active ne peut généralement pas être consulté par la requête. En conséquence, si le le contexte de persistance n'a pas été vidé et la requête de base de données serait affectée par les modifications en attente dans le contexte de persistance, des données incorrectes sont susceptibles d'être récupérées à partir de la requête. L'opération find () du gestionnaire d'entités, d'autre part, interroge un entité unique avec une clé primaire donnée. Il peut toujours vérifier le contexte de persistance avant aller à la base de données, donc des données incorrectes ne sont pas un problème.

La bonne nouvelle est que par défaut, le fournisseur de persistance s'assurera que les requêtes peuvent incorporer les modifications transactionnelles en attente dans le résultat de la requête. Ça pourrait accomplissez ceci en vidant le contexte de persistance dans la base de données, ou il peut tirer parti ses propres informations d'exécution pour garantir que les résultats sont corrects.

Et pourtant, il arrive que le fournisseur de persistance assure l'intégrité des requêtes n'est pas nécessairement le comportement dont vous avez besoin. Le problème est que ce n'est pas toujours facile pour le fournisseur pour déterminer la meilleure stratégie pour répondre aux besoins d'intégrité d'un requete. Il n'existe aucun moyen pratique pour le fournisseur de déterminer de manière logique niveau quels objets ont changé et doivent donc être incorporés dans la requête résultats. Si la solution du fournisseur pour garantir l'intégrité des requêtes consiste à vider la persistance contexte à la base de données, vous pourriez avoir un problème de performances si cela est fréquent occurrence.

Pour mettre ce problème en contexte, envisagez une application de forum de discussion, qui a sujets de conversation modélisés en tant qu'entités de conversation. Chaque entité de conversation fait référence à un ou plusieurs messages représentés par une entité Message. Périodiquement, des conversations sont archivés lorsque le dernier message ajouté à la conversation date de plus de 30 jours.

Épisode 313

Ceci est accompli en changeant le statut de l'entité Conversation de ACTIVE à INACTIF. Les deux requêtes pour obtenir la liste des conversations actives et le dernier message la date d'une conversation donnée est indiquée dans la liste 7-16 .

Annonce 7-16. Requêtes de conversation

```
@NamedQueries ({
    @NamedQuery (nom = "findActiveConversations",
                query = "SELECT c" +
                        "FROM Conversation c" +
                        "WHERE c.status = 'ACTIVE'"),
    @NamedQuery (name = "findLastMessageDate",
                query = "SELECT MAX (m.postingDate)" +
                        "FROM Conversation c JOIN c.messages m" +
                        "WHERE c =: conversation")
})
```

Référencement 7-17 montre la méthode utilisée pour effectuer cette maintenance, en acceptant un argument Date qui spécifie l'âge minimum pour les messages afin d'être toujours considéré comme une conversation active. Dans cet exemple, deux requêtes sont en cours d'exécution. La requête "findActiveConversations" recueille toutes les conversations actives, tandis que "findLastMessageDate" renvoie la dernière date à laquelle un message a été ajouté à une entité de conversation. Au fur et à mesure que le code parcourt la conversation entités, il appelle la requête "findLastMessageDate" pour chacune. Comme ces deux les requêtes sont liées, il est raisonnable pour un fournisseur de persistance de supposer que le les résultats de la requête "findLastMessageDate" dépendront des modifications faite aux entités Conversation. Si le fournisseur garantit l'intégrité du "findLastMessageDate" en vidant le contexte de persistance, cela pourrait devenir un opération très coûteuse si des centaines de conversations actives sont vérifiées.

Annonce 7-17. Archivage des entités de conversation

```
public void archiveConversations (Date minAge) {
    Liste <Conversation> active =
        em.createNamedQuery ("findActiveConversations",
                             Conversation.class)
            .getResultList ();
```

Épisode 314


```
        TypedQuery <Date> maxAge =
                em.createNamedQuery ("findLastMessageDate", Date.class);
        for (Conversation c: active) {
                maxAge.setParameter ("conversation", c);
                Date lastMessageDate = maxAge.getSingleResult ();
                if (lastMessageDate.before (minAge)) {
                        c.setStatus ("INACTIF");
                }
        }
}
```

Pour offrir plus de contrôle sur les exigences d'intégrité des requêtes, EntityManager et les interfaces de requête prennent en charge une méthode setFlushMode () pour définir le mode de vidage, un indiquez au fournisseur comment il doit gérer les modifications et les requêtes en attente. Là sont deux réglages de mode de rinçage possibles, AUTO et COMMIT, qui sont définis par le Type énuméré FlushModeType. Le paramètre par défaut est AUTO, ce qui signifie que le Le fournisseur doit s'assurer que les modifications transactionnelles en attente sont incluses dans les résultats de la requête. Si une requête peut chevaucher des données modifiées dans le contexte de persistance, ce paramètre assurez-vous que les résultats sont corrects. Le réglage actuel du mode de rinçage peut être récupéré via la méthode getFlushMode ().

Le mode de vidage COMMIT indique au fournisseur que les requêtes ne se chevauchent pas avec les modifications données dans le contexte de persistance, il n'a donc pas besoin de faire quoi que ce soit pour être correct résultats. Selon la manière dont le fournisseur implémente sa prise en charge de l'intégrité des requêtes, peut signifier qu'il n'a pas besoin de vider le contexte de persistance avant d'exécuter un requête car vous avez indiqué qu'aucune donnée modifiée en mémoire ne affectent les résultats de la requête de base de données.

Bien que le mode de vidage soit défini sur le gestionnaire d'entités, le mode de vidage est en réalité un propriété du contexte de persistance. Pour les gestionnaires d'entités à portée de transaction, cela signifie le mode de vidage doit être changé à chaque transaction. Étendu et application les gestionnaires d'entités gérées conserveront leur paramètre de mode de vidage entre les transactions. Le mode Flush ne s'appliquera pas du tout aux contextes de persistance qui ne sont pas synchronisés avec la transaction.

La définition du mode de vidage sur le gestionnaire d'entités s'applique à toutes les requêtes, lors de la définition le mode de vidage pour une requête limite le paramètre à cette portée. Réglage du mode de rinçage sur le La requête remplace le paramètre du gestionnaire d'entités, comme vous vous en doutez. Si le responsable de l'entité le paramètre est AUTO et une requête a le paramètre COMMIT, le fournisseur garantira la requête

Épisode 315

l'intégrité pour toutes les requêtes autres que celle avec le paramètre COMMIT. De même, si le le paramètre du gestionnaire d'entités est COMMIT et une requête a un paramètre AUTO, seule la requête avec le paramètre AUTO est garanti pour incorporer les modifications en attente de la persistance le contexte.

De manière générale, si vous allez exécuter des requêtes dans des transactions où les données est en cours de modification, AUTO est la bonne réponse. Si vous êtes préoccupé par la performance implications de la garantie de l'intégrité des requêtes, envisagez de changer le mode de vidage en COMMIT

sur une base par requête. La modification de la valeur sur le gestionnaire d'entités, bien que pratique, peut conduisent à des problèmes si plus de requêtes sont ajoutées à l'application ultérieurement et qu'elles nécessitent AUTO sémantique.

Pour en revenir à l'exemple au début de cette section, nous pouvons activer le mode flush l'objet TypedQuery pour la requête "findLastMessageDate" à COMMIT car il le fait pas besoin de voir les modifications apportées aux entités de conversation. Le suivant fragment montre comment cela serait accompli pour l'archiveConversations () méthode montrée dans l'extrait 7-17 :

```
public void archiveConversations (Date minAge) {
    // ...
    TypedQuery <Date> maxAge = em.createNamedQuery (
                                        "findLastMessageDate", Date.class);
    maxAge.setFlushMode (FlushModeType.COMMIT);
    // ...
}
```

Délais de requête

De manière générale, lorsqu'une requête s'exécute, elle se bloque jusqu'à ce que la requête de base de données Retour. En plus de la préoccupation évidente concernant les requêtes et les applications emballées réactivité, cela peut également être un problème si la requête participe à une transaction et un délai d'expiration a été défini sur la transaction JTA ou sur la base de données. Le timeout sur la transaction ou la base de données peut entraîner l'abandon prématuré de la requête, mais cela entraînera également transaction à annuler, empêchant tout autre travail dans la même transaction.

Si une application doit définir une limite de temps de réponse aux requêtes sans utiliser transaction ou provoquant une annulation de transaction, le javax.persistence.query.timeout La propriété peut être définie sur la requête ou dans le cadre de l'unité de persistance. Cette propriété définit le nombre de millisecondes pendant lequel la requête doit être autorisée à s'exécuter avant son abandon.

Épisode 316

Référencement 7-18 montre comment définir une valeur de délai d'expiration pour une requête donnée. Cet exemple utilise le mécanisme d'indication de requête, que nous aborderons plus en détail plus loin dans la section "Requête "Astuces". La définition des propriétés sur l'unité de persistance est traitée dans le chapitre13.

Annonce 7-18. Définition d'un délai d'expiration de requête

```
public Date getLastUserActivity () {
    TypedQuery <Date> lastActive =
        em.createNamedQuery ("findLastUserActivity", Date.class);
    lastActive.setHint ("javax.persistence.query.timeout", 5000);
    essayez {
        return lastActive.getSingleResult ();
    } catch (QueryTimeoutException e) {
        return null;
    }
}
```

Malheureusement, définir un délai d'expiration de requête n'est pas un comportement portable. Ce n'est peut-être pas pris en charge par toutes les plates-formes de base de données et il n'est pas nécessaire d'être pris en charge par tous fournisseurs de persistance. Par conséquent, les applications qui souhaitent activer les délais d'expiration des requêtes doit être préparé pour trois scénarios. Le premier est que la propriété est ignorée en silence et n'a aucun effet. La seconde est que la propriété est activée et que toute sélection, mise à jour, ou supprimer l'opération qui s'exécute plus longtemps que la valeur de délai spécifiée est abandonnée, et un QueryTimeoutException est levée. Cette exception peut être gérée et n'entraînera aucun

transaction active à marquer pour être annulée. Référencement7-18 montre une approche pour gérer cette exception. Le troisième scénario est que la propriété est activée, mais en faisant ainsi la base de données force une annulation de transaction lorsque le délai est dépassé. Dans ce cas, un PersistenceException sera levée et la transaction marquée pour la restauration. En général, si activé, l'application doit être écrite pour gérer l'exception QueryTimeoutException, mais ne doit pas échouer si le délai est dépassé et que l'exception n'est pas levée.

Mise à jour et suppression en masse

Comme leurs homologues SQL, les instructions JP QL bulk UPDATE et DELETE sont conçues pour apporter des modifications à un grand nombre d'entités en une seule opération sans nécessiter les entités individuelles à récupérer et à modifier à l'aide du gestionnaire d'entités. Contrairement à SQL,

Épisode 317

qui opère sur des tables, les instructions JP QL UPDATE et DELETE doivent prendre toute la plage de mappages pour l'entité en compte. Ces opérations sont difficiles pour les fournisseurs implémenter correctement, et par conséquent, il existe des restrictions sur l'utilisation de ces opérations cela doit être bien compris par les développeurs.

La syntaxe complète des instructions UPDATE et DELETE est décrite au chapitre 8 . le Les sections suivantes décrivent comment utiliser efficacement ces opérations et les problèmes peut en résulter en cas d'utilisation incorrecte.

Utilisation de la mise à jour et de la suppression en masse

La mise à jour groupée des entités est effectuée avec l'instruction UPDATE. Cette déclaration opère sur un seul type d'entité et définit une ou plusieurs propriétés à valeur unique du entité (soit un champ d'état, soit une association à valeur unique) sous réserve des conditions du Clause WHERE. Il peut également être utilisé à l'état incorporé dans un ou plusieurs objets incorporables référencé par l'entité. En termes de syntaxe, il est quasiment identique à la version SQL avec l'exception de l'utilisation d'expressions d'entité au lieu de tables et de colonnes. Référencement7-19 illustre l'utilisation d'une instruction UPDATE en masse. Notez que l'utilisation de REQUIRES_NEW Le type d'attribut de transaction est significatif et est décrit à la suite des exemples.

Annonce 7-19. Mise à jour groupée des entités

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "BulkQueries")
    EntityManager em;

    @TransactionAttribute (TransactionAttributeType.REQUIRES_NEW)
    public void assignManager (Département de service, Responsable des employés) {
        em.createQuery ("UPDATE Employee e" +
                        "SET e.manager =? 1" +
                        "WHERE e.department =? 2")
          .setParameter (1, gestionnaire)
          .setParameter (2, dept)
          .executeUpdate ();
    }
}
```

Épisode 318

La suppression groupée des entités est effectuée avec l'instruction DELETE. Encore une fois, le la syntaxe est la même que la version SQL, sauf que la cible dans la clause FROM est un entité au lieu d'une table, et la clause WHERE est composée d'expressions d'entité à la place d'expressions de colonne. Le listing 7-20 montre la suppression en masse d'entités.

Liste 7-20. Suppression groupée d'entités

```
@Apatride
public class ProjectService {
    @PersistenceContext (unitName = "BulkQueries")
    EntityManager em;

    @TransactionAttribute (TransactionAttributeType.REQUIRES_NEW)
    public void removeEmptyProjects () {
        em.createQuery ("SUPPRIMER DU Projet p" +
                        "O p.employés EST VIDE")
          .executeUpdate ();
    }
}
```

Le premier problème à prendre en compte lors de l'utilisation de ces instructions est que la persistance le contexte n'est pas mis à jour pour refléter les résultats de l'opération. Des opérations groupées sont émises en tant que SQL contre la base de données, en contournant les structures en mémoire de la persistance le contexte. Par conséquent, la mise à jour du salaire de tous les employés ne changera pas le valeurs pour toutes les entités gérées en mémoire dans le cadre d'un contexte de persistance. le le développeur ne peut compter que sur les entités récupérées une fois l'opération en bloc terminée.

Lors de l'utilisation de contextes de persistance à portée de transaction, l'opération en bloc doit soit exécuter une transaction toute seule, soit être la première opération de la transaction. L'exécution de l'opération en bloc dans sa propre transaction est l'approche préférée car il minimise le risque de récupérer accidentellement des données avant que le changement en bloc ne se produise. Exécuter l'opération en bloc, puis travailler avec des entités une fois qu'elle est terminée est également sûr, car toute opération ou requête find () ira à la base de données pour être mise à jour résultats. Les exemples dans la liste7-19 et liste7-20 a utilisé la transaction REQUIRES_NEW pour garantir que les opérations en bloc ont eu lieu dans leurs propres transactions.

Une stratégie typique pour les fournisseurs de persistance traitant des opérations en masse consiste à invalider tout cache en mémoire de données liées à l'entité cible. Cela force les données à être extraite de la base de données la prochaine fois qu'elle est requise. Combien de données mises en cache reçoivent invalidé dépend de la sophistication du fournisseur de persistance. Si le fournisseur

Épisode 319

peut détecter que la mise à jour n'a d'incidence que sur un petit nombre d'entités, ces entités spécifiques peut être invalidé, laissant les autres données mises en cache en place. De telles optimisations sont limitées, cependant, et si le fournisseur ne peut pas être sûr de la portée du changement, tout le cache doit être invalidé. Cela peut avoir un impact sur les performances de l'application si les changements de masse sont fréquents.

Attention, les opérations natives de mise à jour et de suppression de sQL ne doivent pas être exécutées sur tables mappées par une entité. les opérations Jp QL indiquent au fournisseur ce qui est mis en cache L'état de l'entité doit être invalidé afin de rester cohérent avec la base de données. les opérations sQL natives contournent ces vérifications et peuvent rapidement conduire à des situations

où le cache en mémoire est obsolète par rapport à la base de données.

Le danger présent dans les opérations en vrac et la raison pour laquelle ils doivent se produire en premier transaction est que toute entité gérée activement par un contexte de persistance restera de cette façon, inconscient des changements réels qui se produisent au niveau de la base de données. L'active le contexte de persistance est séparé et distinct de tout cache de données que le fournisseur peut utiliser pour les optimisations. Considérez la séquence d'opérations suivante:

1. Une nouvelle transaction démarre.
2. L'entité A est créée en appelant persist () pour rendre l'entité géré.
3. L'entité B est extraite d'une opération find () et modifiée.
4. Une suppression en bloc supprime l'entité A.
5. Une mise à jour groupée modifie les mêmes propriétés sur l'entité B qui étaient modifié à l'étape 3.
6. La transaction est validée.

Que devrait-il arriver aux entités A et B dans cette séquence? (Avant de répondre, rappelez-vous que les opérations en bloc se traduisent directement en SQL et contournent le contexte de persistance!) dans le cas de l'entité A, le fournisseur doit supposer que le contexte de persistance est correct et donc tentera toujours d'insérer la nouvelle entité même si elle aurait dû être supprimée. Dans le cas de l'entité B, encore une fois, le fournisseur doit supposer que la version gérée est la version correcte et tentera de mettre à jour la version dans la base de données, annulant le volume mettre à jour le changement.

Épisode 320

Cela nous amène à la question des contextes de persistance étendue. Opérations en vrac et les contextes de persistance étendue sont une combinaison particulièrement dangereuse car le le contexte de persistance survit au-delà des limites des transactions, mais le fournisseur ne actualiser le contexte de persistance pour refléter l'état modifié de la base de données après un l'opération est terminée. Lorsque le contexte de persistance étendue est ensuite associé à une transaction, il tentera de synchroniser son état actuel avec la base de données. Parce que les entités gérées dans le contexte de persistance sont désormais obsolètes par rapport au base de données, toute modification apportée depuis l'opération en bloc peut entraîner des résultats incorrects être stocké. Dans cette situation, la seule option est d'actualiser l'état de l'entité ou de s'assurer que les données sont versionnées de manière à ce que le changement incorrect puisse être détecté. Verrouillage les stratégies et l'actualisation de l'état de l'entité sont abordées au chapitre 12.

Suppression groupée et relations

Dans notre discussion sur l'opération remove () dans le chapitre précédent, nous avons souligné que la maintenance des relations est toujours sous la responsabilité du développeur. La seule fois une suppression en cascade se produit lorsque l'option REMOVE cascade est définie pour une relation. Même dans ce cas, le fournisseur de persistance ne mettra pas automatiquement à jour l'état de tout entités qui font référence à l'entité supprimée. Comme vous allez le voir, la même exigence est également vrai lors de l'utilisation d'instructions DELETE.

Une instruction DELETE en JP QL correspond plus ou moins à une instruction DELETE en SQL. Écrire la déclaration en JP QL vous donne l'avantage de travailler avec des entités au lieu de tables, mais la sémantique est exactement la même. Cela a des implications sur la façon dont les applications doivent écrire des instructions DELETE afin de s'assurer qu'elles s'exécutent correctement et laissez la base de données dans un état cohérent.

Les instructions DELETE sont appliquées à un ensemble d'entités dans la base de données, contrairement à remove (), qui s'applique à une seule entité dans le contexte de persistance. Une conséquence de ceci est que

Les instructions DELETE ne se répercutent pas sur les entités associées. Même si l'option en cascade REMOVE est fixé sur une relation, il ne sera pas suivi. Il est de votre responsabilité de vous assurer que les relations sont correctement mises à jour par rapport aux entités qui ont été supprimées.
Le fournisseur de persistance n'a également aucun contrôle sur les contraintes dans la base de données. Si vous tenter de supprimer les données qui sont la cible d'une relation de clé étrangère dans une autre table, vous pouvez obtenir une violation de contrainte d'intégrité référentielle en retour.

Épisode 321

Regardons un exemple qui met ces problèmes en contexte. Supposons, par exemple, que une entreprise souhaite réorganiser sa structure de service. Il veut supprimer un numéro des départements et ensuite affecter les employés à de nouveaux départements. La première étape consiste à supprimez les anciens départements, l'instruction suivante doit donc être exécutée:

```
SUPPRIMER DU Département d
WHERE d.name IN ('CA13', 'CA19', 'NY30')
```

C'est une opération simple. Nous voulons supprimer les entités du département qui correspondent à la liste de noms donnée en utilisant une instruction DELETE au lieu de rechercher le entités et en utilisant l'opération remove () pour les éliminer. Mais lorsque cette requête est exécuté, une exception PersistenceException est levée, signalant qu'une clé étrangère la contrainte d'intégrité a été violée. Une autre table a une référence de clé étrangère vers une des lignes que nous essayons de supprimer. En vérifiant la base de données, on voit que la table mappée par l'entité Employee a une contrainte de clé étrangère par rapport à la table mappée par le Entité départementale. Étant donné que la valeur de clé étrangère dans la table Employee n'est pas NULL, le La clé parent de la table Department ne peut pas être supprimée.

Nous devons d'abord mettre à jour les entités Employé en question pour nous assurer qu'elles ne indiquez le département que nous essayons de supprimer:

```
MISE À JOUR Employé e
SET e.department = null
WHERE e.department.name IN ('CA13', 'CA19', 'NY30')
```

Avec cette modification, l'instruction DELETE d'origine fonctionnera comme prévu. Considérez maintenant ce qui se serait passé si la contrainte d'intégrité n'avait pas été dans la base de données. L'opération DELETE se serait terminée avec succès, mais les valeurs de clé étrangère serait toujours assis à la table des employés. La prochaine fois que le fournisseur de persistance a essayé pour charger les entités Employee avec des clés étrangères pendantes, il serait impossible de résoudre l'entité cible. Le résultat de cette opération est spécifique au fournisseur, mais il peut conduire à un Une exception PersistenceException est levée, se plaignant de la relation non valide.

Astuces de requête

Les conseils de requête sont le point d'extension JPA des fonctionnalités de requête. Un indice est simplement une chaîne nom et valeur de l'objet. Les astuces permettent d'ajouter des fonctionnalités à JPA sans introduire une nouvelle API. Cela inclut des fonctionnalités standard telles que les délais d'expiration des requêtes illustrés

Épisode 322

plus tôt, ainsi que des fonctionnalités spécifiques au fournisseur. Notez que lorsqu'il n'est pas explicitement couvert par la Spécification JPA, aucune hypothèse ne peut être faite sur la portabilité des indices entre fournisseurs, même si les noms sont les mêmes. Chaque requête peut être associée à n'importe quel nombre d'indices, défini dans les métadonnées d'unité de persistance dans le cadre de @NamedQuery annotation, ou sur les interfaces Query ou TypedQuery à l'aide de la méthode setHint (). le l'ensemble actuel d'indices activés pour une requête peut être récupéré avec la méthode getHints (), qui renvoie une carte de paires nom et valeur.

Afin de simplifier la portabilité entre les fournisseurs, les fournisseurs de persistance sont doivent ignorer les indices qu'ils ne comprennent pas. Le listing 7-21 montre le Conseil "eclipselink.cache-usage" pris en charge par l'implémentation de référence JPA pour indiquer que le cache ne doit pas être vérifié lors de la lecture d'un employé à partir du base de données. Contrairement à la méthode refresh () de l'interface EntityManager, cette astuce ne fait que le résultat de la requête écrase la valeur actuelle mise en cache.

Liste 7-21. Utilisation des conseils de requête

```
public Employee findEmployeeNoCache (int empId) {
    TypedQuery <Employee> q = em.createQuery (
        "SELECT e FROM Employee e WHERE e.id =: empId", Employee.class);
    // force la lecture de la base de données
    q.setHint ("utilisation eclipselink.cache", "DoNotCheckCache");
    q.setParameter ("empId", empId);
    essayez {
        return q.getSingleResult ();
    } catch (NoResultException e) {
        return null;
    }
}
```

Si cette requête devait être exécutée fréquemment, une requête nommée serait plus efficace. La définition de requête nommée suivante intègre l'indicateur de cache utilisé précédemment:

```
@NamedQuery (nom = "findEmployeeNoCache",
                query = "SELECT e FROM Employee e WHERE e.id =: empId",
                hints = {@ QueryHint (nom = "eclipselink.cache-usage",
                                      value = "DoNotCheckCache")})
```


Épisode 323

L'élément hints accepte un tableau d'annotations @QueryHint, permettant à tout nombre d'indices à définir pour une requête. Cependant, une limitation de l'utilisation des annotations pour la requête nommée est que les indices sont limités à avoir des valeurs qui sont des chaînes, alors que lors de l'utilisation de la méthode Query.setHint (), tout type d'objet peut être passé comme indice valeur. Cela peut être particulièrement pertinent lorsque vous utilisez des conseils de fournisseurs propriétaires. Ça aussi représente un autre cas d'utilisation qui pourrait être ajouté à la liste dans le "Dynamic Named "Requêtes".

Meilleures pratiques relatives aux requêtes

L'application typique utilisant JPA aura de nombreuses requêtes définies. C'est la nature de applications d'entreprise dont les informations sont constamment interrogées à partir de la base de données pour tout, des rapports complexes aux listes déroulantes dans l'interface utilisateur. Donc, l'utilisation efficace des requêtes peut avoir un impact majeur sur l'ensemble de votre application

performance et réactivité. Lorsque vous effectuez les tests de performance de votre
requêtes, nous vous recommandons d'examiner certains des points de discussion ci-dessous
sections.

Requêtes nommées

Tout d'abord, nous recommandons les requêtes nommées dans la mesure du possible. Persistance
les fournisseurs prendront souvent des mesures pour précompiler les requêtes nommées JP QL en SQL dans le cadre de
la phase de déploiement ou d'initialisation d'une application. Cela évite les frais généraux
d'analyse continue de JP QL et de génération de SQL. Même avec un cache pour converti
requêtes, la définition de requête dynamique sera toujours moins efficace que l'utilisation de requêtes nommées.

Les requêtes nommées appliquent également les meilleures pratiques d'utilisation des paramètres de requête. Requete
les paramètres aident à conserver le nombre de chaînes SQL distinctes analysées par la base de données
au minimum. Parce que les bases de données conservent généralement un cache d'instructions SQL à portée de main
pour les requêtes fréquemment consultées, il s'agit d'un élément essentiel pour garantir la base de données de pointe
performance.

Comme nous l'avons vu dans la section "Définition de requête dynamique", les paramètres de requête
aident également à éviter les problèmes de sécurité causés par la concaténation de valeurs dans des chaînes de requête.
Pour les applications exposées au Web, la sécurité doit être une préoccupation à tous les niveaux d'un
application. Vous pouvez soit consacrer beaucoup d'efforts à la validation des paramètres d'entrée, soit
vous pouvez utiliser des paramètres de requête et laisser la base de données faire le travail à votre place.

Épisode 324

Lors de la dénomination des requêtes, choisissez une stratégie de dénomination au début de l'application
cycle de développement, étant entendu que l'espace de noms de requête est global pour chaque
unité de persistance. Les collisions entre les noms de requêtes sont susceptibles d'être une source commune
de frustration s'il n'y a pas de modèle de dénomination établi. Nous trouvons cela pratique et
recommande de préfixer le nom de la requête avec le nom de l'entité en cours
renvoyé, séparé par un point.

L'utilisation de requêtes nommées permet de remplacer les requêtes JP QL par des requêtes SQL
ou même avec des langages et des cadres d'expression spécifiques au fournisseur. Pour les applications
migrant à partir d'une solution de mappage relationnel objet existante, il est fort probable que le
le fournisseur fournira une assistance pour appeler sa solution de requête existante à l'aide de
fonction de requête nommée dans JPA. Nous abordons les requêtes nommées SQL au chapitre 11 .

La possibilité de créer dynamiquement des requêtes nommées dans le code peut être utile si l'un des
les cas que nous avons décrits précédemment s'appliquent à vous, ou si vous avez un autre cas d'utilisation
qui rend la création dynamique pertinente. En général, cependant, il est préférable et plus sûr si vous
peut déclarer toutes vos requêtes nommées de manière statique.

Signaler des requêtes

Si vous exécutez des requêtes qui renvoient des entités à des fins de reporting et
intention de modifier les résultats, envisagez d'exécuter des requêtes à l'aide d'une transaction-
gestionnaire d'entités de portée mais en dehors d'une transaction. Le fournisseur de persistance peut être
capable de détecter l'absence de transaction et d'optimiser les résultats de détachement, souvent en
ignorer certaines des étapes requises pour créer une version gérée provisoire de l'entité
résultats.

De même, si une entité est coûteuse à construire en raison de relations enthousiastes ou
un mappage de table complexe, pensez à sélectionner les propriétés des entités individuelles à l'aide d'un
requête de projection au lieu de récupérer le résultat complet de l'entité. Si tout ce dont vous avez besoin est le nom et
le numéro de téléphone du bureau pour 500 employés, sélectionner uniquement ces deux champs est probablement loin
plus efficace que la construction complète de 1 000 instances d'entité.

Conseils aux fournisseurs

Il est probable que les fournisseurs vous séduiront avec une variété d'indices pour activer différents
optimisations des performances pour les requêtes. Les conseils de requête peuvent bien être un outil essentiel
répondre à vos attentes de performance. Si la portabilité du code source à plusieurs fournisseurs

Épisode 325

est important, vous devez résister à l'envie d'intégrer des conseils de requête de fournisseur dans votre application
code. L'emplacement idéal pour les conseils de requête est dans un fichier de mappage XML (que nous décrivons
au chapitre 13) ou à tout le moins dans le cadre d'une définition de requête nommée. Les indices sont souvent
fortement dépendante de la plate-forme cible et pourrait devoir être modifiée au fil du temps car
différents aspects de l'application ont un impact sur l'équilibre global des performances. Garder
conseils découplés de votre code si possible.

Haricots apatrides

Nous avons essayé de démontrer de nombreux exemples dans le contexte d'un haricot apatride
car nous pensons que c'est la meilleure façon d'organiser les requêtes dans une application Java EE.
Utilisation de n'importe quel type de bean sans état, que ce soit un bean session sans état, une portée dépendante
Le bean CDI, ou un haricot de printemps à portée prototype, présente un certain nombre d'avantages par rapport au simple
intégrer des requêtes partout dans le code de l'application.

- Les clients peuvent exécuter des requêtes en invoquant un nom approprié
  méthode métier au lieu de s'appuyer sur un nom de requête cryptique ou
  plusieurs copies de la même chaîne de requête.
- Les méthodes Bean peuvent optimiser leur utilisation des transactions en fonction
  si les résultats doivent être gérés ou détachés.
- L'utilisation d'un contexte de persistance à l'échelle des transactions garantit que
  nombre d'instances d'entités ne restent pas gérées longtemps après
  sont nécessaires.

Cela ne veut pas dire que les autres composants sont des emplacements inappropriés pour émettre des requêtes,
mais les beans stateless sont une bonne pratique bien établie pour l'hébergement de requêtes dans Java EE
environnement.

Mise à jour et suppression en masse

Si des opérations de mise à jour et de suppression en bloc doivent être utilisées, assurez-vous qu'elles ne sont exécutées que dans
une transaction isolée où aucun autre changement n'est effectué. Il existe de nombreuses façons
dont ces requêtes peuvent avoir un impact négatif sur un contexte de persistance actif. Entrelacement
ces requêtes avec d'autres opérations non groupées nécessitent une gestion minutieuse par le
application.

Épisode 326

La gestion des versions et le verrouillage des entités nécessitent une attention particulière lors de la mise à jour en bloc
les opérations sont utilisées. Les opérations de suppression en bloc peuvent avoir de vastes ramifications

selon la façon dont le fournisseur de persistance peut réagir et ajuster la mise en cache des entités
en réponse. Par conséquent, nous considérons les opérations de mise à jour et de suppression en masse comme étant hautement
spécialisé, à utiliser avec précaution.

Différences de fournisseur

Prenez le temps de vous familiariser avec le SQL généré par votre fournisseur de persistance
pour différentes requêtes JP QL. Bien que la compréhension de SQL ne soit pas nécessaire pour écrire
Requêtes JP QL, savoir ce qui se passe en réponse aux différentes opérations JP QL est
une partie essentielle du réglage des performances. Les jointures dans JP QL ne sont pas toujours explicites, et vous
peut être surpris par le SQL complexe généré pour un JP QL apparemment simple
requete.

Les avantages des fonctionnalités telles que la pagination des requêtes dépendent également de l'approche
utilisé par votre fournisseur de persistance. Il existe un certain nombre de techniques différentes qui
peut être utilisé pour réaliser la pagination, dont beaucoup souffrent de performances et
problèmes d'évolutivité. Parce que JPA ne peut pas dicter une approche particulière qui fonctionnera bien dans
tous les cas, familiarisez-vous avec l'approche utilisée par votre fournisseur et si
est configurable.

Enfin, comprendre la stratégie du fournisseur pour savoir quand et à quelle fréquence
le contexte de persistance est nécessaire avant d'examiner les optimisations telles que la modification du
mode de rinçage. En fonction de l'architecture de mise en cache et des optimisations de requête utilisées par un
fournisseur, la modification du mode de vidage peut ou non faire une différence pour votre application.

Résumé

Nous avons commencé ce chapitre par une introduction à JP QL, le langage de requête défini par
JPA. Nous avons brièvement discuté des origines de JP QL et de son rôle dans l'écriture de requêtes qui interagissent
avec des entités. Nous avons également fourni un aperçu des principales fonctionnalités de JP QL pour les développeurs
déjà expérimenté avec SQL ou EJB QL.

Dans la discussion sur l'exécution des requêtes, nous avons présenté les méthodes pour définir
interroge à la fois de manière dynamique lors de l'exécution et de manière statique dans le cadre des métadonnées d'unité de persistance.
Nous avons examiné les interfaces Query et TypedQuery et les types de résultats de requête possibles

Épisode 327

en utilisant JP QL. Nous avons également examiné la liaison de paramètres, les stratégies de gestion de gros résultats
ensembles, et comment s'assurer que les requêtes dans les transactions avec des données modifiées sont complètes
avec succès.

Dans la section sur la mise à jour et la suppression en bloc, nous avons expliqué comment exécuter ces types
des requêtes et comment s'assurer qu'elles sont utilisées en toute sécurité par l'application. Nous avons fourni
des détails sur la manière dont les fournisseurs de persistance traitent les opérations en masse et sur leur impact
ont sur le contexte de persistance active.

Nous avons terminé notre discussion sur les fonctionnalités de requête en examinant les indices de requête. Nous avons montré
comment spécifier des astuces et fournir un exemple à l'aide d'indices pris en charge par JPA
Implémentation de référence.

Enfin, nous avons résumé notre vision des meilleures pratiques relatives aux requêtes, en
aux requêtes nommées, différentes stratégies pour les différents types de requêtes, ainsi que
les détails d'implémentation qui doivent être compris pour différents fournisseurs de persistance.

Dans le chapitre suivant, nous continuons à nous concentrer sur les requêtes en examinant JP QL en détail.

Épisode 328

CHAPITRE 8

Langage de requête

L'API Java Persistence fournit deux méthodes d'interrogation des entités: le Java
Langage de requête de persistance (JPQL) et l'API Criteria.

Le langage de requête de persistance Java (JP QL) est la requête standard basée sur une chaîne
langue de JPA. C'est un langage de requête portable conçu pour combiner la syntaxe et
sémantique de requête simple de SQL avec l'expressivité d'une expression orientée objet
Langue. Les requêtes écrites en utilisant ce langage peuvent être compilées de manière portable en SQL sur tous
principaux serveurs de bases de données.

L'API Criteria est utilisée pour créer des requêtes de type sécurisé à l'aide de la programmation Java
API de langage lors de la requête d'entités et de leurs relations.

Dans le dernier chapitre, nous avons examiné la programmation en utilisant les interfaces de requête et
a présenté une brève introduction à JP QL pour les utilisateurs déjà expérimentés avec SQL. Cette
Le chapitre explore le langage de requête en détail, décomposant le langage morceau par
pièce avec des exemples pour démontrer ses caractéristiques.

Présentation de JP QL

Afin de décrire ce qu'est JP QL, il est important de préciser ce qu'il n'est pas. JP QL
n'est pas SQL. Malgré les similitudes entre les deux langues en termes de mots-clés
et la structure générale, il existe des différences très importantes. Tentative d'écriture de JP QL
comme si c'était SQL est le moyen le plus simple d'être frustré par la langue. Les similitudes
entre les deux langages sont intentionnels (donnant aux développeurs une idée de ce que JP QL peut
accomplir), mais la nature orientée objet de JP QL nécessite un autre type de réflexion.

Si JP QL n'est pas SQL, qu'est-ce que c'est? En termes simples, JP QL est un langage pour interroger des entités.
Au lieu de tableaux et de lignes, la devise du langage est constituée d'entités et d'objets. Il
nous fournit un moyen d'exprimer les requêtes en termes d'entités et de leurs relations,
fonctionnant sur l'état persistant de l'entité tel que défini dans le modèle objet, et non dans le

modèle de base de données physique.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_8

Épisode 329

Si JPA prend en charge les requêtes SQL, pourquoi introduire un nouveau langage de requête? Il y a un quelques raisons importantes de considérer JP QL sur SQL. Le premier est la portabilité. JP QL peut être traduit dans les dialectes SQL de tous les principaux fournisseurs de bases de données. La seconde est que les requêtes sont écrites par rapport au modèle de domaine des entités persistantes, sans qu'il soit nécessaire savoir exactement comment ces entités sont mappées à la base de données.

Comparons un peu les API JPQL et Criteria.

Les requêtes JPQL sont généralement plus concises et lisibles que les requêtes Critères. JPQL est facile à apprendre pour les programmeurs ayant des connaissances préalables en SQL.

Les requêtes JPQL ne sont pas de type sécurisé, ce qui signifie qu'elles nécessitent un cast lors de la récupération le résultat de la requête de Entity Manager. Pour cette raison, les erreurs de conversion de type peuvent ne pas être pris au moment de la compilation.

De plus, les requêtes JPQL ne prennent pas en charge les paramètres ouverts.

Les requêtes de l'API Criteria sont sécurisées et ne nécessitent donc pas de conversion.

Lors de la comparaison des performances entre les API JPQL et Criteria, l'API Criteria les requêtes offrent de meilleures performances car les requêtes dynamiques JPQL doivent être analysées à chaque moment où ils sont appelés.

L'un des inconvénients courants des requêtes de l'API Criteria est qu'elles sont généralement plus verbeux que les requêtes JPQL. Cela signifie qu'ils auront besoin des programmeurs pour créer de nombreux objets et effectuer des opérations sur ces objets avant de soumettre le Requête de l'API de critères à Entity Manager.

Les exemples de ce chapitre démontrent la puissance présente même dans le plus simple des Expressions JP QL.

Adopter JP QL ne signifie pas perdre toutes les fonctionnalités SQL que vous avez développées habitué à utiliser. Une large sélection de fonctionnalités SQL sont directement prises en charge, notamment sous-requêtes, requêtes agrégées, instructions UPDATE et DELETE, nombreuses fonctions SQL, et plus.

Terminologie

Les requêtes appartiennent à l'une des quatre catégories suivantes: sélectionner, agréger, mettre à jour et supprimer. Sélectionner Les requêtes récupèrent l'état persistant d'une ou plusieurs entités, filtrant les résultats selon les besoins. Les requêtes agrégées sont des variantes de requêtes sélectionnées qui regroupent les résultats et produisent données récapitulatives. Ensemble, les requêtes de sélection et d'agrégation sont parfois appelées rapport requêtes, car elles sont principalement axées sur la génération de données pour le reporting. Mettre à jour et

Épisode 330

les requêtes de suppression sont utilisées pour modifier ou supprimer de manière conditionnelle des ensembles entiers d'entités. Vous serez trouvez chaque type de requête décrit en détail dans sa propre section de ce chapitre.

Les requêtes opèrent sur l'ensemble des entités et des éléments incorporables définis par une unité de persistance. Cet ensemble d'entités et d'éléments incorporables est connu sous le nom de schéma de persistance abstraite, le dont la collection définit le domaine global à partir duquel les résultats peuvent être récupérés.

Remarque pour permettre à ce chapitre d'être utilisé comme compagnon du langage de requête chapitre de la spécification Java persistence apI, la même terminologie est utilisée lorsque c'est possible.

Dans les expressions de requête, les entités sont désignées par leur nom. Si une entité n'a pas été explicitement nommé (à l'aide de l'attribut name de l'annotation @Entity, par exemple), le un nom de classe non qualifié est utilisé par défaut. Ce nom est le nom de schéma abstrait du entité dans le contexte d'une requête.

Les entités sont composées d'une ou plusieurs propriétés de persistance implémentées sous forme de champs ou les propriétés JavaBean. Le type de schéma abstrait d'une propriété persistante sur une entité fait référence à la classe ou au type primitif utilisé pour implémenter cette propriété. Par exemple, si le L'entité Employee a un nom de propriété de type String, le type de schéma abstrait de cette La propriété dans les expressions de requête est également String. Propriétés persistantes simples sans le mappage de relation comprend l'état persistant de l'entité et est appelé champs d'état. Les propriétés persistantes qui sont également des relations sont appelées champs d'association.

Comme vous l'avez vu dans le dernier chapitre, les requêtes peuvent être définies de manière dynamique ou statique. Les exemples de ce chapitre sont constitués de requêtes qui peuvent être utilisées de manière dynamique ou statiquement, en fonction des besoins de l'application.

Enfin, il est important de noter que les requêtes ne sont pas sensibles à la casse sauf dans deux cas: les noms d'entités et les noms de propriétés doivent être spécifiés exactement comme ils sont nommés

Exemple de modèle de données

Figure 8-1 montre le modèle de domaine pour les requêtes de ce chapitre. Poursuivre la exemples que nous avons utilisés tout au long du livre, il montre de nombreux types de relations, y compris unidirectionnel, bidirectionnel et auto-référencé des relations. Nous avons ajouté les noms de rôle à ce diagramme pour créer la relation noms de propriété explicites.

Épisode 331

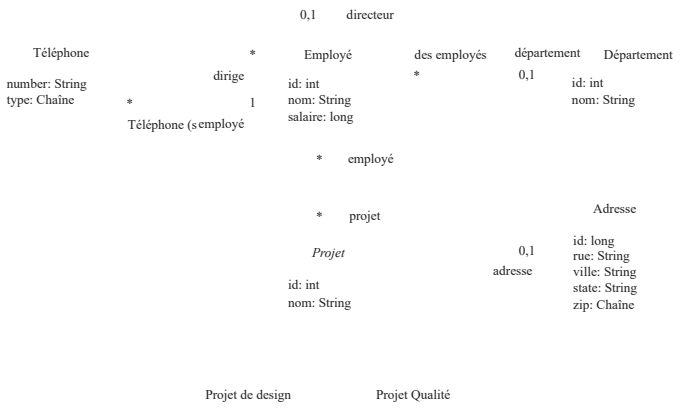

Figure 8-1. *Exemple de modèle de domaine d'application*

Les mappages relationnels d'objet pour ce modèle ne sont pas inclus dans ce chapitre sauf où nous décrivons l'équivalent SQL d'une requête particulière. Il n'est pas nécessaire de savoir comment un objet est mappé afin d'écrire des requêtes car le langage de requête est basé entièrement sur le modèle objet et les relations logiques entre les entités. C'est le boulot du traducteur de requêtes pour prendre les expressions de requête orientées objet et interpréter le mappage des métadonnées afin de produire le SQL requis pour exécuter la requête sur le

base de données.

Exemple d'application

Apprendre une nouvelle langue peut être une expérience stimulante. C'est une chose à lire page après page de texte décrivant les caractéristiques de la langue, mais c'est autre chose complètement pour mettre ces fonctionnalités en pratique. Pour vous habituer à écrire des requêtes, pensez à en utilisant une application comme celle illustrée dans le Listing 8-1 . Cette application simple lit requêtes de la console et les exécute contre les entités d'un particulier unité de persistance.

Épisode 332

Liste 8-1. Demande de test de requêtes

```
persistance du package;

import java.io. *;
import java.util. *;
import javax.persistence. *;
import org.apache.commons.lang.builder. *;

public class QueryTester {

    public static void main (String [] args) lève l'exception {
        Chaîne unitName = args [0];

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory (unitName);
        EntityManager em = emf.createEntityManager ();
        Lecteur BufferedReader =
            nouveau BufferedReader (nouveau InputStreamReader (System.in));

        pour (;;) {
            System.out.print ("JP QL>");
            Requête de chaîne = reader.readLine ();
            if (query.equals ("quitter")) {
                Pause;
            }
            if (query.length () == 0) {
                continuer;
            }

            essayez {
                Résultat de la liste = em.createQuery (requête) .getResultList ();
                if (result.size ()> 0) {
                    nombre int = 0;
                    for (Objet o: résultat) {
                        System.out.print (++ count + "");
                        printResult (o);
                    }
```

Épisode 333


```
                    } autre {
                        System.out.println ("0 résultat renvoyé");
                    }
                } catch (Exception e) {
                    e.printStackTrace ();
                }
            }
        }

        private static void printResult (Object result) jette une exception {
            if (résultat == null) {
                System.out.print ("NULL");
            } else if (résultat instanceof Object []) {
                Object [] row = (Object []) result;
                System.out.print ("[");
                pour (int i = 0; i <row.length; i ++) {
                    printResult (ligne [i]);
                }
                System.out.print ("]");
            } else if (résultat instanceof Long ||
                        résultat instanceof Double ||
                        résultat instanceof String) {
                System.out.print (result.getClass (). GetName () + ":" + result);
            } autre {
                System.out.print (ReflectionToStringBuilder.toString (résultat,
                                                    ToStringStyle.SHORT_PREFIX_STYLE));
            }
            System.out.println ();
        }
}
```

La seule condition requise pour utiliser cette application est le nom d'une persistance unité contenant les entités sur lesquelles vous souhaitez interroger. L'application lira le nom de l'unité de persistance à partir de la ligne de commande et essayez de créer un EntityManagerFactory pour ce nom. Si l'initialisation réussit, les requêtes peuvent être tapé à l'invite JP QL>. La requête sera exécutée et les résultats imprimés. le

Épisode 334

le format de chaque résultat est le nom de la classe suivi de chacune des propriétés de cette classe. Cet exemple utilise la bibliothèque Apache Jakarta Commons-Lang pour générer l'objet résumé. Le listing 8-2 montre un exemple de session avec l'application.

Liste 8-2. Exemple de session avec QueryTester

```
JP QL> SELECT p FROM Phone p WHERE p.type NOT IN ('bureau', 'domicile')
1 téléphone [id = 5, numéro = 516-555-1234, type = cellule, employé = employé @ 13c0b53]
2 Téléphone [id = 9, numéro = 650-555-1234, type = cellule, employé = employé @ 193f6e2]
3 Téléphone [id = 12, nombre = 650-555-1234, type = cellule, employé = employé @ 36527f]
```

4 Téléphone [id = 18, nombre = 585-555-1234, type = cellule, employé = employé @ bd6a5f]
5 Téléphone [id = 21, nombre = 650-555-1234, type = cellule, employé = employé @ 979e8b]
JP QL> SELECT d.name, AVG (e.salary) FROM Department d JOIN d.employees e ➥
GROUP BY d.nom
1 [java.lang.String: QA
java.lang.Double: 52500.0
]
2 [java.lang.String: Ingénierie
java.lang.Double: 56833.333333333336
]
JP QL> quitter

Sélectionner des requêtes

Les requêtes Select sont le type de requête le plus important et facilitent la récupération groupée de données de la base de données. Sans surprise, les requêtes de sélection sont également la forme la plus courante de requête utilisée dans les applications. La forme générale d'une requête de sélection est la suivante:

SELECT <expression_sélectionner>
FROM <clause_from>
[WHERE <expression_conditionnelle>]
[ORDER BY <order_by_clause>]

La forme la plus simple d'une requête de sélection se compose de deux parties obligatoires: le SELECT clause et la clause FROM. La clause SELECT définit le format des résultats de la requête, tandis que la clause FROM définit l'entité ou les entités dont les résultats seront

Épisode 335

obtenu. Considérez la requête complète suivante qui récupère tous les employés du compagnie:

SÉLECTIONNER e
DE Employé e

La structure de cette requête est très similaire à une requête SQL, mais avec quelques différences importantes. La première différence est que le domaine de la requête défini dans le La clause FROM n'est pas une table mais une entité; dans ce cas, l'entité Employé. Comme dans SQL, il a a été aliasé sur l'identifiant e. Cette valeur aliasée est connue sous le nom de variable d'identification et est la clé par laquelle l'entité sera référencée dans le reste de l'instruction select. Contrairement aux requêtes en SQL, où un alias de table est facultatif, l'utilisation de variables d'identification est obligatoire dans JP QL.

La deuxième différence est que la clause SELECT de cet exemple n'énumère pas les champs de la table ou utilisez un caractère générique pour sélectionner tous les champs. Au lieu de cela, seul le la variable d'identification est répertoriée afin d'indiquer que le type de résultat de la requête est le Entité d'employé, pas un ensemble tabulaire de lignes.

Lorsque le processeur de requêtes parcourt le jeu de résultats renvoyé par la base de données, il convertit les données tabulaires de ligne et de colonne en un ensemble d'instances d'entité. le La méthode getResultList () de l'interface Query retournera une collection de zéro ou plus Objets employés après avoir évalué la requête. Malgré les différences de structure et syntaxe, chaque requête est traduisible en SQL.

Afin d'exécuter une requête, le moteur de requête construit d'abord un SQL optimal représentation de la requête JP QL. La requête SQL résultante est ce qui est réellement exécuté sur la base de données. Dans cet exemple simple, le SQL pourrait ressembler à ceci, en fonction des métadonnées de mappage pour l'entité Employé:

SELECT id, nom, salaire, manager_id, dept_id, address_id
DE EMP

L'instruction SQL doit lire toutes les colonnes mappées requises pour créer le
instance d'entité, y compris les colonnes de clé étrangère. Même si l'entité est mise en cache en mémoire,
le moteur de requête lira toujours toutes les données requises pour garantir que le
la version est à jour. Notez que si les relations entre l'employé et le
Les entités Department ou Address nécessitaient un chargement hâtif, l'instruction SQL
soit être étendu pour récupérer les données supplémentaires, soit plusieurs instructions auraient été
regroupés afin de construire complètement l'entité Employé. Chaque vendeur va

Épisode 336

fournir une méthode pour afficher le SQL qu'il génère à partir de la traduction de JP QL. Pour
réglage des performances en particulier, comprendre comment votre fournisseur aborde SQL
generation peut vous aider à rédiger des requêtes plus efficaces.

Maintenant que nous avons examiné une requête simple et couvert la terminologie de base, le
les sections suivantes se déplacent à travers chacune des clauses de la requête de sélection, expliquant
la syntaxe et les fonctionnalités disponibles.

Notez comme nous le montrons dans le chapitre 2 , la nouvelle fonctionnalité de streaming de requêtes incluse dans Jpa 2.2 nous aidera à éviter de récupérer trop de données et de provoquer des erreurs. cependant, il est toujours recommandé et plus efficace d'utiliser la pagination de l'ensemble de résultats méthode.

Clause SELECT

La clause SELECT d'une requête peut prendre plusieurs formes, y compris simples et complexes
expressions de chemin, expressions scalaires, expressions de constructeur, fonctions d'agrégation et
séquences de ces types d'expression. Les sections suivantes présentent les expressions de chemin
et discuter des différents styles de clauses SELECT et comment ils déterminent le résultat
type de requête. Nous reportons la discussion des expressions scalaires jusqu'à l'exploration du conditionnel
expressions dans la clause WHERE. Ils sont décrits en détail dans la section intitulée «Scalaire
Expressions." Les fonctions d'agrégation sont détaillées plus loin dans le chapitre de la section intitulée
«Requêtes agrégées».

Expressions de chemin

Les expressions de chemin sont les éléments constitutifs des requêtes. Ils sont utilisés pour sortir
d'une entité, soit à travers une relation avec une autre entité (ou un ensemble d'entités)
ou à l'une des propriétés persistantes d'une entité. Navigation qui aboutit à l'un des
les champs d'état persistants (champ ou propriété) d'une entité sont appelés champ d'état
chemin. La navigation qui mène à une seule entité est appelée association à valeur unique
chemin, alors que la navigation vers une collection d'entités est appelée une valeur de collection
chemin d'association.

L'opérateur point (.) Signifie la navigation de chemin dans une expression. Par exemple, si
l'entité Employee a été mappée à la variable d'identification e, e.name est un état
expression de chemin de champ résolvant le nom de l'employé. De même, l'expression de chemin

Épisode 337

e.department est une association à valeur unique allant de l'employé au ministère pour
auquel il ou elle est assigné. Enfin, e.directs est une association à valeur de collection qui
décide de la collecte des employés relevant d'un employé qui est également un gestionnaire.

Ce qui rend les expressions de chemin si puissantes, c'est qu'elles ne sont pas limitées à un seul
la navigation. Au lieu de cela, les expressions de navigation peuvent être enchaînées pour parcourir
graphiques d'entités complexes tant que le chemin se déplace de gauche à droite sur une valeur unique
les associations. Un chemin ne peut pas continuer à partir d'un champ d'état ou d'une association à valeur de collection.
En utilisant cette technique, nous pouvons construire des expressions de chemin telles que e.department.name,
qui est le nom du département auquel appartient l'employé. Notez ce chemin
Les expressions peuvent naviguer dans et entre les objets incorporés ainsi que les entités normales.
La seule restriction sur les objets incorporés dans une expression de chemin est que la racine du chemin
l'expression doit commencer par une entité.

Les expressions de chemin sont utilisées dans chaque clause d'une requête de sélection, déterminant tout
du type de résultat de la requête aux conditions dans lesquelles les résultats doivent être
filtré. L'expérience des expressions de chemin est la clé pour rédiger des requêtes efficaces.

Entités et objets

La première et la plus simple forme de la clause SELECT est une variable d'identification unique. le
le type de résultat pour une requête de ce style est l'entité à laquelle la variable d'identification
est associé. Par exemple, la requête suivante renvoie tous les départements du
compagnie:

```
SELECT d
DU Département d
```

Le mot-clé OBJECT peut être utilisé pour indiquer que le type de résultat de la requête est le
entité liée à la variable d'identification. Cela n'a aucun impact sur la requête, mais cela peut être
utilisé comme indice visuel.

```
CHOISIR UN OBJET (d)
DU Département d
```

Le seul problème avec l'utilisation de OBJECT est que même si les expressions de chemin peuvent résoudre
à un type d'entité, la syntaxe du mot clé OBJECT est limitée aux variables d'identification.
L'expression OBJECT (e.department) est illégale même si Department est une entité
type. Pour cette raison, nous ne recommandons pas la syntaxe OBJECT. Il existe principalement pour

Épisode 338

compatibilité avec les versions précédentes du langage nécessitant le mot clé OBJECT
dans l'hypothèse qu'une future révision de SQL inclurait la même terminologie.

Une expression de chemin résolue en un champ d'état ou une association à valeur unique peut également être
utilisé dans la clause SELECT. Le type de résultat de la requête dans ce cas devient le type
de l'expression de chemin, soit le type de champ d'état, soit le type d'entité d'un
association. La requête suivante renvoie les noms de tous les employés:

```
SELECT e.name
DE Employé e
```

Le type de résultat de l'expression de chemin dans la clause SELECT est String, donc l'exécution
cette requête utilisant getResultList () produira une collection de zéro ou plus String
objets. Les expressions de chemin résolues dans les champs d'état peuvent également être utilisées dans le cadre du scalaire
expressions, permettant au champ d'état d'être transformé dans les résultats de la requête. Nous discutons

cette technique plus loin dans la section intitulée «Expressions scalaires».

Les entités atteintes à partir d'une expression de chemin peuvent également être renvoyées. La requête suivante illustre le renvoi d'une entité différente à la suite de la navigation dans le chemin:

```
SELECT e.department
DE Employé e
```

Le type de résultat de cette requête est l'entité Department, car c'est le résultat de traversant la relation de service de l'employé au service. Exécuter le requête aboutira donc à une collection de zéro ou plusieurs objets Department, y compris doublons.

Pour supprimer les doublons, l'opérateur DISTINCT doit être utilisé:

```
SELECT DISTINCT e.department
DE Employé e
```

L'opérateur DISTINCT est fonctionnellement équivalent à l'opérateur SQL du même Nom. Une fois l'ensemble de résultats collecté, dupliquez les valeurs (en utilisant l'identité de l'entité si la requête type de résultat est une entité) sont supprimés afin que seuls les résultats uniques soient renvoyés.

Le type de résultat d'une requête de sélection est le type correspondant à chaque ligne du résultat ensemble produit en exécutant la requête. Cela peut inclure des entités, des types primitifs et d'autres types d'attributs persistants, mais jamais un type de collection. La requête suivante est illégale:

```
SELECT d. Employés
DU Département d
```


Épisode 339

L'expression de chemin d.employees est un chemin de valeur de collection qui produit un type de collection. Restreindre les requêtes de cette manière évite au fournisseur d'avoir à combiner des lignes successives de la base de données en un seul objet de résultat.

Il est possible de sélectionner des objets intégrables dans une expression de chemin. le La requête suivante renvoie uniquement les objets intégrables ContactInfo pour tous les employés:

```
SELECT e.contactInfo
DE Employé e
```

La chose à retenir lors de la sélection des éléments incorporables est que les objets renvoyés pas être géré. Si vous émettez une requête pour renvoyer les employés (sélectionnez e FROM Employé e) puis, à partir des résultats, accédez à leurs objets intégrés ContactInfo, vous obtenir des éléments intégrés qui ont été gérés. Modifications apportées à l'un de ces objets serait sauvegardé lorsque la transaction est validée. Modification de l'un des ContactInfo les résultats d'objet renvoyés par une requête qui a sélectionné directement ContactInfo, cependant, n'aurait aucun effet persistant.

Combinaison d'expressions

Plusieurs expressions peuvent être spécifiées dans la même clause SELECT en les séparant par des virgules. Le type de résultat de la requête dans ce cas est un tableau de type Object, où le Les éléments du tableau sont les résultats de la résolution des expressions dans l'ordre dans lequel ils sont apparus dans la requête.

Considérez la requête suivante qui renvoie uniquement le nom et le salaire d'un employé:

```
SELECT e.name, e.salary
DE Employé e
```

Quand ceci est exécuté, une collection de zéro ou plusieurs instances de tableaux de type Object sera retourné. Chaque tableau de cet exemple comporte deux éléments, le premier étant une chaîne contenant le nom de l'employé et le second étant un Double contenant l'employé un salaire. La pratique consistant à signaler uniquement un sous-ensemble des champs d'état d'une entité est appelée

projection car les données d'entité sont projetées hors de l'entité sous forme de tableau.

La projection est une technique utile pour les applications Web dans lesquelles seules quelques pièces des informations sont affichées à partir d'un grand nombre d'instances d'entités. Selon comment l'entité a été mappée, il peut nécessiter une requête SQL complexe pour récupérer complètement le état de l'entité. Si seulement deux champs sont requis, l'effort supplémentaire consacré à la construction de l'entité instance aurait pu être gaspillée. Une requête de projection qui ne renvoie que le minimum la quantité de données est plus utile dans ces cas.

Épisode 340

Expressions de constructeur

Une forme plus puissante de la clause SELECT impliquant plusieurs expressions est la expression de constructeur, qui spécifie que les résultats de la requête doivent être stockés en utilisant un type d'objet spécifié par l'utilisateur. Considérez la requête suivante:

```
SELECT NEW example.EmployeeDetails (e.name, e.salary, e.department.name)
DE Employé e
```

Le type de résultat de cette requête est la classe Java example.EmployeeDetails. Comme le le processeur de requêtes itère sur les résultats de la requête, il instancie de nouvelles instances de EmployeeDétails à l'aide du constructeur qui correspond aux types d'expression répertoriés dans le requete. Dans ce cas, les types d'expression sont String, Double et String, donc la requête engine recherchera un constructeur avec ces types de classe pour les arguments. Chaque ligne du la collection de requêtes résultante est donc une instance de EmployeeDetails contenant le nom de l'employé, salaire et nom du service.

Le type d'objet de résultat doit être référencé à l'aide du nom qualifié complet du objet. Cependant, la classe ne doit en aucun cas être mappée à la base de données. Tout classe avec un constructeur compatible avec les expressions répertoriées dans la clause SELECT peut être utilisé dans une expression de constructeur.

Les expressions de constructeur sont des outils puissants pour la construction de données grossières transférer des objets ou afficher des objets à utiliser dans d'autres niveaux d'application. Au lieu de manuellement construction de ces objets, une seule requête peut être utilisée pour rassembler des objets de vue prêt pour une présentation sur une page Web.

Clause FROM

La clause FROM est utilisée pour déclarer une ou plusieurs variables d'identification, éventuellement dérivés de relations jointes, qui forment le domaine sur lequel la requête doit tirer ses résultats. La syntaxe de la clause FROM consiste en une ou plusieurs identifications variables et déclarations de clause de jointure.

Variables d'identification

La variable d'identification est le point de départ de toutes les expressions de requête. Chaque requête doit avoir au moins une variable d'identification définie dans la clause FROM, et que La variable doit correspondre à un type d'entité. Lorsqu'une déclaration de variable d'identification

Épisode 341

n'utilise pas d'expression de chemin (c'est-à-dire, lorsqu'il s'agit d'un nom d'entité unique), il est appelé une déclaration de variable de plage. Cette terminologie vient de la théorie des ensembles car la variable est dit de s'étendre sur l'entité.

Les déclarations de variable de plage utilisent la syntaxe <nom_entité> [AS] <identificateur>. Nous avons utilisé cette syntaxe dans tous nos exemples précédents, mais sans l'option Mot clé AS. L'identifiant doit suivre les règles de dénomination Java standard et peut être référencé tout au long de la requête sans tenir compte de la casse. Plusieurs déclarations peuvent être spécifié en les séparant par des virgules.

Les expressions de chemin peuvent également être associées à des variables d'identification dans le cas de jointures et sous-requêtes. La syntaxe des déclarations de variables d'identification dans ces cas sera couvert dans les deux sections suivantes.

Rejoint

Une jointure est une requête qui combine les résultats de plusieurs entités. Les jointures dans les requêtes JP QL sont équivalent logiquement aux jointures SQL. En fin de compte, une fois la requête traduite en SQL, elle est très probable que les jointures entre entités produiront des jointures similaires entre les tables pour dont les entités sont mappées. Il est donc important de comprendre quand les jointures se produisent à l'écriture de requêtes efficaces.

Les jointures se produisent chaque fois que l'une des conditions suivantes est remplie dans une requête SELECT.

- Deux déclarations de variable de plage ou plus sont répertoriées dans la clause FROM et apparaissent dans la clause SELECT.
- L'opérateur JOIN est utilisé pour étendre une variable d'identification à l'aide d'un expression de chemin.
- Une expression de chemin n'importe où dans la requête navigue dans un champ d'association, à la même entité ou à une entité différente.
- Une ou plusieurs conditions WHERE comparent les attributs de différents variables d'identification.

La sémantique d'une jointure entre entités est la même que celle des jointures SQL entre tables. La plupart des requêtes contiennent une série de conditions de jointure, qui sont des expressions qui définissent le règles pour faire correspondre une entité à une autre. Les conditions de jointure peuvent être spécifiées explicitement, comme comme utilisation de l'opérateur JOIN dans la clause FROM d'une requête, ou implicitement comme résultat de path la navigation.

Épisode 342

Une jointure interne entre deux entités renvoie les objets des deux types d'entités qui satisfaire toutes les conditions de jointure. La navigation de chemin d'une entité à une autre est une forme de jointure interne. La jointure externe de deux entités est l'ensemble des objets des deux types d'entités qui satisfont aux conditions de jointure ainsi qu'à l'ensemble d'objets d'un type d'entité (désigné comme entité de gauche) qui n'ont pas de condition de jointure correspondante dans l'autre.

En l'absence de conditions de jointure entre deux entités, les requêtes produiront un Produit cartésien. Chaque objet du premier type d'entité est associé à chaque objet du deuxième type d'entité, en mettant au carré le nombre de résultats [1] . Les produits cartésiens sont rares avec Requêtes JP QL compte tenu des capacités de navigation du langage, mais elles sont possibles si deux déclarations de variable de plage dans la clause FROM sont spécifiées sans conditions spécifiées dans la clause WHERE.

Une discussion plus approfondie et des exemples de chaque style de jointure sont fournis ci-dessous sections.

Jointures intérieures

Jusqu'à présent, tous les exemples de requêtes utilisaient la forme la plus simple de la clause FROM, une

type d'entité unique associé à une variable d'identification. Cependant, en tant que langage relationnel, JP QL prend en charge les requêtes qui s'appuient sur plusieurs entités et les relations entre leur.

Les jointures internes entre deux entités peuvent être spécifiées de l'une des manières répertoriées précédemment. La première forme préférée, car il est explicite et évident qu'une jointure est se produit, est l'opérateur JOIN dans la clause FROM. Une autre forme nécessite plusieurs plages déclarations de variable dans la clause FROM et la clause WHERE conditions pour fournir la jointure conditions.

Champs d'association d'opérateur et de collection JOIN

La syntaxe d'une jointure interne à l'aide de l'opérateur JOIN est [INNER] JOIN <path_expression> [AS] <identifiant>. Considérez la requête suivante:

```
SÉLECTIONNER p
FROM Employé e JOIN e.phones p
```

[1] Le nombre exact de résultats sera M * N, où M est le nombre d'instances d'entité du premier type et N est le nombre d'instances d'entité du second type.

Épisode 343

Cette requête utilise l'opérateur JOIN pour joindre l'entité Employee à l'entité Phone à travers la relation téléphonique. La condition de jointure dans cette requête est définie par l'objet-cartographie relationnelle de la relation téléphonique. Aucun critère supplémentaire ne doit être spécifié afin de lier les deux entités. En joignant les deux entités ensemble, cette requête renvoie tous les instances d'entité Téléphone associées aux employés de l'entreprise.

La syntaxe des jointures est similaire aux expressions JOIN prises en charge par ANSI SQL. Pour les lecteurs qui ne sont peut-être pas familiers avec cette syntaxe, envisagent la forme SQL équivalente de la requête précédente écrite à l'aide du formulaire de jointure traditionnel:

```
SELECT p.id, p.phone_num, p.type, p.emp_id
DE emp e, téléphone p
WHERE e.id = p.emp_id
```

Le mappage de table pour l'entité Phone remplace l'expression e.phones. Le O La clause inclut également les critères nécessaires pour joindre les deux tables ensemble à travers la jointure colonnes définies par le mappage des téléphones.

Notez que la relation téléphones a été mappée à la variable d'identification p. Même si l'entité Phone n'apparaît pas directement dans la requête, la cible du La relation téléphones est l'entité Téléphone, et cela détermine la variable d'identification type. Cette détermination implicite du type de variable d'identification peut prendre un certain temps habitué. Une connaissance de la manière dont les relations sont définies dans le modèle objet est nécessaire pour naviguer dans une requête écrite.

Chaque occurrence de p en dehors de la clause FROM fait désormais référence à un seul téléphone appartenant à un employé. Même si un champ d'association de collection a été spécifié dans le JOIN, la variable d'identification fait réellement référence aux entités atteintes par association, pas la collection elle-même. La variable peut maintenant être utilisée comme si l'entité Phone étaient répertoriés directement dans la clause FROM. Par exemple, au lieu de renvoyer l'entité Phone cas, les numéros de téléphone peuvent être renvoyés à la place.

```
SELECT p. Numéro
FROM Employé e JOIN e.phones p
```

Dans la définition des expressions de chemin précédemment, il a été noté qu'un chemin ne pouvait pas continuer à partir d'un champ d'état ou d'un champ d'association de collection. Pour contourner cette situation, le champ d'association de collection doit être joint dans la clause FROM afin qu'un nouveau

une variable d'identification est créée pour le chemin, ce qui lui permet d'être la racine du nouveau chemin expressions.

Épisode 344

IN VERSUS JOIN

eJBQL tel que défini par les spécifications eJB 2.0 et eJB 2.1 utilisait un opérateur spécial IN dans le From clause pour mapper les associations de collection aux variables d'identification. Prise en charge de cet opérateur a été transféré à Jp QL. la forme équivalente de la requête utilisée précédemment dans cette section pourrait être spécifié comme:

```
CHOISIR DISTINCT p
FROM Employé e, IN (e.phones) p
```

l'opérateur IN est destiné à indiquer que la variable p est une énumération des téléphones collection. l'opérateur JOIN est un moyen plus puissant et plus expressif de déclarer des relations et est l'opérateur recommandé pour les requêtes.

Opérateur JOIN et champs d'association à valeur unique

L'opérateur JOIN fonctionne avec les expressions de chemin d'association à valeur de collection et expressions de chemin d'association à valeur unique. Prenons l'exemple suivant:

```
SELECT d
FROM Employé e REJOINDRE e.département d
```

Cette requête définit une jointure d'un employé au service à travers le service relation. Ceci est sémantiquement équivalent à l'utilisation d'une expression de chemin dans SELECT clause pour obtenir le département pour l'employé. Par exemple, la requête suivante doit aboutir à des représentations SQL similaires sinon identiques impliquant une jointure entre les Entités des employés et des services:

```
SELECT e.department
DE Employé e
```

Le cas d'utilisation principal de l'utilisation d'une expression de chemin d'association à valeur unique dans le La clause FROM (plutôt que d'utiliser simplement une expression de chemin dans la clause SELECT) est pour rejoint. La navigation dans le chemin équivaut à la jointure interne de toutes les entités associées traversées dans l'expression du chemin.

Épisode 345

La possibilité de jointures internes implicites résultant d'expressions de chemin est quelque chose à Soyez conscient de. Prenons l'exemple suivant qui renvoie les différents départements basés en Californie qui participent au projet Release1:

```
SELECT DISTINCT e.department
```

```
FROM Projet p REJOINDRE p. Employés e
WHERE p.name = 'Release1' ET
        e.address.state = 'CA'
```

Il y a en fait quatre jointures logiques ici, pas deux. Le traducteur traitera la requête comme s'il avait été écrit avec des jointures explicites entre les différentes entités. Nous couvrirons le syntaxe pour les jointures multiples plus loin dans la section "Jointures multiples", mais pour l'instant, considérez la requête suivante équivalente à la requête précédente, lecture des conditions de jointure à partir de de gauche à droite:

```
CHOISIR DISTINCT d
FROM Project p REJOINDRE p.employés e REJOINDRE e.department d JOIN e.address a
WHERE p.name = 'Release1' ET
        a.state = 'CA'
```

Nous disons quatre jointures logiques car le mappage physique réel pourrait impliquer plus les tables. Dans ce cas, les entités Employé et Projet sont liées via un plusieurs-à-plusieurs association utilisant une table de jointure. Par conséquent, le SQL réel pour une telle requête utilise cinq tables, pas quatre.

```
SELECT DISTINCT d.id, d.name
FROM projet p, emp_projects ep, emp e, dept d, adresse a
O p.id = ep.project_id ET
        ep.emp_id = e.id ET
        e.dept_id = d.id ET
        e.address_id = a.id ET
        p.name = 'Release1' ET
        a.state = 'CA'
```

La première forme de la requête est certainement plus facile à lire et à comprendre. cependant, pendant le réglage des performances, il peut être utile de comprendre combien de jointures peuvent se produire comme résultat d'expressions de chemin apparemment triviales.

Épisode 346

Rejoignez les conditions dans la clause WHERE

Les requêtes SQL ont traditionnellement joint les tables entre elles en répertoriant les tables à joindre la clause FROM et la fourniture de critères dans la clause WHERE de la requête pour déterminer les conditions de jointure. Pour joindre deux entités sans utiliser de relation, utilisez une variable de plage déclaration pour chaque entité dans la clause FROM.

L'exemple de jointure précédent entre les entités Employee et Department pourrait également ont été écrits comme ceci:

```
CHOISIR DISTINCT d
DE Département d, employé e
WHERE d = e.département
```

Ce style de requête est généralement utilisé pour compenser le manque de relation entre deux entités dans le modèle de domaine. Par exemple, il n'y a pas association entre l'entité du Ministère et le Salarié qui est le gestionnaire du département.

Nous pouvons utiliser une condition de jointure dans la clause WHERE pour rendre cela possible.

```
SÉLECTIONNER d, m
FROM Département d, Employé m
WHERE d = m.département ET
```

m.directs N'EST PAS VIDE

Dans cet exemple, nous utilisons l'une des expressions de collection spéciales, IS NOT EMPTY, pour vérifier que la collection des subordonnés directs à l'employé n'est pas vide. Tout employé avec une collection non vide de directives est par définition un gestionnaire.

Jointures multiples

Plus d'une jointure peut être mise en cascade si nécessaire. Par exemple, la requête suivante renvoie l'ensemble distinct de projets appartenant aux employés appartenant à un service:

```
CHOISIR DISTINCT p
DU Département d REJOINDRE d. Employés e REJOINDRE e.projets p
```

Le processeur de requêtes interprète la clause FROM de gauche à droite. Une fois qu'une variable a déclaré, il peut être référencé par la suite par d'autres expressions JOIN. Dans ce cas, la relation projets de l'entité Employé est parcourue une fois la variable Employé a été déclaré.

Épisode 347

Jointures de carte

Une expression de chemin qui navigue à travers une association à valeur de collection implémentée comme une carte est un cas particulier. Contrairement à une collection normale, chaque élément d'une carte correspond à deux informations: la clé et la valeur. Lorsque vous travaillez avec JP QL, c'est Il est important de noter que les variables d'identification basées sur des cartes se réfèrent à la valeur par défaut. Par exemple, considérons le cas où la relation téléphonique de l'employé l'entité est modélisée comme une carte, où la clé est le type de numéro (travail, cellule, domicile, etc.) et la valeur est le numéro de téléphone. La requête suivante énumère les numéros de téléphone pour tous les employés:

```
SELECT e.name, p
FROM Employé e JOIN e.phones p
```

Ce comportement peut être mis en évidence explicitement grâce à l'utilisation du mot clé VALUE. Pour exemple, la requête précédente est fonctionnellement identique à la suivante:

```
SELECT e.name, VALUE (p)
FROM Employé e JOIN e.phones p
```

Pour accéder à la clé au lieu de la valeur d'un élément de carte donné, nous pouvons utiliser la clé mot-clé pour remplacer le comportement par défaut et renvoyer la valeur de clé pour un élément de carte donné. L'exemple suivant illustre l'ajout du type de téléphone à la requête précédente:

```
SELECT e.name, KEY (p), VALUE (p)
FROM Employé e JOIN e.phones p
WHERE KEY (p) IN ('Work', 'Cell')
```

Enfin, si nous voulons que la clé et la valeur soient renvoyées ensemble dans le forme d'un objet java.util.Map.Entry, nous pouvons spécifier le mot clé ENTRY dans le même mode. Notez que le mot clé ENTRY ne peut être utilisé que dans la clause SELECT. La clé et VALUE peuvent également être utilisés dans le cadre d'expressions conditionnelles dans les champs WHERE et HAVING clauses de la requête.

Notez que dans chacun des exemples de jointure de carte, nous avons joint une entité par rapport à l'un de ses Mappez les attributs et est sorti avec une clé, une valeur ou une paire clé-valeur (entrée). Cependant, quand vu du point de vue des tables, la jointure ne se fait qu'au niveau du la clé primaire de l'entité source et les valeurs de la carte. Aucune installation n'est actuellement disponible en JPA pour rejoindre l'entité source contre les clés de la carte.

Épisode 348

Jointures externes

Une jointure externe entre deux entités produit un domaine dans lequel un seul côté du la relation doit être complète. En d'autres termes, la jointure externe de l'employé à Service à travers la relation de service employé renvoie tous les employés et le service auquel l'employé a été affecté, mais le service est retourné seulement s'il est disponible. Ceci est en contraste avec une jointure interne qui ne renverrait que ceux employés affectés à un service.

Une jointure externe est spécifiée à l'aide de la syntaxe suivante: LEFT [OUTER] JOIN <chemin_ expression> [AS] <identificateur>. La requête suivante illustre une jointure externe entre deux entités:

```
SÉLECTIONNER e, d
FROM Employé e GAUCHE REJOINDRE e.département d
```

Si l'employé n'a pas été affecté à un service, l'objet de service (le deuxième élément du tableau Object) sera nul.

Dans une génération SQL de fournisseur typique, vous verrez que la requête précédente serait équivalent à ce qui suit:

```
SELECT e.id, e.name, e.salary, e.manager_id, e.dept_id, e.address_id,
            d.id, d.name
FROM employé e GAUCHE OUTER JOIN département d
ON (d.id = e.department_id)
```

Le SQL résultant montre que lorsqu'une jointure externe est générée à partir de JP QL, elle spécifie une condition d'égalité ON entre la colonne de jointure qui mappe la relation étant joint à travers, et la clé primaire à laquelle il fait référence.

Une expression ON supplémentaire peut être fournie pour ajouter des contraintes aux objets qui renvoyé du côté droit de la jointure. Par exemple, nous pouvons modifier le précédent JP QL requête pour avoir une condition ON supplémentaire pour limiter les départements retournés à ces seuls qui ont un préfixe «QA»:

```
SÉLECTIONNER e, d
FROM Employé e GAUCHE REJOINDRE e.département d
     ON d.name COMME 'QA%'
```


Épisode 349

Cette requête renvoie toujours tous les employés, mais les résultats n'incluront aucun départements ne correspondant pas à la condition ON ajoutée. Le SQL généré ressemblerait à cette:

```
SELECT e.id, e.name, e.salary, e.department_id, e.manager_id, e.address_id,
                        d.id, d.name
FROM employé e gauche service de jointure externe d
     ON ((d.id = e.department_id) et (d.name comme 'QA%'))
```

Notez que cette requête est très différente de l'utilisation d'une expression WHERE:

```
SÉLECTIONNER e, d
FROM Employé e GAUCHE REJOINDRE e.département d
WHERE d.name COMME 'QA%'
```

La clause WHERE entraîne une sémantique de jointure interne entre Employee et Department, donc cette requête ne renverrait que les employés qui étaient dans un service avec un 'QA' nom préfixé.

Tip la possibilité d'ajouter des conditions de jointure externe avec ON a été ajoutée dans Jpa 2.1.

Récupérer les jointures

Les jointures de récupération sont destinées à aider les concepteurs d'applications à optimiser l'accès à leur base de données et préparez les résultats de la requête pour le détachement. Ils permettent aux requêtes de spécifier un ou plusieurs relations qui doivent être parcourues et préchargées par le moteur de recherche afin qu'elles ne sont pas chargés ultérieurement lors de l'exécution.

Par exemple, si nous avons une entité Employee avec une relation de chargement différé avec sa adresse, la requête suivante peut être utilisée pour indiquer que la relation doit être résolu avec empressement lors de l'exécution de la requête:

```
SÉLECTIONNER e
FROM Employé e JOIN FETCH e.address
```

Notez qu'aucune variable d'identification n'est définie pour l'expression de chemin e.address. Cette est parce que même si l'entité Address est jointe afin de résoudre le relation, il ne fait pas partie du type de résultat de la requête. Le résultat de l'exécution de la requête est toujours une collection d'instances d'entité Employee, sauf que la relation d'adresse sur

Épisode 350

chaque entité ne provoquera pas un voyage secondaire vers la base de données lors de son accès. Ça aussi permet d'accéder en toute sécurité à la relation d'adresse si l'entité Employé devient détaché. Une jointure de récupération se distingue d'une jointure régulière en ajoutant le mot clé FETCH à l'opérateur JOIN.

Afin d'implémenter les jointures d'extraction, le fournisseur doit activer l'association extraite dans une jointure régulière du type approprié: interne par défaut ou externe si le mot clé LEFT a été spécifié. L'expression SELECT de la requête doit également être développée pour inclure la relation jointe. Exprimé en JP QL, une interprétation équivalente du fournisseur de L'exemple précédent de jointure d'extraction ressemblerait à ceci:

```
SELECT e, a
FROM Employé e JOIN e.address a
```

La seule différence est que le fournisseur ne renvoie pas réellement les entités Address à l'appelant. Étant donné que les résultats sont traités à partir de cette requête, le moteur de requête crée l'entité Address en mémoire et l'affecte à l'entité Employee, puis la supprime à partir de la collection de résultats qu'il crée pour le client. Cela charge avec impatience l'adresse relation, qui peut ensuite être accessible par navigation à partir de l'entité Employee.

Une conséquence de l'implémentation des jointures de récupération de cette manière est que la récupération d'une collection l'association entraîne des résultats en double. Par exemple, considérons une requête de service où la relation d'employés de l'entité Département est recherchée avec impatience. La jointure de récupération requête, cette fois en utilisant une jointure externe pour s'assurer que les départements sans employés sont récupéré, serait écrit comme suit:

```
SELECT d
DU Département d GAUCHE REJOINDRE FETCH d. Employés
```

Exprimée en JP QL, l'interprétation du fournisseur remplacerait la récupération par un rejoindre à travers la relation collaborateurs:

```
SÉLECTIONNER d, e
DU Département d GAUCHE REJOINDRE d. Employés e
```

Une fois de plus, au fur et à mesure que les résultats sont traités, l'entité Employé est construite en mémoire mais supprimée de la collection de résultats. Chaque entité du Département dispose désormais d'un Collecte des employés résolus, mais le client reçoit une référence pour chaque service par employé. Par exemple, si quatre départements de cinq employés chacun ont été récupérés, le résultat serait une collection de 20 instances de département, avec chaque département

Épisode 351

dupliqué cinq fois. Les instances d'entité réelles pointent toutes vers le même versions, mais les résultats sont pour le moins assez étranges.

Pour éliminer les valeurs en double, soit l'opérateur DISTINCT doit être utilisé, soit le les résultats doivent être placés dans une structure de données telle qu'un ensemble. Parce qu'il n'est pas possible de écrire une requête SQL qui utilise l'opérateur DISTINCT tout en préservant la sémantique du fetch join, le fournisseur devra éliminer les doublons en mémoire une fois que les résultats auront été récupéré. Cela peut avoir des implications sur les performances des grands ensembles de résultats.

Étant donné les résultats quelque peu particuliers générés par une jointure d'extraction vers une collection, il peut ne pas être le moyen le plus approprié de charger avec empressement des entités associées dans tous les cas. Si un la collecte nécessite une récupération avide de manière régulière, pensez à établir la relation avide par défaut. Certains fournisseurs de persistance proposent également des lectures par lots comme alternative à récupérer les jointures qui émettent plusieurs requêtes dans un même lot, puis corréler les résultats à chargez avidement les relations. Une autre alternative consiste à utiliser un graphe d'entité pour dynamiquement déterminer les attributs de relation à charger par une requête. Les graphiques d'entité sont décrit en détail au chapitre 11 .

Clause WHERE

La clause WHERE d'une requête est utilisée pour spécifier les conditions de filtrage afin de réduire le jeu de résultats. Dans cette section, nous explorons les fonctionnalités de la clause WHERE et les types d'expressions qui peut être formé pour filtrer les résultats de la requête.

La définition de la clause WHERE est d'une simplicité trompeuse. C'est simplement le mot-clé WHERE, suivi d'une expression conditionnelle. Cependant, comme les sections suivantes démontrer, JP QL prend en charge un ensemble puissant d'expressions conditionnelles pour filtrer le plus sophistiqué des requêtes.

Paramètres d'entrée

Les paramètres d'entrée pour les requêtes peuvent être spécifiés en utilisant la notation positionnelle ou nommée. La notation positionnelle est définie en préfixant le numéro de variable avec un point d'interrogation. Considérez la requête suivante:

```
SÉLECTIONNER e
DE Employé e
O e.salary>? 1
```

Épisode 352

À l'aide de l'interface de requête, toute valeur double ou valeur compatible avec le type
l'attribut salaire, peut être lié au premier paramètre afin d'indiquer le
limite des salaires des employés dans cette requête. Le même paramètre de position peut se produire plus
une fois dans la requête. La valeur liée au paramètre sera substituée à chaque
de ses occurrences.

Les paramètres nommés sont spécifiés à l'aide d'un signe deux-points suivi d'un identificateur. Voici la
même requête, cette fois en utilisant un paramètre nommé:

```
SÉLECTIONNER e
DE Employé e
WHERE e.salary>: sal
```

Les paramètres d'entrée sont traités en détail au chapitre 7 .

Formulaire d'expression de base

Une grande partie de la prise en charge des expressions conditionnelles dans JP QL est empruntée directement à SQL. Cette
est intentionnel et aide à faciliter la transition pour les développeurs déjà familiarisés avec
SQL. La principale différence entre les expressions conditionnelles dans JP QL et SQL est que JP
Les expressions QL peuvent exploiter les variables d'identification et les expressions de chemin pour naviguer
relations lors de l'évaluation de l'expression.

Les expressions conditionnelles sont construites dans le même style que SQL conditionnel
expressions, utilisant une combinaison d'opérateurs logiques, d'expressions de comparaison, de primitives
et les opérations de fonction sur les champs, et ainsi de suite. Bien qu'un résumé des opérateurs soit
fourni plus tard, la grammaire des expressions conditionnelles n'est pas répétée ici. Le JPA
la spécification contient la grammaire sous forme de Backus-Naur (BNF) et est l'endroit où chercher
pour les règles exactes sur l'utilisation des expressions de base. Cependant, les sections suivantes
expliquer les opérateurs et expressions de niveau supérieur, en particulier ceux propres à JP QL,
et ils fournissent des exemples pour chacun.

La syntaxe littérale est également similaire à SQL (voir la section «Littéraux»).

La priorité des opérateurs est la suivante.

1. Opérateur de navigation (.)
2. Unaire +/–
3. Multiplication (*) et division (/)
4. Addition (+) et soustraction (-)

Épisode 353

5. Opérateurs de comparaison =,>,> =, <, <=, <>, [NOT] BETWEEN,
   [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] VIDE, [NOT]
   MEMBRE DE]
6. Opérateurs logiques (AND, OR, NOT)

ENTRE Expressions

L'opérateur BETWEEN peut être utilisé dans des expressions conditionnelles pour déterminer si le
le résultat d'une expression appartient à une plage de valeurs inclusive. Numérique, chaîne et date
les expressions peuvent être évaluées de cette manière. Prenons l'exemple suivant:

```
SÉLECTIONNER e
```

DE Employé e
O e.salary ENTRE 40000 ET 45000

Tout employé gagnant de 40 000 $ à 45 000 $ inclusivement est inclus dans les résultats. C'est identique à la requête suivante utilisant des opérateurs de comparaison de base:

SÉLECTIONNER e
DE Employé e
O e.salary> = 40000 ET e.salary <= 45000

L'opérateur BETWEEN peut également être annulé avec l'opérateur NOT.

AIMEZ les expressions

JP QL prend en charge la condition SQL LIKE pour fournir une forme limitée de modèle de chaîne correspondant à. Chaque expression LIKE se compose d'une expression de chaîne à rechercher et d'un chaîne de modèle et séquence d'échappement facultative qui définit les conditions de correspondance. le les caractères génériques utilisés par la chaîne de modèle sont le trait de soulignement (_) pour un caractère unique les caractères génériques et le signe de pourcentage (%) pour les caractères génériques à plusieurs caractères.

SELECT d
DU Département d
WHERE d.name LIKE '__Eng%'

Nous utilisons un préfixe de deux caractères de soulignement pour remplacer les deux premiers caractères de la chaîne candidats, donc des exemples de noms de département correspondant à cette requête serait CAEngOtt ou USEngCal, mais pas CADocOtt. Notez que les correspondances de motifs sont à la casse sensible.

Épisode 354

Si la chaîne de modèle contient un trait de soulignement ou un signe de pourcentage qui doit être littéralement correspond, la clause ESCAPE peut être utilisée pour spécifier un caractère qui, lors du préfixe d'un caractère générique, indique qu'il doit être traité littéralement:

SELECT d
DU Département d
WHERE d.name LIKE 'QA \ _%' ESCAPE '\'

Échapper le trait de soulignement en fait une partie obligatoire de l'expression. Par exemple, QA_East correspondrait, mais pas QANorth.

Sous-requêtes

Les sous-requêtes peuvent être utilisées dans les clauses WHERE et HAVING d'une requête. Une sous-requête est un requête de sélection complète à l'intérieur d'une paire de parenthèses qui est incorporée dans un conditionnel expression. Les résultats de l'exécution de la sous-requête (qui sera soit un résultat scalaire, soit une collection de valeurs) sont ensuite évaluées dans le contexte de l'expression conditionnelle.
Les sous-requêtes sont une technique puissante pour résoudre les scénarios de requêtes les plus complexes.

Considérez la requête suivante:

SÉLECTIONNER e
DE Employé e
WHERE e.salary = (SELECT MAX (emp.salary)
FROM Employé emp)

Cette requête renvoie l'employé avec le salaire le plus élevé parmi tous les employés. Une sous-requête constituée d'une requête agrégée (décrite plus loin dans ce chapitre) est utilisée pour renvoyer la valeur de salaire maximum, puis ce résultat est utilisé comme clé pour filtrer liste des employés par salaire. Une sous-requête peut être utilisée dans la plupart des expressions conditionnelles et peut apparaissent sur le côté gauche ou droit d'une expression.

La portée d'un nom de variable identificateur commence dans la requête où il est défini et s'étend à toutes les sous-requêtes. Les identifiants de la requête principale peuvent être référencés par un sous-requête, et les identifiants introduits par une sous-requête peuvent être référencés par n'importe quelle sous-requête qu'il crée. Si une sous-requête déclare une variable identificateur du même nom, elle remplace la déclaration parente et empêche la sous-requête de faire référence à la variable parente.

Épisode 355

Remarque Le remplacement d'un nom de variable d'identification dans une sous-requête n'est pas garanti être pris en charge par tous les fournisseurs. des noms uniques doivent être utilisés pour garantir la portabilité.

La possibilité de faire référence à une variable de la requête principale dans la sous-requête permet les deux requêtes à corréler. Prenons l'exemple suivant:

```
SÉLECTIONNER e
DE Employé e
O EXISTE (SÉLECTIONNEZ 1
                DE Téléphone p
                O p.employé = e ET p.type = 'Cellule')
```

Cette requête renvoie tous les employés qui ont un numéro de téléphone portable. C'est aussi un exemple de sous-requête qui renvoie une collection de valeurs. L'expression EXISTS dans ce exemple renvoie true si des résultats sont renvoyés par la sous-requête. Renvoyer le littéral 1 de la sous-requête est une pratique standard avec les expressions EXISTS car la les résultats sélectionnés par la sous-requête n'ont pas d'importance; seul le nombre de résultats est pertinent. Remarque que la clause WHERE de la sous-requête fait référence à la variable identificateur e du et l'utilise pour filtrer les résultats de la sous-requête. Conceptuellement, la sous-requête peut être pensée d'exécuter une fois pour chaque employé. En pratique, de nombreux serveurs de bases de données optimiseront ces types de requêtes dans des jointures ou des vues en ligne afin d'optimiser les performances.

Cette requête peut également avoir été écrite à l'aide d'une jointure entre l'employé et Entités téléphoniques avec l'opérateur DISTINCT utilisé pour filtrer les résultats. L'avantage de l'utilisation de la sous-requête corrélée est que la requête principale reste déchargée des jointures à d'autres entités. Très souvent, si une jointure est utilisée uniquement pour filtrer les résultats, il existe un équivalent condition de sous-requête qui peut être utilisée alternativement pour supprimer les contraintes sur le join de la requête principale ou même pour améliorer les performances de la requête.

La clause FROM d'une sous-requête peut également créer de nouvelles variables d'identification à partir de expressions de chemin à l'aide d'une variable d'identification de la requête principale. Par exemple, le la requête précédente pourrait également avoir été écrite comme suit:

```
SÉLECTIONNER e
DE Employé e
O EXISTE (SÉLECTIONNEZ 1
                DES e.phones p
                WHERE p.type = 'Cellule')
```


Épisode 356

Dans cette version de la requête, la sous-requête utilise le chemin d'association de collection téléphones à partir de la variable d'identification des employés e dans la sous-requête. Ceci est ensuite mappé à une variable d'identification locale p utilisée pour filtrer les résultats par type de téléphone. Chaque l'occurrence de p fait référence à un seul téléphone associé à l'employé.

Pour mieux illustrer la façon dont le traducteur gère cette requête, considérez l'équivalent requête écrite en SQL:

```
SELECT e.id, e.name, e.salary, e.manager_id, e.dept_id, e.address_id
DE EMP e
O EXISTE (SÉLECTIONNEZ 1
                DU téléphone p
                O p.emp_id = e.id ET
                        p.type = 'Cellule')
```

L'expression e.phones est convertie en table mappée par l'entité Phone. La clause WHERE pour la sous-requête ajoute ensuite la condition de jointure nécessaire pour corréler la sous-requête à la requête principale, dans ce cas l'expression p.emp_id = e.id. le les critères de jointure appliqués à la table PHONE aboutissent à tous les téléphones appartenant au employé.

IN Expressions

L'expression IN peut être utilisée pour vérifier si une expression de chemin à valeur unique est membre d'une collection. La collection peut être définie en ligne comme un ensemble de valeurs littérales ou peut être dérivé d'une sous-requête. La requête suivante illustre la notation littérale en sélectionnant tous les employés résidant à New York ou en Californie:

```
SÉLECTIONNER e
DE Employé e
WHERE e.address.state IN ('NY', 'CA')
```


Épisode 357

La forme de sous-requête de l'expression est similaire, en remplaçant la liste littérale par un requête imbriquée. La requête suivante renvoie les employés qui travaillent dans des services qui sont contribution à des projets commençant par le préfixe QA:

```
SÉLECTIONNER e
DE Employé e
WHERE e.department IN (SELECT DISTINCT d
                        FROM Department d JOIN d .employees de JOIN
                        de.projets p
                        WHERE p.name LIKE 'QA%')
```

L'expression IN peut également être annulée à l'aide de l'opérateur NOT. Par exemple, le La requête suivante renvoie toutes les entités Phone représentant des numéros de téléphone autres que pour au bureau ou à la maison:

```
SÉLECTIONNER p
DE Téléphone p
```

WHERE p.type NOT IN ('Bureau', 'Maison')

Expressions de collection

L'opérateur IS EMPTY est l'équivalent logique de IS NULL, mais pour les collections. Requêtes peut utiliser l'opérateur EST VIDE ou sa forme inversée N'EST PAS VIDE pour vérifier si un Le chemin d'association de collection se résout en une collection vide ou a au moins une valeur. Par exemple, la requête suivante renvoie tous les employés qui sont managers en vertu de ayant au moins un subordonné direct:

```
SÉLECTIONNER e
DE Employé e
O e.directs N'EST PAS VIDE
```

Notez que les expressions IS EMPTY sont traduites en SQL en tant qu'expressions de sous-requête. le query tqwranslator peut utiliser une sous-requête d'agrégation ou utiliser l'expression SQL EXISTS. Par conséquent, la requête suivante est équivalente à la précédente:

```
SÉLECTIONNER m
DE Employé m
O (SÉLECTIONNER LE NOMBRE (e)
        DE Employé e
        O e.manager = m)> 0
```


Épisode 358

L'opérateur MEMBER OF et sa forme niée NOT MEMBER OF sont une manière abrégée de vérifier si une entité est membre d'un chemin d'association de collection. Le suivant La requête renvoie tous les gestionnaires qui sont saisis à tort comme relevant d'eux-mêmes:

```
SÉLECTIONNER e
DE Employé e
O e MEMBRE DE e.directs
```

Une utilisation plus typique de l'opérateur MEMBER OF est en conjonction avec une entrée paramètre. Par exemple, la requête suivante sélectionne tous les employés affectés à un projet spécifié:

```
SÉLECTIONNER e
DE Employé e
O: projet MEMBRE DE e.projects
```

Comme l'expression IS EMPTY, l'expression MEMBER OF sera traduite en SQL en utilisant une expression EXISTS ou la forme de sous-requête de l'expression IN. le l'exemple précédent équivaut à la requête suivante:

```
SÉLECTIONNER e
DE Employé e
WHERE: projet IN (SELECT p
                  DE e.projects p)
```

Expressions EXISTES

La condition EXISTS renvoie vrai si une sous-requête renvoie des lignes. Exemples d'existants ont été démontrés plus tôt dans l'introduction aux sous-requêtes. L'opérateur EXISTS peut également être annulée avec l'opérateur NOT. La requête suivante sélectionne tous les employés qui ne pas avoir de téléphone portable:

```
SÉLECTIONNER e
DE Employé e
O N'EXISTE PAS (SELECT p
```

```
DES e.phones p
WHERE p.type = 'Cellule')
```


Épisode 359

TOUTES, TOUTES et CERTAINES expressions

Les opérateurs ANY, ALL et SOME peuvent être utilisés pour comparer une expression aux résultats de une sous-requête. Prenons l'exemple suivant:

```
SÉLECTIONNER e
DE Employé e
O e.directs N'EST PAS VIDE ET
        e.salary <ALL (SELECT d.salary
                        DE e.directs d)
```

Cette requête renvoie les managers qui sont moins payés que tous les salariés qui travaillent pour eux. La sous-requête est évaluée, puis chaque valeur de la sous-requête est comparée à l'expression de gauche, dans ce cas le salaire du manager. Lorsque l'opérateur ALL est utilisé, le la comparaison entre le côté gauche de l'équation et tous les résultats de la sous-requête doit être vraie pour que la condition générale soit vraie.

L'opérateur ANY se comporte de la même manière, mais la condition générale est vraie tant qu'à au moins une des comparaisons entre l'expression et le résultat de la sous-requête est vraie. Pour exemple, si ANY a été spécifié au lieu de ALL dans l'exemple précédent, le résultat de la la requête serait tous les gestionnaires qui ont été payés moins qu'au moins un de leurs employés. L'opérateur SOME est un alias pour l'opérateur ANY.

Il existe une symétrie entre les expressions IN et l'opérateur ANY. Prendre en compte suite de la variation de l'exemple du département de projet utilisé précédemment, modifié uniquement légèrement pour utiliser ANY au lieu de IN:

```
SÉLECTIONNER e
DE Employé e
WHERE e.department = TOUT (SELECT DISTINCT d
                            FROM Department d JOIN d .employees de JOIN
                            de.projets p
                            WHERE p.name LIKE 'QA%')
```

Héritage et polymorphisme

JPA prend en charge l'héritage entre les entités. Par conséquent, le langage de requête prend en charge résultats polymorphes où plusieurs sous-classes d'une entité peuvent être renvoyées par la même requete.

Épisode 360

Dans l'exemple de modèle, Project est une classe de base pour QualityProject et Projet de design. Si une variable d'identification est formée à partir de l'entité Projet, la requête les résultats comprendront un mélange d'objets Project, QualityProject et DesignProject

et les résultats peuvent être transtypés dans les sous-classes par l'appelant si nécessaire. Le suivant query récupère tous les projets avec au moins un employé:

SÉLECTIONNER p
FROM Projet p
O les employés ne sont pas vides

Discrimination de sous-classe

Si nous voulons restreindre le résultat de la requête à une sous-classe particulière, nous pouvons utiliser cela sous-classe particulière dans la clause FROM au lieu de la racine. Cependant, si nous voulons restreindre les résultats à plus d'une sous-classe dans la requête mais pas à toutes les sous-classes, nous pouvons utilisez plutôt une expression de type dans la clause WHERE pour filtrer les résultats. Une expression de type se compose du mot-clé TYPE suivi d'une expression entre parenthèses qui se résout en une entité. Le résultat d'une expression de type est le nom de l'entité, qui peut ensuite être utilisé pour à des fins de comparaison de type. L'avantage d'une expression de type est que l'on peut distinguer entre les types sans s'appuyer sur un mécanisme de discrimination dans le modèle de domaine lui-même.

L'exemple suivant illustre l'utilisation d'une expression TYPE pour renvoyer uniquement la conception et projets de qualité:

SÉLECTIONNER p
FROM Projet p
WHERE TYPE (p) = DesignProject OU TYPE (p) = QualityProject

Notez qu'il n'y a pas de guillemets autour de DesignProject et QualityProject identifiants. Ceux-ci sont traités comme des noms d'entités dans JP QL et non comme des chaînes. Malgré cela distinction, les paramètres d'entrée peuvent être utilisés à la place des noms codés en dur dans la requête cordes. La création d'une requête paramétrée qui renvoie des instances d'un type de sous-classe donné est simple, comme illustré par la requête suivante:

SÉLECTIONNER p
FROM Projet p
WHERE TYPE (p) =: projectType

Épisode 361

Abattu

Dans la plupart des cas, au moins une des sous-classes contient un état supplémentaire, tel que l'attribut qaRating dans QualityProject. Un attribut de sous-classe est accessible directement si la requête ne concernait que les entités de la sous-classe, mais lorsque la requête s'étendait sur une superclasse, le downcasting doit être utilisé. Le downcasting est la technique de une expression qui fait référence à une superclasse soit appliquée à une sous-classe spécifique. C'est réalisé en utilisant l'opérateur TREAT.

TREAT peut être utilisé dans la clause WHERE pour filtrer les résultats en fonction de l'état du sous-type de les instances. La requête suivante renvoie tous les projets de conception ainsi que toute la qualité les projets dont la cote de qualité est supérieure à 4:

SÉLECTIONNER p
FROM Projet p
WHERE TREAT (p AS QualityProject) .qaRating> 4
OR TYPE (p) = DesignProject

La syntaxe de l'expression commence par le mot clé TREAT, suivi de son argument entre parenthèses. L'argument est une expression de chemin, suivie du mot clé AS puis le nom d'entité du sous-type cible. L'expression de chemin doit se résoudre en un

superclasse du type cible. L'expression abaissée résultante est résolue en cible
sous-type, afin que tous les attributs spécifiques au sous-type puissent être ajoutés au chemin résultant
expression, tout comme qaRating l'était dans l'exemple.

Plusieurs expressions TREAT peuvent être incluses dans la clause WHERE, chaque
le même ou un type d'entité différent.

Normalement, lorsqu'une jointure est effectuée, elle inclut toutes les sous-classes de la cible
type d'entité dans la relation en cours de jonction. Pour limiter la jointure afin de ne considérer qu'un
hiérarchie de sous-classe spécifique, une expression TREAT peut être utilisée dans la clause FROM. Attribution
il un identifiant fournit le bonus supplémentaire que l'identifiant peut être référencé à la fois dans
la clause WHERE ainsi que la clause SELECT. La requête suivante renvoie tous les
les employés qui travaillent sur des projets de qualité avec une cote de qualité supérieure à 4, plus le nom
du projet sur lequel ils travaillent et de sa cote de qualité:

```
SELECT e, q.name, q.qaRating
FROM Employé e REJOINDRE UN TRAITEMENT (e.projects AS QualityProject) q
WHERE q.qaClassement> 4
```


Épisode 362

L'expression TREAT peut être utilisée de la même manière pour d'autres types de jointures.
comme jointures externes et extraire les jointures.

Tip Downcasting avec l'expression TREAT a été ajouté dans Jpa 2.1.

L'impact de l'héritage entre entités sur le SQL généré est important
à comprendre pour des raisons de performances et est décrit au chapitre 10 .

Expressions scalaires

Une expression scalaire est une valeur littérale, une séquence arithmétique, une expression de fonction, un type
expression ou expression de cas qui se résout en une seule valeur scalaire. Il peut être utilisé dans
la clause SELECT pour mettre en forme les champs projetés dans les requêtes de rapport ou dans le cadre de conditions
expressions dans la clause WHERE ou HAVING d'une requête. Sous-requêtes qui se résolvent en scalaire
les valeurs sont également considérées comme des expressions scalaires, mais ne peuvent être utilisées que lors de la composition
critères dans la clause WHERE d'une requête. Les sous-requêtes ne peuvent jamais être utilisées dans SELECT
clause.

Littéraux

Il existe un certain nombre de types de littéraux différents qui peuvent être utilisés dans JP QL, y compris les chaînes,
numériques, booléens, énumérations, types d'entités et types temporels.

Tout au long de ce chapitre, nous avons montré de nombreux exemples de chaîne, d'entier et
littéraux booléens. Les guillemets simples sont utilisés pour délimiter les littéraux de chaîne et échappés dans un
chaîne en préfixant le guillemet avec un autre guillemet simple. Chiffres exacts et approximatifs
peut être défini selon les conventions du langage de programmation Java ou par
en utilisant la syntaxe standard SQL-92. Les valeurs booléennes sont représentées par les littéraux TRUE
et FAUX.

Les requêtes peuvent référencer des types d'énumération Java en spécifiant le nom complet de
la classe enum. L'exemple suivant illustre l'utilisation d'une énumération dans un conditionnel
expression, à l'aide de l'énumération PhoneType illustrée dans la liste 5-8 du chapitre 5 :

```
SÉLECTIONNER e
DE Employé e REJOINDRE e.phoneNumbers p
WHERE KEY (p) = com.acme.PhoneType.Home
```

Épisode 363

Un type d'entité est juste le nom d'entité d'une entité définie et n'est valide que lorsqu'il est utilisé avec l'opérateur TYPE. Les devis ne sont pas utilisés. Voir la section «Héritage et Polymorphisme »pour des exemples d'utilisation d'un littéral de type d'entité.

Les littéraux temporels sont spécifiés à l'aide de la syntaxe d'échappement JDBC, qui définit que les accolades entourent le littéral. Le premier caractère de la séquence est soit un «d» soit un «T» pour indiquer que le littéral est une date ou une heure, respectivement. Si le littéral représente un horodatage, «ts» est utilisé à la place. Après l'indicateur de type se trouve un séparateur d'espace, et puis les informations de date, d'heure ou d'horodatage réelles entourées de guillemets simples. le formes générales des trois types littéraux temporels, avec des exemples suit:

{d 'aaaa-mm-jj'}	par exemple {d '2009-11-05'}
{t 'hh-mm-ss'}	par exemple {t '12 -45-52 '}
{ts 'aaaa-mm-jj hh-mm-ss.f'} ex	{ts '2009-11-05 12-45-52.325'}

Toutes les informations temporelles entre guillemets simples sont exprimées sous forme de chiffres. le la partie fractionnaire de l'horodatage (la partie «.f») peut comporter plusieurs chiffres et est facultative.

Lorsque vous utilisez l'un de ces littéraux temporels, rappelez-vous qu'ils ne sont interprétés que par des pilotes prenant en charge la syntaxe d'échappement JDBC. Le fournisseur n'essaiera normalement pas de traduire ou prétraiter les littéraux temporels.

Expressions de fonction

Les expressions scalaires peuvent tirer parti des fonctions qui peuvent être utilisées pour transformer les résultats des requêtes. Le tableau 8-1 résume la syntaxe de chacune des expressions de fonction prises en charge.

Tableau 8-1. *Expressions de fonction prises en charge*

Une fonction	La description
ABS (nombre)	renvoie la version non signée de l'argument numérique. le type de résultat est le même que le type d'argument (entier, flottant ou double).
CONCAT (chaîne1, chaîne2)	renvoie une nouvelle chaîne qui est la concaténation de son arguments, chaîne1 et chaîne2.
DATE ACTUELLE	renvoie la date actuelle telle que définie par la base de données serveur.

(*suite*)

Épisode 364

Tableau 8-1. (*suite*)

Une fonction	La description
HEURE ACTUELLE	renvoie l'heure actuelle telle que définie par la base de données serveur.
CURRENT_TIMESTAMP	renvoie l'horodatage actuel tel que défini par le

	serveur de base de données.
INDEX (variable d'identification)	renvoie la position d'une entité dans une liste ordonnée.
LENGTH (chaîne)	renvoie le nombre de caractères de la chaîne argument.
LOCATE (chaîne1, chaîne2 [, début])	renvoie la position de string1 dans string2, facultativement à partir de la position indiquée par start. le résultat est zéro si la chaîne est introuvable.
LOWER (chaîne)	renvoie la forme minuscule de l'argument chaîne.
MOD (nombre1, nombre2)	renvoie le module des arguments numériques nombre1 et nombre2 sous la forme d'un entier.
TAILLE (collection)	renvoie le nombre d'éléments de la collection, ou zéro si la collection est vide.
SQRT (nombre)	renvoie la racine carrée de l'argument numérique sous la forme d'un double.
SUBSTRING (chaîne, début, fin)	renvoie une partie de la chaîne d'entrée, en commençant à l'index indiqué par start jusqu'à la longueur des caractères. Les index de chaîne sont mesurés à partir de un.
UPPER (chaîne)	renvoie la forme majuscule de l'argument chaîne.
TRIM ([[LEADING | TRAILING | BOTH] [char] FROM] chaîne)	supprime les caractères de début et / ou de fin d'un chaîne. Si les options LEADING, TRAILING ou BOTH Le mot clé n'est pas utilisé, à la fois les caractères de début et de fin sont enlevés. le caractère de découpe par défaut est l'espace personnage.

Épisode 365

La fonction SIZE nécessite une attention particulière car il s'agit d'une notation abrégée pour une sous-requête agrégée. Par exemple, considérez la requête suivante qui renvoie tout départements avec seulement deux employés:

```
SELECT d
DU Département d
O TAILLE (d. Employés) = 2
```

Tout comme les expressions de collection IS EMPTY et MEMBER OF, la fonction SIZE sera traduit en SQL à l'aide d'une sous-requête. La forme équivalente de l'exemple précédent utilisant un la sous-requête est la suivante:

```
SELECT d
DU Département d
O (SÉLECTIONNER LE NOMBRE (e)
        DE d.employés e) = 2
```

Le cas d'utilisation de la fonction INDEX peut ne pas être évident au début. Lors de l'utilisation collections ordonnées, chaque élément de la collection contient en fait deux pièces de information: la valeur stockée dans la collection et sa position numérique dans le collection. Les requêtes peuvent utiliser la fonction INDEX pour déterminer la position numérique d'un élément dans une collection, puis utilisez ce numéro à des fins de rapport ou de filtrage. Pour Par exemple, si les numéros de téléphone d'un employé sont stockés par ordre de priorité, les La requête renverrait le premier (et le plus important) numéro de chaque employé:

```
SELECT e.name, p. Numéro
FROM Employé e JOIN e.phones p
O INDEX (p) = 0
```

Fonctions de base de données natives

L'un des avantages de JP QL est qu'il dissocie l'application du sous-jacent
base de données. Cependant, il est parfois nécessaire d'utiliser des fonctions natives qui sont soit
indigène à la base de données ou défini par l'administrateur système. Alors que l'utilisation de
ces fonctions peuvent lier la requête à la base de données cible, il y a toujours un argument pour
en utilisant l'indépendance de cartographie d'entité de JP QL.

Les fonctions de la base de données sont accessibles dans les requêtes JP QL via l'utilisation du
Expression FUNCTION. Le mot-clé FUNCTION, suivi du nom de la fonction et
les arguments de la fonction, doivent être résolus en une valeur scalaire qui est arithmétique, booléenne, chaîne,

Épisode 366

ou un type temporel, tel que la date, l'heure ou l'horodatage. Les expressions FUNCTION peuvent être utilisées
partout où les types scalaires tiennent dans une expression, et le type de résultat doit correspondre à ce que le reste
de l'expression attend. Les arguments doivent être soit des littéraux, soit des expressions qui résolvent
aux scalaires ou aux paramètres d'entrée.

La requête suivante appelle une fonction de base de données nommée shouldGetBonus. L'ID de
le département du salarié et les projets sur lesquels il travaille sont passés en paramètres et
le type de retour de la fonction est un booléen. Le résultat crée une condition qui rend la requête
retourne l'ensemble de tous les employés qui reçoivent une prime.

```
CHOISIR DISTINCT e
FROM Employé e REJOINDRE e.projects p
FONCTION WHERE ('shouldGetBonus', e.dept.id, p.id)
```

Astuce, le mot-clé FUNCTION a été introduit dans Jpa 2.1.

Expressions de cas

L'expression de cas JP QL est une adaptation de l'expression de cas ANSI SQL-92, prenant
en tenant compte des capacités du langage JP QL. Les expressions de cas sont des outils puissants
pour introduire une logique conditionnelle dans une requête, avec l'avantage que le résultat d'un cas
expression peut être utilisée partout où une expression scalaire est valide.

Les expressions de cas sont disponibles sous quatre formes, en fonction de la flexibilité requise par
la requête. La première forme et la plus flexible est l'expression de cas général. Tout autre cas
Les types d'expression peuvent être composés en fonction de l'expression de casse générale. Il a la
forme suivante:

```
CASE {WHEN <cond_expr> THEN <scalar_expr>} + ELSE <scalar_expr> END
```

Le cœur de l'expression case est la clause WHEN, dont il doit y avoir au moins
un. Le processeur de requêtes résout l'expression conditionnelle de chaque clause WHEN dans
commande jusqu'à ce qu'il en trouve une qui réussit. Il évalue ensuite l'expression scalaire pour cela
WHEN et la renvoie comme résultat de l'expression case. Si aucune des clauses WHEN
les expressions conditionnelles donnent un vrai résultat, l'expression scalaire de la clause ELSE
est évalué et renvoyé à la place. L'exemple suivant montre le général

Épisode 367

expression de cas, énumérant le nom et le type de chaque projet qui a des employés
qui lui est assigné:

```
SELECT p. Nom,
        CASE WHEN TYPE (p) = DesignProject ALORS 'Développement'
             WHEN TYPE (p) = QualityProject ALORS 'QA'
             ELSE 'Non-développement'
        FIN
FROM Projet p
O les employés ne sont pas vides
```

Notez l'utilisation de l'expression case dans le cadre de la clause select. Les expressions de cas sont un outil puissant pour transformer les données d'entité dans les requêtes de rapport.

Une légère variation de l'expression de cas général est l'expression de cas simple. Au lieu de vérifier une expression conditionnelle dans chaque clause WHEN, il identifie une valeur et résout une expression scalaire dans chaque clause WHEN. Le premier à correspondre aux déclencheurs de valeur une deuxième expression scalaire qui devient la valeur de l'expression case. Il a la forme suivante:

```
CAS <valeur> {WHEN <scalar_expr1> THEN <scalar_expr2>} +
ELSE <expr_scalaire> END
```

La <valeur> sous cette forme d'expression est soit une expression de chemin conduisant à un champ d'état ou une expression de type pour une comparaison polymorphe. On peut simplifier le dernier exemple en le convertissant en une simple expression de cas.

```
SELECT p. Nom,
        TYPE DE CAS (p)
             QUAND DesignProject PUIS 'Développement'
             QUAND QualityProject ALORS «QA»
             ELSE 'Non-développement'
        FIN
FROM Projet p
O les employés ne sont pas vides
```

La troisième forme de l'expression case est l'expression coalesce. Cette forme de l'expression case accepte une séquence d'une ou plusieurs expressions scalaires. Il a la forme suivante:

```
COALESCE (<expr_scalaire> {, <expr_scalaire>} +)
```


Épisode 368

Les expressions scalaires de l'expression COALESCE sont résolues dans l'ordre. Le premier pour renvoyer une valeur non nulle devient le résultat de l'expression. L'exemple suivant démontre cet usage en renvoyant soit le nom descriptif de chaque département, soit l'identifiant du service si aucun nom n'a été défini:

```
SELECT COALESCE (d.name, d.id)
DU Département d
```

La quatrième et dernière forme d'expression de cas est quelque peu inhabituelle. Il accepte deux expressions scalaires et résout les deux. Si les résultats des deux expressions sont égal, le résultat de l'expression est nul. Sinon, il renvoie le résultat du premier scalaire

expression. Cette forme de l'expression de cas est identifiée par le mot clé NULLIF.

NULLIF (<expr_scalaire1>, <expr_scalaire2>)

Une astuce utile avec NULLIF consiste à exclure les résultats d'une fonction d'agrégation. Pour Par exemple, la requête suivante renvoie un décompte de tous les départements et un décompte de tous départements non nommés «QA»:

```
SELECT COUNT (*), COUNT (NULLIF (d.name, 'QA'))
DU Département d
```

Si le nom du département est 'QA', NULLIF retournera NULL, qui sera alors ignoré par la fonction COUNT. Les fonctions d'agrégation ignorent les valeurs NULL et sont décrites plus loin dans la section «Requêtes agrégées».

ORDER BY Clause

Les requêtes peuvent éventuellement être triées en utilisant ORDER BY et une ou plusieurs expressions consistant en de variables d'identification, de variables de résultat, d'une expression de chemin résolue en une seule entité, ou une expression de chemin se résolvant en un champ d'état persistant. Les mots-clés facultatifs ASC ou DESC après l'expression peut être utilisé pour indiquer des tris croissant ou décroissant, respectivement. L'ordre de tri par défaut est croissant.

L'exemple suivant illustre le tri par un seul champ:

```
SÉLECTIONNER e
DE Employé e
ORDER BY e.name DESC
```

Épisode 369

Plusieurs expressions peuvent également être utilisées pour affiner l'ordre de tri:

```
SÉLECTIONNER e, d
FROM Employé e REJOINDRE e.département d
ORDER BY d.name, e.name DESC
```

Une variable de résultat peut être déclarée dans la clause SELECT dans le but de spécifier un article à commander. Une variable de résultat est en fait un alias pour sa sélection attribuée article. Cela évite à la clause ORDER BY d'avoir à dupliquer des expressions de chemin à partir du SELECT et permet de référencer des éléments de sélection calculés et des éléments qui utilisent fonctions d'agrégation. La requête suivante définit deux variables de résultat dans le SELECT puis les utilise pour classer les résultats dans la clause ORDER BY:

```
SELECT e.name, e.salary * 0,05 AS bonus, d.name AS deptName
FROM Employé e REJOINDRE e.département d
ORDER BY DeptName, bonus DESC
```

Si la clause SELECT de la requête utilise des expressions de chemin de champ d'état, la commande ORDER BY est limitée aux mêmes expressions de chemin que celles utilisées dans la clause SELECT. Par exemple, la requête suivante n'est pas légale:

```
SELECT e.name
DE Employé e
COMMANDER PAR e.salary DESC
```

Étant donné que le type de résultat de la requête est le nom de l'employé, qui est de type String, le reste des champs d'état de l'employé ne sont plus disponibles pour la commande.

Requêtes agrégées

Une requête agrégée est une variante d'une requête de sélection normale. Un groupe de requêtes agrégées résultats et applique des fonctions d'agrégation pour obtenir des informations récapitulatives sur la requête résultats. Une requête est considérée comme une requête agrégée si elle utilise une fonction d'agrégation ou possède une clause GROUP BY et / ou une clause HAVING. La forme la plus typique d'agrégat la requête implique l'utilisation d'une ou plusieurs expressions de regroupement et de fonctions d'agrégation dans la clause SELECT associée à des expressions de regroupement dans la clause GROUP BY. La syntaxe de une requête agrégée est la suivante:

Épisode 370


```
SELECT <expression_sélectionner>
FROM <clause_from>
[WHERE <expression_conditionnelle>]
[GROUP BY <group_by_clause>]
[HAVING <expression_conditionnelle>]
[ORDER BY <order_by_clause>]
```

Les clauses SELECT, FROM et WHERE se comportent sensiblement de la même manière que précédemment sous les requêtes de sélection, à l'exception de certaines restrictions sur la façon dont la clause SELECT est formulé.

La puissance d'une requête agrégée provient de l'utilisation de fonctions d'agrégation sur données groupées. Prenons l'exemple d'agrégat simple suivant:

```
SELECT AVG (e.salary)
DE Employé e
```

Cette requête renvoie le salaire moyen de tous les employés de l'entreprise. AVG est un fonction d'agrégation qui prend une expression de chemin de champ d'état numérique comme argument et calcule la moyenne sur le groupe. Comme il n'y avait pas de clause GROUP BY spécifiée, le groupe ici est l'ensemble des employés.

Considérons maintenant cette variante, où le résultat a été groupé par département Nom:

```
SELECT d.name, AVG (e.salary)
DU Département d REJOINDRE d. Employés e
GROUP BY d.nom
```

Cette requête renvoie le nom de chaque département et le salaire moyen du employés de ce service. L'entité Département est jointe à l'Employé l'entité à travers la relation des employés et ensuite formé dans un groupe défini par le Nom du département. La fonction AVG calcule ensuite son résultat en fonction de l'employé données dans ce groupe.

Cela peut être étendu pour filtrer les données afin que les salaires des managers ne soient pas inclus:

```
SELECT d.name, AVG (e.salary)
DU Département d REJOINDRE d. Employés e
O e.directs EST VIDE
GROUP BY d.nom
```


Épisode 371

Enfin, nous pouvons prolonger cette dernière fois pour ne renvoyer que les départements où le le salaire moyen est supérieur à 50 000 $. Considérez la version suivante du précédent requete:

```
SELECT d.name, AVG (e.salary)
DU Département d REJOINDRE d. Employés e
O e.directs EST VIDE
GROUP BY d.nom
HAVING AVG (e.salary)> 50000
```

Pour mieux comprendre cette requête, passons en revue les étapes logiques qui ont eu lieu pour l'exécuter. Les bases de données utilisent de nombreuses techniques pour optimiser ces types de requêtes, mais conceptuellement, le même processus est suivi.

Tout d'abord, la requête non groupante suivante est exécutée:

```
SELECT d.nom, e.salaire
DU Département d REJOINDRE d. Employés e
O e.directs EST VIDE
```

Cela produira un ensemble de résultats comprenant tous les noms de service et la valeur de salaire paires. Le moteur de requête démarre alors un nouvel ensemble de résultats et effectue un deuxième passage sur le données, collecter toutes les valeurs salariales pour chaque nom de service et les transmettre à la fonction AVG. Cette fonction renvoie la moyenne du groupe, qui est ensuite comparée les critères de la clause HAVING. Si la valeur moyenne est supérieure à 50 000 USD, la requête Le moteur génère une ligne de résultats composée du nom du service et de la valeur salariale moyenne.

Les sections suivantes décrivent les fonctions d'agrégation disponibles pour une utilisation dans l'agrégat requêtes et l'utilisation des clauses GROUP BY et HAVING.

Fonctions d'agrégation

Cinq fonctions d'agrégation peuvent être placées dans la clause select d'une requête: AVG, COUNT, MAX, MIN et SUM.

AVG

La fonction AVG prend une expression de chemin de champ d'état comme argument et calcule le valeur moyenne de ce champ d'état sur le groupe. Le type de champ d'état doit être numérique et le résultat est renvoyé sous forme de Double.

Épisode 372

COMPTER

La fonction COUNT prend une variable d'identification ou une expression de chemin comme son argument. Cette expression de chemin peut se résoudre en un champ d'état ou une association à valeur unique champ. Le résultat de la fonction est une valeur Long représentant le nombre de valeurs dans le groupe. L'argument de la fonction COUNT peut éventuellement être précédé du mot clé DISTINCT, auquel cas les valeurs en double sont éliminées avant le comptage.

La requête suivante compte le nombre de téléphones associés à chaque employé comme ainsi que le nombre de types de numéros distincts (cellule, bureau, domicile, etc.):

```
SELECT e, COUNT (p), COUNT (DISTINCT p. Type)
FROM Employé e JOIN e.phones p
GROUPE PAR e
```

MAX

La fonction MAX prend une expression de champ d'état comme argument et renvoie le maximum valeur dans le groupe pour ce champ d'état.

MIN

La fonction MIN prend une expression de champ d'état comme argument et renvoie le minimum valeur dans le groupe pour ce champ d'état.

SOMME

La fonction SOMME prend une expression de champ d'état comme argument et calcule la somme de les valeurs de ce champ d'état sur le groupe. Le type de champ d'état doit être numérique et le le type de résultat doit correspondre au type de champ. Par exemple, si un champ Double est additionné, le résultat sera renvoyé en tant que Double. Si un champ Long est additionné, la réponse sera renvoyé comme un Long.

Clause GROUP BY

La clause GROUP BY définit les expressions de regroupement sur lesquelles les résultats seront agrégé. Une expression de regroupement doit être une expression de chemin à valeur unique (champ d'état, intégrable menant à un champ d'état ou champ d'association à valeur unique) ou

Épisode 373

variable d'identification. Si une variable d'identification est utilisée, l'entité ne doit avoir aucun état sérialisé ou champs d'objets volumineux.

La requête suivante compte le nombre d'employés dans chaque service:

```
SELECT d.nom, COUNT (e)
DU Département d REJOINDRE d. Employés e
GROUP BY d.nom
```

Notez que la même expression de champ utilisée dans la clause SELECT est répétée dans le Clause GROUP BY. Toutes les expressions non agrégées doivent être répertoriées de cette façon.

Plusieurs fonctions d'agrégation peuvent être appliquées:

```
SELECT d.name, COUNT (e), AVG (e.salary)
DU Département d REJOINDRE d. Employés e
GROUP BY d.nom
```

Cette variante de la requête calcule le salaire moyen de tous les employés de chaque département en plus de compter le nombre d'employés dans le département.

Plusieurs expressions de regroupement peuvent également être utilisées pour décomposer davantage les résultats:

```
SELECT d.nom, e.salaire, COUNT (p)
DU Département d REJOINDRE d. Employés e REJOINDRE e.projets p
GROUP BY d.name, e.salary
```

Les deux expressions de regroupement, le nom du service et le salaire de l'employé, doivent être répertorié à la fois dans la clause SELECT et dans la clause GROUP BY. Pour chaque département, cette requête compte le nombre de projets attribués aux employés en fonction de leur salaire.

En l'absence de clause GROUP BY, les fonctions d'agrégation seront appliquées à l'ensemble ensemble de résultats en tant que groupe unique. Par exemple, la requête suivante renvoie le nombre de salariés et leur salaire moyen dans toute l'entreprise:

```
SELECT COUNT (e), AVG (e.salary)
DE Employé e
```

Clause HAVING

La clause HAVING définit un filtre à appliquer une fois que les résultats de la requête ont été groupé. Il s'agit en fait d'une clause WHERE secondaire, et sa définition est la même: mot-clé HAVING suivi d'une expression conditionnelle. La principale différence avec le

Épisode 374

HAVING est que ses expressions conditionnelles sont principalement limitées aux champs d'état ou champs d'association à valeur unique inclus dans le groupe.

Les expressions conditionnelles de la clause HAVING peuvent également utiliser des fonctions d'agrégation sur les éléments utilisés pour le regroupement ou les fonctions d'agrégation qui apparaissent dans la clause SELECT. À bien des égards, l'utilisation principale de la clause HAVING est de restreindre les sur les valeurs de résultat agrégées. La requête suivante utilise cette technique pour récupérer tous employés affectés à deux ou plusieurs projets:

```
SELECT e, COUNT (p)
FROM Employé e REJOINDRE e.projects p
GROUPE PAR e
COMPTE (p)> = 2
```

Mettre à jour les requêtes

Les requêtes de mise à jour fournissent un équivalent à l'instruction SQL UPDATE mais avec JP QL expressions conditionnelles. La forme d'une requête de mise à jour est la suivante:

```
UPDATE <nom de l'entité> [[AS] <variable d'identification>]
SET <update_statement> {, <update_statement>} *
[WHERE <expression_conditionnelle>]
```

Chaque instruction UPDATE se compose d'une expression de chemin à valeur unique, l'affectation opérateur (=) et une expression. Les choix d'expression pour l'instruction d'affectation sont légèrement restreint par rapport aux expressions conditionnelles régulières. Le côté droit de la l'affectation doit être résolue en une expression littérale simple résolvant en un type de base, fonction expression, variable d'identification ou paramètre d'entrée. Le type de résultat de cette expression doit être compatible avec le chemin d'association simple ou le champ d'état persistant à gauche côté de la mission.

L'exemple simple suivant illustre la requête de mise à jour en donnant aux employés qui gagnent 55000 $ par année une augmentation de 60000 $:

```
MISE À JOUR Employé e
SET e.salary = 60000
O e.salaire = 55000
```


Épisode 375

La clause WHERE d'une instruction UPDATE fonctionne de la même manière qu'une instruction SELECT et peut utiliser la variable d'identification définie dans la clause UPDATE dans les expressions. UNE

une requête de mise à jour légèrement plus complexe mais plus réaliste consisterait à attribuer une augmentation de 5000 $ aux employés ayant travaillé sur un projet particulier:

```
MISE À JOUR Employé e
SET e.salary = e.salary + 5000
O EXISTE (SELECT p
                DE e.projects p
                WHERE p.name = 'Release2')
```

Plusieurs propriétés de l'entité cible peuvent être modifiées avec une seule UPDATE déclaration. Par exemple, la requête suivante met à jour le central téléphonique pour les employés dans la ville d'Ottawa et change la terminologie du type de téléphone de Bureau à Entreprise:

```
METTRE À JOUR le téléphone p
SET p.number = CONCAT ('288', SUBSTRING (p.number, LOCATE ('-', p.number),
4)),
    p.type = 'Entreprise'
O p.employee.address.city = 'Ottawa' ET
      p.type = 'Bureau'
```

Supprimer les requêtes

La requête de suppression offre la même fonctionnalité que l'instruction SQL DELETE, mais avec JP Expressions conditionnelles QL. La forme d'une requête de suppression est la suivante:

```
DELETE FROM <nom de l'entité> [[AS] <variable d'identification>]
[O <condition>]
```

L'exemple suivant supprime tous les employés qui ne sont pas affectés à un service:

```
SUPPRIMER DE L'employé e
O e.department EST NULL
```


Épisode 376

La clause WHERE pour une instruction DELETE fonctionne de la même manière que pour un SELECT déclaration. Toutes les expressions conditionnelles sont disponibles pour filtrer l'ensemble des entités à supprimé. Si la clause WHERE n'est pas fournie, toutes les entités du type donné sont supprimées.

Les requêtes de suppression sont polymorphes. Toute instance de sous-classe d'entité qui répond aux critères de la requête de suppression sera également supprimée. Les requêtes de suppression ne respectent pas les règles en cascade, pourtant. Aucune entité autre que le type référencé dans la requête et ses sous-classes ne être supprimée, même si l'entité a des relations avec d'autres entités avec suppression en cascade activée.

Résumé

Dans ce chapitre, nous vous avons présenté une visite complète du langage Java Persistence Query, en regardant les nombreux types de requêtes et leur syntaxe. Nous avons couvert l'histoire de la langue, de ses racines dans la spécification EJB aux principales améliorations introduites par JPA.

Dans la section sur les requêtes sélectionnées, nous avons exploré chaque clause de requête et de manière incrémentielle construit des requêtes plus complexes au fur et à mesure de la description de la syntaxe complète. Nous avons discuté

les variables d'identification et les expressions de chemin, qui sont utilisées pour naviguer dans
modèle de domaine dans les expressions de requête. Nous avons parlé des jointures et des différentes manières de
déclenchant la jonction intérieure et extérieure. Nous avons étudié les sous-requêtes et comment elles s'intègrent dans le
langage de requête, ainsi que la façon d'interroger les arbres d'héritage d'entités. Nous avons aussi
regardé les différentes expressions conditionnelles et scalaires prises en charge par le langage.

Dans notre discussion sur les requêtes agrégées, nous avons introduit le regroupement supplémentaire et
clauses de filtrage qui étendent les requêtes de sélection. Nous avons également démontré les différents agrégats
les fonctions.

Dans les sections sur les requêtes de mise à jour et de suppression, nous avons décrit la syntaxe complète des requêtes en bloc
instructions de mise à jour et de suppression, dont le comportement d'exécution a été décrit dans le précédent
chapitre.

Dans le chapitre suivant, nous poursuivons notre exploration des fonctionnalités de requête JPA avec une
examen approfondi de l'API Criteria, une API d'exécution pour la construction de requêtes.

Épisode 377

CHAPITRE 9

API de critères

Dans le dernier chapitre, nous avons examiné en détail le langage de requête JP QL et les concepts
qui sous-tendent le modèle de requête JPA. Dans ce chapitre, nous examinons une autre méthode pour
la construction de requêtes utilisant une API de langage de programmation Java au lieu de JP QL ou
SQL natif.

Nous commençons par un aperçu de l'API JPA Criteria et examinons un cas d'utilisation courant
impliquant la construction de requêtes dynamiques dans une application d'entreprise. Ceci est suivi de
une exploration approfondie de l'API Criteria et de son lien avec JP QL.

Une fonctionnalité connexe de l'API Criteria est l'API du métamodèle JPA. Nous concluons ceci
chapitre avec un aperçu de l'API du métamodèle et voir comment il peut être utilisé pour créer
requêtes fortement typées utilisant l'API Criteria.

Notez que ce chapitre suppose que vous avez lu le chapitre 8 et connaissent
tous les concepts et la terminologie qu'il introduit. Dans la mesure du possible, nous utilisons le
des mots clés JP QL en majuscules pour mettre en évidence différents éléments du modèle de requête JPA et
démontrer leur comportement équivalent avec l'API Criteria.

Aperçu

Avant que des langages comme JP QL ne deviennent standardisés, la méthode la plus courante pour
la construction de requêtes dans de nombreux fournisseurs de persistance se faisait via une programmation
API. L'infrastructure de requête d'EclipseLink, par exemple, était le moyen le plus efficace de
débloquez vraiment toute la puissance de son moteur de requête. Et, même avec l'avènement de JP QL,
les API de programmation sont toujours utilisées pour donner accès à des fonctionnalités non encore prises en charge
par le langage de requête standard.

JPA 2.0 a introduit une API Criteria pour la construction de requêtes qui standardise de nombreux
les fonctionnalités de programmation qui existent dans les produits de persistance propriétaires. Plus que
une traduction littérale de JP QL en interface de programmation, il adopte également la meilleure programmation

pratiques des modèles propriétaires, tels que l'enchaînement de méthodes, et utilise pleinement les
Fonctionnalités du langage de programmation Java.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_9

Épisode 378

Les sections suivantes fournissent une vue de haut niveau de l'API Criteria, expliquant comment
et quand il est approprié de l'utiliser. Nous examinons également un exemple plus significatif avec une utilisation
cas commun dans de nombreux environnements d'entreprise.

L'API Criteria

Commençons par un exemple simple pour démontrer la syntaxe et l'utilisation des critères
API. La requête JP QL suivante renvoie tous les employés de l'entreprise avec le nom
de «John Smith:»

```
SÉLECTIONNER e
DE Employé e
WHERE e.name = 'John Smith'
```

Et voici la requête équivalente construite à l'aide de l'API Criteria:

```
CriteriaBuilder cb = em.getCriteriaBuilder ();
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
c.select (emp)
.where (cb.equal (emp.get ("nom"), "John Smith"));
```

Il se passe beaucoup de choses dans quelques lignes de code dans cet exemple, mais tout de suite vous
devrait voir des parallèles entre la version JP QL et la version basée sur des critères. le
Les mots clés JP QL SELECT, FROM, WHERE et LIKE ont des méthodes de correspondance sous la forme de
select (), from (), where (), and like (). La classe d'entité Employee remplace le
nom d'entité lors de l'invocation de from (), et l'attribut name de Employee est toujours en cours
accessible, mais au lieu de l'opérateur point JP QL, nous avons ici la méthode get ().

Au fur et à mesure que nous progressons dans ce chapitre, nous explorons chacune de ces méthodes en détail, mais pour
maintenant, nous regardons la situation dans son ensemble. Tout d'abord, il y a l'interface CriteriaBuilder, obtenue
ici depuis l'interface EntityManager via la méthode getCriteriaBuilder (). le
L'interface CriteriaBuilder est la principale passerelle vers l'API Criteria, agissant comme une usine pour
les différents objets qui se lient pour former une définition de requête. La variable cb est utilisée dans
les exemples de ce chapitre pour représenter l'objet CriteriaBuilder.

La première utilisation de l'interface CriteriaBuilder dans cet exemple consiste à créer une instance
de CriteriaQuery. L'objet CriteriaQuery forme le shell de la définition de requête et
contient généralement les méthodes qui correspondent aux clauses de requête JP QL. La deuxième

Épisode 379

l'utilisation de l'interface CriteriaBuilder dans cet exemple consiste à construire le conditionnel
expressions dans la clause WHERE. Tous les mots-clés, opérateurs et
les fonctions de JP QL sont représentées d'une manière ou d'une autre sur l'interface CriteriaBuilder.

Dans ce contexte, il est facile de voir comment la requête s'articule. Le premier pas
consiste à établir la racine de la requête en invoquant from () pour récupérer un objet Root. C'est

équivalent à déclarer la variable d'identification e dans l'exemple JP QL et la racine
objet constituera la base des expressions de chemin dans le reste de la requête. L'étape suivante
établit la clause SELECT de la requête en passant la racine dans la méthode select ().
La dernière étape consiste à construire la clause WHERE, en passant une expression composée de
Méthodes CriteriaBuilder qui représentent des expressions de condition JP QL dans le where ()
méthode. Lorsque des expressions de chemin sont nécessaires, telles que l'accès à l'attribut de nom dans ce
Par exemple, la méthode get () sur l'objet Root est utilisée pour créer le chemin.

Types paramétrés

L'exemple précédent a démontré l'utilisation de génériques Java dans l'API Criteria. le
L'API utilise largement les types paramétrés: presque toutes les interfaces et méthodes
La déclaration utilise des génériques Java sous une forme ou une autre. Les génériques permettent au compilateur de
détecte de nombreux cas d'utilisation de type incompatible et, comme l'API de collecte Java, supprime
le besoin de coulée dans la plupart des cas.

Toute API utilisant des génériques Java peut également être utilisée sans paramètres de type, mais lorsque
compilé le code émettra des avertissements du compilateur. Par exemple, du code qui utilise un simple
Untyped ("raw") List type générera un avertissement indiquant qu'une référence à un
le type générique doit être paramétré. La ligne de code suivante générera deux de ces
avertissements, un pour utiliser List et un pour utiliser ArrayList:

```
Liste liste = new ArrayList ();
```

L'API Criteria peut être utilisée de la même manière sans lier les objets de critères à des
types, bien que cela élimine clairement les avantages de la frappe. L'exemple précédent pourrait être
réécrit comme:

```
CriteriaBuilder cb = em.getCriteriaBuilder ();
CriteriaQuery c = cb.createQuery (Employee.class);
Root emp = c. De (Employee.class);
c.select (emp)
  .where (cb.equal (emp.get ("nom"), "John Smith"));
```


Épisode 380

Ce code est fonctionnellement identique à l'exemple d'origine, mais se trouve juste être
plus sujet aux erreurs lors du développement. Néanmoins, certaines personnes peuvent être disposées
d'avoir moins de sécurité de type au moment du développement mais plus de lisibilité du code en l'absence de
"Type clutter." Ceci est particulièrement compréhensible étant donné que la plupart des gens courent à
tests minimaux sur une requête avant d'envoyer le code de la requête, et une fois que vous arrivez au
point de savoir que la requête fonctionne, alors vous êtes déjà aussi loin que vous le seriez
avec vérification de type à la compilation.

Si vous êtes dans la catégorie des personnes qui préféreraient que le code soit plus simple à lire
et développer au prix d'une sécurité un peu moins élevée lors de la compilation, alors en fonction de votre
tolérance pour les avertissements du compilateur, vous pouvez les désactiver. Ceci peut être réalisé
en ajoutant une annotation @SuppressWarnings ("non cochée") sur votre classe. cependant,
être averti (sans jeu de mots!) que cela entraînera tous les avertissements de vérification de type
supprimés, pas seulement ceux liés à l'utilisation de l'API Criteria.

Requêtes dynamiques

Pour démontrer une bonne utilisation potentielle de l'API Criteria, nous examinerons une utilisation courante
cas dans de nombreuses applications d'entreprise: création de requêtes dynamiques où la structure
les critères ne sont connus qu'au moment de l'exécution.

Au chapitre 7, nous avons expliqué comment créer des requêtes JP QL dynamiques. Vous construisez
la chaîne de requête au moment de l'exécution, puis transmettez-la à la méthode createQuery () du
Interface EntityManager. Le moteur de requête analyse la chaîne de requête et renvoie une requête

objet que vous pouvez utiliser pour obtenir des résultats. La création de requêtes dynamiques est requise dans les situations où la sortie ou les critères d'une requête varient en fonction des choix de l'utilisateur final.

Considérez une application Web utilisée pour rechercher les informations de contact des employés. Il s'agit d'une fonctionnalité courante dans de nombreuses grandes organisations qui permet aux utilisateurs de rechercher par nom, service, numéro de téléphone ou même lieu, séparément ou en utilisant une combinaison de termes de requête. Référencement9-1 montre un exemple d'implémentation d'un bean session qui accepte un ensemble de critères puis construit et exécute un JP QL requête en fonction des paramètres de critères définis. Il n'utilise pas le API Criteria. Cet exemple et d'autres dans ce chapitre utilisent le modèle de données décrit dans la figure 8-1 du chapitre 8 .

Épisode 381

Liste 9-1. Recherche d'employés à l'aide de la requête JP QL dynamique

```
@Apatride
classe publique SearchService {
    @PersistenceContext (unitName = "EmployeeHR")
    EntityManager em;

    Liste publique <Employee> findEmployees (String name, String deptName,
                                             String projectName, String city) {
        StringBuffer query = new StringBuffer ();
        query.append ("SELECT DISTINCT e");
        query.append ("FROM Employee e LEFT JOIN e.projects p");

        query.append ("WHERE");
        Critères de liste <String> = new ArrayList <String> ();
        if (nom! = null) {critère.add ("e.name =: nom"); }
        if (deptName! = null) {critère.add ("e.dept.name =: dept"); }
        if (projectName! = null) {critère.add ("p.name =: projet"); }
        if (ville! = null) {critère.add ("e.address.city =: ville"); }
        if (critère.size () == 0) {
            lancer une nouvelle RuntimeException ("aucun critère");
        }
        pour (int i = 0; i <critère.size (); i ++) {
            if (i> 0) {query.append ("AND"); }
            query.append (critère.get (i));
        }

        Requête q = em.createQuery (query.toString ());
        if (nom! = null) {q.setParameter ("nom", nom); }
        if (deptName! = null) {q.setParameter ("dept", deptName); }
        if (projectName! = null) {q.setParameter ("projet", projectName); }
        if (ville! = null) {q.setParameter ("ville", ville); }
        return (List <Employee>) q.getResultList ();
    }
}
```

Épisode 382

La méthode findEmployees () dans la liste Le 9-1 doit effectuer un certain nombre de tâches chaque moment où il est invoqué. Il doit construire une chaîne de requête avec un ensemble variable de critères, créer la requête, liez les paramètres, puis exécutez la requête. C'est assez simple implémentation, et fera le travail, mais chaque fois que la chaîne de requête est créée, le le fournisseur doit analyser le JP QL et construire une représentation interne de la requête avant que les paramètres puissent être liés et SQL généré. Ce serait bien si nous pouvions éviter la surcharge d'analyse et construire les différentes options de critères en utilisant l'API Java au lieu de cordes. Considérez le Listing 9-2.

Liste 9-2. Recherche d'employés à l'aide de l'API Criteria

```
@Apatride
classe publique SearchService {
    @PersistenceContext (unitName = "EmployeeHR")
    EntityManager em;

    Liste publique <Employee> findEmployees (String name, String deptName,
                                             String projectName, String city) {

        CriteriaBuilder cb = em.getCriteriaBuilder ();
        CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
        Racine <Employee> emp = c.from (Employee.class);
        c.select (emp);
        c.distinct (vrai);
        Rejoindre le projet <Employé, Projet> =
            emp.join ("projets", JoinType.LEFT);

        Critères de liste <Predicate> = new ArrayList <Predicate> ();
        if (nom! = null) {
            ParameterExpression <String> p =
                cb.parameter (String.class, "nom");
            critères.add (cb.equal (emp.get ("nom"), p));
        }
        if (deptName! = null) {
            ParameterExpression <String> p =
                cb.parameter (String.class, "dept");
            critère.add (cb.equal (emp.get ("dept"). get ("nom"), p));
        }
```


Épisode 383


```
        if (projectName! = null) {
            ParameterExpression <String> p =
                cb.parameter (String.class, "projet");
            critères.add (cb.equal (project.get ("nom"), p));
        }
        if (ville! = null) {
            ParameterExpression <String> p =
                cb.parameter (String.class, "ville");
            critères.add (cb.equal (emp.get ("adresse"). get ("ville"), p));
```

```
        }
        if (critère.size () == 0) {
              lancer une nouvelle RuntimeException ("aucun critère");
        } else if (critère.size () == 1) {
             c.where (critère.get (0));
        } autre {
              c.where (cb.and (critères.toArray (nouveau prédicat [0])));
        }

        TypedQuery <Employee> q = em.createQuery (c);
        if (nom! = null) {q.setParameter ("nom", nom); }
        if (deptName! = null) {q.setParameter ("dept", deptName); }
        if (project! = null) {q.setParameter ("project", projectName); }
        if (ville! = null) {q.setParameter ("ville", ville); }
        return q.getResultList ();
    }
}
```

Référencement 9-2 montre le même service EmployeeSearch de la liste9-1 refait avec l'API Criteria. Ceci est un exemple beaucoup plus large que notre premier aperçu de l'API Criteria, mais encore une fois, vous pouvez voir le modèle général de la façon dont il est construit. La requête de base méthodes de construction et de clause des interfaces CriteriaBuilder et CriteriaQuery sont présents comme avant, mais il y a quelques nouveaux éléments dans cette requête que nous pouvons explorer. le le premier est la jointure entre Employee et Project, ici construite en utilisant la méthode join () sur l'objet racine. L'objet Join renvoyé peut également être utilisé pour construire le chemin expressions telles que l'objet racine. Dans cet exemple, nous pouvons également voir une expression de chemin impliquant plusieurs relations, de l'employé à l'adresse en passant par l'attribut ville.

Épisode 384

Le deuxième élément nouveau de cette requête est l'utilisation de paramètres. Contrairement à JP QL où les paramètres ne sont qu'un alias, dans l'API Criteria, les paramètres sont fortement typés et créé à partir de l'appel de paramètre (). L'objet ParameterExpression renvoyé peut puis être utilisé dans d'autres parties de la requête telles que les clauses WHERE ou SELECT. En terme d'expressions, cet exemple inclut également les méthodes CriteriaBuilder equal () et et (), équivalent aux prédicats JP QL = et AND. Notez l'invocation un peu étrange de la méthode and (). Comme beaucoup d'autres méthodes de l'API Criteria, et () accepte une variable nombre d'arguments, qui à leur tour peuvent être représentés comme un tableau des type d'argument. Malheureusement, les concepteurs de la méthode Collection.toArray () a décidé que, pour éviter de transtyper le type de retour, un tableau à remplir devrait également être passé en argument ou en tableau vide dans le cas où nous voulons collection pour créer le tableau pour nous. La syntaxe de l'exemple est un raccourci pour le code suivant:

```
Prédicat [] p = nouveau prédicat [critère.size ()];
p = critères.toArray (p);
c. où (cb. et (p));
```

La dernière fonctionnalité de cette requête que nous n'avons pas démontrée précédemment est l'exécution de la requête elle-même. Comme nous l'avons démontré au chapitre 7 , l'interface TypedQuery est utilisée pour obtenir des résultats de requête fortement typés. Définitions de requête créées avec l'API Criteria avoir leur type de résultat lié à l'aide de génériques Java et donc toujours générer une TypedQuery objet de la méthode createQuery () de l'interface EntityManager.

Création de requêtes API de critères

Notre examen de haut niveau des exemples d'API Criteria a conclu, les sections suivantes
examinez en détail chaque aspect de la création d'une définition de requête. Dans la mesure du possible, nous essayons de
mettre en évidence la similitude entre les concepts et la terminologie JP QL et Criteria API.

Création d'une définition de requête

Comme nous l'avons démontré dans les sections précédentes, le cœur de l'API Criteria est le
Interface CriteriaBuilder, obtenue à partir de l'interface EntityManager en appelant le
méthode getCriteriaBuilder (). L'interface CriteriaBuilder est volumineuse et sert

Épisode 385

plusieurs objectifs dans l'API Criteria. C'est une usine avec laquelle nous créons la requête
définition elle-même, une instance de l'interface CriteriaQuery, ainsi que de nombreux
les composants de la définition de requête tels que les expressions conditionnelles.

L'interface CriteriaBuilder propose trois méthodes pour créer une nouvelle sélection
définition de la requête, en fonction du type de résultat souhaité de la requête. Le premier et
la méthode la plus courante est la méthode createQuery (Class <T>), passant dans la classe
correspondant au résultat de la requête. C'est l'approche que nous avons utilisée dans la liste9-2.
La deuxième méthode est createQuery (), sans aucun paramètre, et correspond à un
requête avec un type de résultat Object. La troisième méthode, createTupleQuery (), est utilisée pour
requêtes de projection ou de rapport où la clause SELECT de la requête contient plus de
une expression et vous souhaitez travailler avec le résultat d'une manière plus fortement typée. Il
est en réalité juste une méthode pratique qui équivaut à appeler createQuery (Tuple.
classe). Notez que Tuple est une interface qui contient un assortiment d'objets ou de données
et applique le typage aux pièces agrégées. Il peut être utilisé chaque fois que plusieurs éléments sont
renvoyés et vous souhaitez les combiner en un seul objet typé.

Il est à noter que, malgré le nom, une instance CriteriaQuery n'est pas une requête
objet qui peut être appelé pour obtenir des résultats de la base de données. C'est une définition de requête qui
peut être passé à la méthode createQuery () de l'interface EntityManager à la place de
une chaîne JP QL. La seule vraie différence entre une définition de requête de critères et un JP QL
string est la méthode de construction de la définition de requête (API de programmation par rapport au texte) et
que les requêtes de critères sont généralement typées, de sorte que le type de résultat n'a pas à être spécifié
lors de l'appel de createQuery () sur EntityManager afin d'obtenir un TypedQuery
exemple. Vous pouvez également trouver utile de penser à une instance CriteriaQuery entièrement définie comme
étant similaire à la représentation interne d'une requête JP QL qu'un fournisseur de persistance
peut être utilisé après l'analyse de la chaîne JP QL.

L'API Criteria est composée d'un certain nombre d'interfaces qui fonctionnent ensemble pour modéliser
la structure d'une requête JPA. Au fur et à mesure de votre progression dans ce chapitre, vous le trouverez peut-être utile
pour faire référence aux relations d'interface illustrées à la Figure 9-1 .

Figure 9-1. *Interfaces API de critères*

Structure basique

Dans la discussion de JP QL au chapitre 8 , vous avez appris qu'il existe six clauses possibles pour être utilisé dans une requête de sélection: SELECT, FROM, WHERE, ORDER BY, GROUP BY et HAVING. Chacun des ces clauses JP QL ont une méthode équivalente sur l'une des définitions de requête de l'API Criteria les interfaces. Le tableau 9-1 résume ces méthodes.

Table 9-1. *Critères API Sélectionnez les méthodes de clause de requête*

Clause JP QL	Interface API de critères	Méthode
SÉLECTIONNER	Critères Requête	sélectionner()
	Sous-requête	sélectionner()
DE	RésuméQuery	de()
OÙ	RésuméQuery	où()
COMMANDÉ PAR	Critères Requête	commandé par()
PAR GROUPE	RésuméQuery	par groupe()
AYANT	RésuméQuery	ayant()

Comme nous l'avons démontré, il existe une forte symétrie entre le langage JP QL et les méthodes de l'API Criteria. Dans la mesure du possible, le même nom a été utilisé, ce qui il est facile d'anticiper le nom d'une méthode API Criteria, même si vous ne l'avez pas utilisée avant. Au cours des prochaines sections, nous examinons chacune de ces méthodes de clause en détail et comment les expressions sont formées à l'aide de l'API Criteria.

Objets critères et mutabilité

L'utilisation typique de l'API Criteria entraînera la création de nombreux objets différents. Dans en plus des objets CriteriaBuilder et CriteriaQuery principaux, chaque composant de chaque expression est représentée par un objet ou un autre. Tous les objets ne sont pas créés Cependant, l'utilisation égale et efficace de l'API Criteria nécessite une connaissance du codage modèles assumés dans sa conception.

Le premier problème que nous devons considérer est celui de la mutabilité. La majorité des objets créés via l'API Criteria sont en fait immuables. Il n'y a pas de méthodes de setter ou des méthodes de mutation sur ces interfaces. Presque tous les objets créés à partir du les méthodes de l'interface CriteriaBuilder appartiennent à cette catégorie.

Épisode 388

L'utilisation d'objets immuables signifie que les arguments passés dans le Les méthodes CriteriaBuilder sont riches en détails. Toutes les informations pertinentes doivent être transmises en ce que l'objet peut être complet au moment de sa création. L'avantage de cela l'approche est qu'elle facilite les invocations chaînées de méthodes. Parce que pas de mutation les méthodes doivent être appelées sur les objets renvoyés par les méthodes utilisées pour construire expressions, le contrôle peut passer immédiatement au composant suivant de l'expression.

Seules les méthodes CriteriaBuilder qui créent des objets de définition de requête produisent résultats vraiment mutables. Les objets CriteriaQuery et Subquery sont destinés à être modifié plusieurs fois en appelant des méthodes telles que select (), from () et where (). Mais même ici, il faut faire attention car invoquer deux fois des méthodes peut avoir l'un des deux effets. Dans la plupart des cas, l'invocation d'une méthode deux fois remplace le contenu de l'objet avec l'argument fourni. Par exemple, appeler select () deux fois avec deux des arguments différents ne donnent en fait que l'argument de la deuxième invocation restant dans le cadre de la définition de la requête.

Dans certains cas, cependant, invoquer une méthode deux fois est en fait un ajout. Appel depuis () deux fois avec des arguments différents entraîne l'ajout de plusieurs racines de requête à la requête définition. Bien que nous nous référions à ces cas dans les sections où ils sont décrits, vous devez Familiarisez-vous avec les commentaires Javadoc sur l'API Criteria, car ils signalent également ce comportement.

Le deuxième problème est la présence de méthodes getter sur les objets de l'API Criteria. Celles-ci se comportent comme prévu, renvoyant des informations sur le composant de la requête que chaque objet représente. Mais il convient de noter que ces méthodes intéressent principalement les développeurs d'outils qui souhaitent travailler avec des définitions de requête de manière vraiment générique. dans le dans la grande majorité des cas, et ceux que nous démontrons dans ce chapitre, vous n'aurez pas à utilisez les méthodes getter dans la construction de vos requêtes de critères.

Racines de requête et expressions de chemin

Un objet CriteriaQuery nouvellement créé est essentiellement un shell vide. À l'exception de définir le type de résultat de la requête, aucun contenu supplémentaire n'a encore été ajouté

remplissez la requête. Comme pour les requêtes JP QL, le développeur est responsable de la définition du diverses clauses de la requête nécessaires pour extraire les données souhaitées de la base de données. Sémantiquement parlant, il n'y a pas de différence entre JP QL et la requête de l'API Criteria définitions. Les deux ont des clauses SELECT, FROM, WHERE, GROUP BY, HAVING et ORDER; seulement la manière de les définir est différente. Avant de pouvoir remplir les différentes clauses du définition de requête, revenons d'abord sur deux concepts clés définis au chapitre 8 et regardez le syntaxe d'API Criteria équivalente pour ces concepts.

Épisode 389

Racines de requête

Le premier concept fondamental à revoir est la variable d'identification utilisée dans le Clause FROM des requêtes JP QL aux déclarations d'alias qui couvrent l'entité, l'intégration et autres types de schémas abstraits. Dans JP QL, la variable d'identification prend un importance, car c'est la clé pour lier les différentes clauses de la requête. Mais avec l'API Criteria, nous représentons des composants de requête avec des objets et donc ont rarement des pseudonymes dont nous devons nous préoccuper. Pourtant, afin de définir un FROM clause, nous avons besoin d'un moyen d'exprimer les types de schémas abstraits qui nous intéressent interroger contre.

L'interface AbstractQuery (parent de CriteriaQuery) fournit le from () pour définir le type de schéma abstrait qui constituera la base de la requête. Cette accepte un type d'entité comme paramètre et ajoute une nouvelle *racine* à la requête. Une racine dans une requête de critères correspond à une variable d'identification dans JP QL, qui à son tour correspond à une déclaration de variable de plage ou à une expression de jointure. Dans le Listing 9-2 , nous avons utilisé le code suivant pour obtenir notre racine de requête:

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
```

La méthode from () renvoie une instance de Root correspondant au type d'entité. L'interface racine est elle-même étendue à partir de l'interface From, qui expose les fonctionnalités pour les jointures. L'interface From étend Path, qui étend encore plus Expression, puis Sélection, permettant à la racine d'être utilisée dans d'autres parties de la définition de requête. Le rôle de chacune de ces interfaces est décrite dans les sections suivantes. Les appels à la méthode from () sont additif. Chaque appel ajoute une autre racine à la requête, résultant en un produit cartésien lorsque plus d'une racine est définie si aucune autre contrainte n'est appliquée dans la clause WHERE. L'exemple suivant du chapitre 8 illustre plusieurs racines de requête, remplaçant un jointure conventionnelle avec l'approche SQL plus traditionnelle:

```
CHOISIR DISTINCT d
DE Département d, employé e
WHERE d = e.département
```


Épisode 390

Pour convertir cette requête en l'API Criteria, nous devons appeler de () deux fois, en ajoutant les entités Department et Employee en tant que racines de requête. L'exemple suivant démontre cette approche:

```
CriteriaQuery <Department> c = cb.createQuery (Department.class);
Racine <Department> dept = c.from (Department.class);
Racine <Employee> emp = c.from (Employee.class);
c.select (dept)
  .distinct (vrai)
  .where (cb.equal (dept, emp.get ("département")));
```

Expressions de chemin

Le deuxième concept fondamental à revoir est l'expression du chemin. L'expression du chemin est la clé de la puissance et de la flexibilité du langage JP QL, et c'est également un élément de l'API Criteria. Nous avons discuté des expressions de chemin en détail dans le chapitre8 donc si vous pensez que vous avez besoin d'un rappel, nous vous recommandons de revenir en revue cette section.

Nous avons examiné les racines des requêtes dans la section précédente, et les racines ne sont en fait qu'un type d'expression de chemin. Requête racines en main, nous pouvons maintenant regarder comment obtenir et étendre les expressions de chemin. Considérez la requête de base JP QL suivante, qui renvoie tous les employés résidant à New York:

```
SÉLECTIONNER e
DE Employé e
WHERE e.address.city = 'New York'
```

En termes d'API Criteria, la racine de requête de cette expression est le Entité des employés. Cette requête contient également une expression de chemin dans la clause WHERE. À représenter cette expression de chemin à l'aide de l'API Criteria, nous utiliserions ce qui suit expression:

```
emp.get ("adresse"). get ("ville")
```

L'objet emp dans cet exemple correspond à la racine de la requête pour Employee. le get () est dérivée de l'interface Path étendue par l'interface Root et est équivalent à l'opérateur point utilisé dans les expressions de chemin JP QL pour naviguer le long d'un chemin. Étant donné que la méthode get () renvoie un objet Path, les appels de méthode peuvent être chaînés ensemble, en évitant la déclaration inutile de variables locales intermédiaires.

L'argument de get () est le nom de l'attribut qui nous intéresse. Parce que le Le résultat de la construction d'une expression de chemin est un objet Expression que nous pouvons utiliser pour construire expressions conditionnelles, nous pouvons alors exprimer la requête complète comme suit:

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
c.select (emp)
  .where (cb.equal (emp.get ("adresse"). get ("ville"), "New York"));
```

Tout comme JP QL, les expressions de chemin peuvent être utilisées dans les différentes clauses de la définition de la requête. Avec l'API Criteria, il est nécessaire de conserver l'objet racine dans une variable locale et utilisez-la pour former des expressions de chemin si nécessaire. Encore une fois c'est il convient de souligner que la méthode from () de AbstractQuery ne doit jamais être invoquée plus d'une fois pour chaque racine souhaitée. L'appeler plusieurs fois entraînera des racines en cours de création et un produit cartésien sinon prudent. Stockez toujours les objets racine localement et s'y référer si nécessaire.

La clause SELECT

La clause SELECT d'une requête peut prendre plusieurs formes. La forme la plus simple implique une seule expression, tandis que d'autres impliquent plusieurs expressions ou l'utilisation d'un expression de constructeur pour créer de nouvelles instances d'objet. Chaque forme est exprimée différemment dans l'API Criteria.

Sélection d'expressions uniques

La méthode select () de l'interface CriteriaQuery est utilisée pour former la clause SELECT dans une définition de requête de l'API Criteria. Toutes les formes de la clause SELECT peuvent être représentées via la méthode select (), bien que des méthodes pratiques existent également pour simplifier le codage. le select () nécessite un argument de type Selection, une interface étendue par Expression, ainsi que CompoundSelection pour gérer le cas où le type de résultat d'un query est un Tuple ou un tableau de résultats.

> Remarque Certains fournisseurs peuvent autoriser l'omission de l'appel à select () dans le cas où il existe une seule racine de requête et correspond au type de résultat déclaré pour le requete. il s'agit d'un comportement non portable.

Épisode 392

Jusqu'à présent, nous avons passé une racine de requête à la méthode select (), donc indiquant que nous voulons que l'entité soit le résultat de la requête. Nous pourrions également fournir un expression à valeur unique telle que la sélection d'un attribut à partir d'une entité ou de tout autre compatible expression scalaire. L'exemple suivant illustre cette approche en sélectionnant le attribut name de l'entité Employee:

```
CriteriaQuery <String> c = cb.createQuery (String.class);
Racine <Employee> emp = c.from (Employee.class);
c.select (emp. <String> get ("nom"));
```

Cette requête renverra tous les noms d'employés, y compris les doublons. Dupliquer les résultats d'une requête peut être supprimée en appelant distinct (true) de AbstractQuery interface. Ce comportement est identique au mot clé DISTINCT dans une requête JP QL.

Notez également la syntaxe inhabituelle que nous avons utilisée pour déclarer que l'attribut «nom» était de type String. Le type de l'expression fournie à la méthode select () doit être compatible avec le type de résultat utilisé pour créer l'objet CriteriaQuery. Par exemple, si l'objet CriteriaQuery a été créé en appelant createQuery (Project.class) sur l'interface CriteriaBuilder, alors ce sera une erreur d'essayer de définir une expression résolution de l'entité Employé en utilisant la méthode select (). Quand une méthode appelle une telle comme select () utilise un typage générique pour appliquer une contrainte de compatibilité, le type peut être préfixé au nom de la méthode afin de le qualifier dans les cas où le type pourrait pas autrement être déterminé automatiquement. Nous devons utiliser cette approche dans ce cas car la méthode select () a été déclarée comme suit:

```
CriteriaQuery <T> select (Selection <? Extend T> selection);
```

L'argument de select () doit être un type compatible avec le type de résultat de la définition de la requête. La méthode get () renvoie un objet Path, mais cet objet Path est toujours de type Path <Object> car le compilateur ne peut pas déduire le type correct basé sur le nom de l'attribut. Pour déclarer que l'attribut est vraiment de type String, nous devons qualifiez l'invocation de méthode en conséquence. Cette syntaxe doit être utilisée chaque fois que le Le chemin est passé comme argument pour lequel le paramètre a été fortement typé, comme l'argument de la méthode select () et certaines expressions CriteriaBuilder méthodes. Nous n'avons pas eu à les utiliser jusqu'à présent dans nos exemples car nous avons été les utiliser dans des méthodes comme equal (), où le paramètre a été déclaré comme étant de type

Expression <?>. Étant donné que le type est générique, il est valide de passer un argument de type
Chemin <Objet>. Plus loin dans le chapitre, nous examinons les versions fortement typées des critères
Méthodes API qui suppriment cette exigence.

Épisode 393

Sélection de plusieurs expressions

Lors de la définition d'une clause SELECT qui implique plusieurs expressions, l'API Criteria
L'approche requise dépend de la manière dont la définition de requête a été créée. Si le type de résultat
est Tuple, puis un objet CompoundSelection <Tuple> doit être passé à select (). Si la
le type de résultat est une classe non persistante qui sera créée à l'aide d'une expression de constructeur,
alors l'argument doit être un objet CompoundSelection <[T]>, où [T] est la classe
type de la classe non persistante. Enfin, si le type de résultat est un tableau d'objets, alors un
L'objet CompoundSelection <Object []> doit être fourni. Ces objets sont créés avec
les méthodes tuple (), construct () et array () de l'interface CriteriaBuilder,
respectivement. L'exemple suivant montre comment fournir plusieurs expressions
à une requête Tuple:

```
CriteriaQuery <Tuple> c = cb.createTupleQuery ();
Racine <Employee> emp = c.from (Employee.class);
c.select (cb.tuple (emp.get ("id"), emp.get ("nom")));
```

Pour plus de commodité, la méthode multiselect () de l'interface CriteriaQuery
peut également être utilisé pour définir la clause SELECT. La méthode multiselect () créera le
type d'argument approprié compte tenu du type de résultat de la requête. Cela peut prendre trois formes
selon la manière dont la définition de requête a été créée.

Le premier formulaire est destiné aux requêtes dont le type de résultat est Object ou Object []. La liste
des expressions qui composent chaque résultat sont simplement passées à la méthode multiselect ().

```
CriteriaQuery <Object []> c = cb.createQuery (Object []. Class);
Racine <Employee> emp = c.from (Employee.class);
c.multiselect (emp.get ("id"), emp.get ("nom"));
```

Notez que si le type de résultat de la requête est déclaré comme Object au lieu de Object [], le
le comportement de multiselect () sous cette forme change légèrement. Le résultat est toujours une instance
of Object, mais si plusieurs arguments sont passés dans multiselect (), le résultat doit
être casté en Object [] afin d'accéder à une valeur particulière. Si un seul argument est
passé dans multiselect (), aucun tableau n'est créé et le résultat peut être converti directement
de Object au type souhaité. En général, il est plus pratique d'être explicite sur le
type de résultat de la requête. Si vous souhaitez travailler avec un tableau de résultats, alors déclarer la requête
le type de résultat Object [] évite de lancer plus tard et rend la forme du résultat plus
explicite si la requête est appelée séparément du code qui la crée.

Épisode 394

Le deuxième formulaire est un parent proche du premier formulaire, mais pour les requêtes qui aboutissent à
Tuple. Encore une fois, la liste des expressions est transmise à l'appel multiselect ().

```
CriteriaQuery <Tuple> c = cb.createTupleQuery ();
Racine <Employee> emp = c.from (Employee.class);
```

```
c.multiselect (emp.get ("id"), emp.get ("nom"));
```

La troisième et dernière forme est pour les requêtes avec des expressions de constructeur qui résultent dans les types non persistants. La méthode multiselect () est à nouveau appelée avec une liste de expressions, mais il utilise le type de la requête pour comprendre et créer automatiquement l'expression de constructeur appropriée, dans ce cas un objet de transfert de données de type EmployeeInfo.

```
CriteriaQuery <EmployeeInfo> c = cb.createQuery (EmployeeInfo.class);
Racine <Employee> emp = c.from (Employee.class);
c.multiselect (emp.get ("id"), emp.get ("nom"));
```

Ceci est équivalent à ce qui suit:

```
CriteriaQuery <EmployeeInfo> c = cb.createQuery (EmployeeInfo.class);
Racine <Employee> emp = c.from (Employee.class);
c.select (cb.construct (EmployeeInfo.class,
                        emp.get ("id"),
                        emp.get ("nom")));
```

Aussi pratique que la méthode multiselect () soit pour les expressions de constructeur, il y a encore des cas où vous devrez utiliser la méthode construct () de la Interface CriteriaBuilder. Par exemple, si le type de résultat de la requête est Object [] et qu'il inclut également une expression de constructeur pour une partie seulement des résultats, ce qui suit serait obligatoire:

```
CriteriaQuery <Object []> c = cb.createQuery (Object []. Class);
Racine <Employee> emp = c.from (Employee.class);
c.multiselect (emp.get ("id"),
               cb.construct (EmployeeInfo.class,
                             emp.get ("id"),
                             emp.get ("nom")));
```

Épisode 395

Utilisation d'alias

Comme JP QL, les alias peuvent également être définis sur des expressions dans la clause SELECT, qui être inclus dans l'instruction SQL résultante. Ils sont peu utiles d'une programmation perspective lorsque nous construisons la clause ORDER BY via l'utilisation de la sélection objets utilisés pour construire la clause SELECT.

Les alias sont utiles lorsque la requête a un type de résultat Tuple. Les alias seront disponible via les objets Tuple résultants. Pour définir un alias, la méthode alias () du L'interface de sélection (parent à Expression) doit être appelée. L'exemple suivant démontre cette approche:

```
CriteriaQuery <Tuple> c = cb.createTupleQuery ();
Racine <Employee> emp = c.from (Employee.class);
c.multiselect (emp.get ("id"). alias ("id"), emp.get ("name").
alias ("fullName"));
```

Cet exemple montre en fait deux facettes de la méthode alias (). Le premier est qu'il retourne lui-même, de sorte qu'il peut être appelé dans le cadre de l'appel à select () ou multiselect (). La seconde est, encore une fois, qu'il se retourne, et mute donc ce qui devrait être un objet autrement immuable. La méthode alias () est une exception à la règle selon laquelle seules les interfaces de définition de requête (CriteriaQuery et Subquery) contiennent des mutations opérations. L'appel d'alias () modifie l'objet Selection d'origine et le renvoie depuis

l'invocation de la méthode. Il est incorrect de définir plusieurs fois l'alias d'un objet Selection.

L'utilisation de l'alias lors de l'itération sur les résultats de la requête est aussi simple que de demander le expression par nom. L'exécution de la requête précédente lui permettrait d'être traitée comme suit:

```
TypedQuery <Tuple> q = em.createQuery (c);
pour (Tuple t: q.getResultList ()) {
    String id = t.get ("id", String.class);
    String fullName = t.get ("fullName", String.class);
    // ...
}
```


Épisode 396

La clause FROM

Dans la section «Query Roots», nous avons couvert la méthode from () de AbstractQuery interface et le rôle des racines de requête dans la formation de la définition de requête. Nous élaborons maintenant sur cette discussion et regardez comment les jointures sont exprimées à l'aide de l'API Criteria.

Jointures intérieure et extérieure

Les expressions de jointure sont créées à l'aide de la méthode join () de l'interface From, qui est étendu à la fois par Root, que nous avons abordé précédemment, et Join, qui est le type d'objet renvoyé en créant des expressions de jointure. Cela signifie que toute racine de requête peut se joindre, et qui se joint peuvent s'enchaîner les uns avec les autres. La méthode join () nécessite une expression de chemin argument et éventuellement un argument pour spécifier le type de jointure, JoinType.INNER ou JoinType.LEFT, respectivement pour les jointures internes et externes.

> Tip la valeur énumérée JoinType.RIGHT spécifie qu'une jointure externe droite devrait être appliqué. La prise en charge de cette option n'est pas requise par la spécification, donc les applications qui l'utilisent ne seront pas portables.

Lors de la jonction à travers un type de collection (sauf pour Map, dont nous parlerons plus loin dans ce chapitre), la jointure aura deux types paramétrés: le type de la source et le type de la cible. Cela maintient la sécurité du type des deux côtés de la jonction et le rend clair quels types sont joints.

La méthode join () est additive, donc chaque appel entraîne la création d'une nouvelle jointure; par conséquent, l'instance Join renvoyée par l'appel de la méthode doit être conservée dans une variable locale pour former des expressions de chemin plus tard. Parce que Join étend également le chemin, il se comporte comme des objets racine lors de la définition des chemins.

Dans la liste 9-2 , nous avons démontré une jointure externe de l'employé au projet.

```
Rejoindre <Employé, Projet> project = emp.join ("projets", JoinType.LEFT);
```

Si l'argument JoinType.LEFT avait été omis, le type de jointure aurait été défini par défaut être une jointure intérieure. Tout comme dans JP QL, plusieurs jointures peuvent être associées au même De l'instance. Par exemple, pour naviguer dans la relation directe entre l'employé et

Épisode 397

puis aux entités du Ministère et du Projet exigeraient ce qui suit, qui
suppose une jonction intérieure:

Rejoindre <Employé, Employé> dirige = emp.join ("dirige");
Rejoignez <Employé, Projet> projects = directs.join ("projects");
Rejoignez <Employé, Département> dept = directs.join ("dept");

Les jointures peuvent également être mises en cascade dans une seule instruction. La jointure résultante sera tapée par la source et la cible de la dernière jointure dans l'instruction:

Rejoindre <Employé, Projet> project = dept.join ("employés"). Join ("projects");

Les jointures entre les relations de collection qui utilisent Map sont un cas particulier. JP QL utilise le Mots clés KEY et VALUE pour extraire la clé ou la valeur d'un élément Map à utiliser dans d'autres parties de la requête. Dans l'API Criteria, ces opérateurs sont gérés par la clé () et la valeur () méthodes de l'interface MapJoin. Prenons l'exemple suivant en supposant une jointure de carte dans la relation téléphonique de l'entité Employé:

SELECT e.name, KEY (p), VALUE (p)
FROM Employé e JOIN e.phones p

Pour créer cette requête à l'aide de l'API Criteria, nous devons capturer le résultat de la jointure comme un MapJoin, dans ce cas en utilisant la méthode joinMap (). L'objet MapJoin a trois types paramètres: le type de source, le type de clé et le type de valeur. Cela peut paraître un peu plus intimidant, mais rend explicite les types impliqués.

CriteriaQuery <Objet> c = cb.createQuery ();
Racine <Employee> emp = c.from (Employee.class);
MapJoin <Employé, Chaîne, Téléphone> phone = emp.joinMap ("téléphones");
c.multiselect (emp.get ("nom"), phone.key (), phone.value ());

Nous devons utiliser la méthode joinMap () dans ce cas car il n'y a aucun moyen de surcharger la méthode join () pour renvoyer un objet Join ou un objet MapJoin quand tout ce que nous sommes passer est le nom de l'attribut. Les relations Collection, Set et List sont de même manipulé avec les méthodes joinCollection (), joinSet () et joinList () pour les cas où une interface de jointure spécifique doit être utilisée. La version fortement typée de la méthode join (), que nous démontrerons plus tard, est capable de gérer tous les types de jointure l'appel unique join ().

Épisode 398

Récupérer les jointures

Comme avec JP QL, l'API Criteria prend en charge la jointure fetch, une construction de requête qui permet données à pré-extraire dans le contexte de persistance en tant qu'effet secondaire d'une requête qui renvoie une entité différente, mais liée. L'API Criteria crée des jointures d'extraction grâce à l'utilisation de la méthode fetch () sur l'interface FetchParent. Il est utilisé à la place de join () dans les cas où la sémantique de récupération est requise et accepte les mêmes types d'arguments.

Prenons l'exemple suivant que nous avons utilisé dans le chapitre précédent pour démontrer

extraire les jointures de relations à valeur unique:

```
SÉLECTIONNER e
FROM Employé e JOIN FETCH e.address
```

Pour recréer cette requête avec l'API Criteria, nous utilisons la méthode fetch ().

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
emp.fetch ("adresse");
c.select (emp);
```

Notez que lorsque vous utilisez la méthode fetch (), le type de retour est Fetch, pas Join. Récupérer les objets ne sont pas des chemins et ne peuvent être étendus ou référencés nulle part ailleurs dans la requête.

Les jointures d'extraction à valeur de collection sont également prises en charge et utilisent une syntaxe similaire. Dans ce qui suit exemple, nous montrons comment récupérer les entités Phone associées à chaque employé, en utilisant une jointure externe pour empêcher les entités Employee d'être ignorées si elles n'en ont pas entités téléphoniques associées. Nous utilisons le paramètre distinct () pour supprimer tous les doublons.

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
emp.fetch ("téléphones", JoinType.LEFT);
c.select (emp)
  .distinct (vrai);
```

La clause WHERE

Comme vous l'avez vu dans le tableau 9-1 et dans plusieurs exemples, la clause WHERE d'une requête dans le L'API Criteria est définie via la méthode where () de l'interface AbstractQuery. La méthode where () accepte zéro ou plusieurs objets Predicate, ou un seul

Épisode 399

Argument de l'expression <Boolean>. Chaque appel à where () rendra tout précédemment défini WHERE expressions à rejeter et à remplacer par les expressions nouvellement transmises.

Voici comment la clause WHERE est définie:

```
where_clause :: = WHERE expression_conditionnelle
```

Construire des expressions

La clé pour créer des expressions avec l'API Criteria est le CriteriaBuilder interface. Cette interface contient des méthodes pour tous les prédicats, expressions et fonctions supportées par le langage JP QL ainsi que d'autres fonctionnalités spécifiques au API Criteria. Le tableau 9-2, le tableau 9-3, le tableau 9-4 et le tableau 9-5 résument le mappage entre les opérateurs, expressions et fonctions JP QL à leurs méthodes équivalentes sur l'interface CriteriaBuilder. Notez que dans certains cas, il n'y a pas d'égalité directe à un et une combinaison de méthodes CriteriaBuilder est nécessaire pour obtenir la même résultat. Dans d'autres cas, la méthode des critères équivalents est en fait sur une classe autre que CriteriaBuilder.

Tableau 9-2. *Mappage de prédicat JP QL vers CriteriaBuilder*

Opérateur JP QL	CriteriaBuilder, méthode
ET	et()
OU	ou()
NE PAS	ne pas()

=	égal()
<>	inégal()
>	supérieur à (), gt ()
> =	GreaterThanOrEqualTo (), ge ()
<	lessThan (), lt ()
<=	lessThanOrEqualTo (), le ()
ENTRE	entre()
EST NULL	isNull ()

(*suite*)

Épisode 400

Tableau 9-2. (*suite*)

Opérateur JP QL	CriteriaBuilder, méthode
EST NON NULLE	est non nulle()
EXISTE	existe ()
N'EXISTE PAS	n'existe pas())
EST VIDE	est vide()
N'EST PAS VIDE	n'est pas vide()
MEMBRE DE	isMember ()
PAS MEMBRE DE	isNotMember ()
COMME	comme()
PAS COMME	pas comme()
DANS	dans()
PAS DEDANS	pas dedans())

Tableau 9-3. *Mappage d'expressions scalaires JP QL vers CriteriaBuilder*

Expression JP QL	CriteriaBuilder, méthode
TOUT	tout()
TOUT	tout()
CERTAINS	certains()
-	nég (), diff ()
+	somme()
*	prod ()
/	quot ()
SE FONDRE	se fondre()
NULLIF	nullif ()
CAS	selectCase ()

Épisode 401

Tableau 9-4. *Mappage de fonctions JP QL vers CriteriaBuilder*

Fonction JP QL	CriteriaBuilder, méthode
abdos	abdos()
CONCAT	concat ()
DATE ACTUELLE	date actuelle()
HEURE ACTUELLE	heure actuelle()
CURRENT_TIMESTAMP	currentTimestamp ()
LONGUEUR	longueur()
LOCALISER	Localiser()
INFÉRIEUR	inférieur()
MOD	mod ()
TAILLE	Taille()
SQRT	sqrt ()
SUBSTRING	sous-chaîne ()
PLUS HAUT	plus haut()
RÉDUIRE	réduire()

Tableau 9-5. *Mappage des fonctions d'agrégation JP QL vers CriteriaBuilder*

Fonction d'agrégation JP QL	CriteriaBuilder, méthode
AVG	moy ()
SOMME	somme (), sumAsLong (), sumAsDouble ()
MIN	min (), moins ()
MAX	max (), le plus grand ()
COMPTER	compter()
COUNT DISTINCT	countDistinct ()

Épisode 402

En plus de la traduction directe des opérateurs, expressions et fonctions JP QL, certaines techniques spécifiques à l'API Criteria doivent être prises en compte lorsque développer des expressions. Les sections suivantes examinent ces techniques en détail et explorez les parties de l'API Criteria qui n'ont pas d'équivalent dans JP QL.

Prédicats

Dans la liste 9-2 , nous avons passé un tableau d'objets Predicate à la méthode and (). Cela a le comportement de combinaison de toutes les expressions avec l'opérateur AND. Comportement équivalent car l'opérateur OR existe via la méthode or (). Un raccourci qui fonctionne pour les opérateurs AND consiste à passer toutes les expressions comme arguments à la méthode where (). Passer plusieurs arguments à where () combine implicitement les expressions en utilisant la sémantique de l'opérateur AND.

L'API Criteria propose également un style différent de création d'expressions AND et OR pour ceux qui souhaitent construire les choses progressivement plutôt que sous forme de liste. La conjonction () et les méthodes disjunction () de l'interface CriteriaBuilder créent des objets Predicate qui résolvent toujours respectivement vrai et faux. Une fois obtenus, ces primitifs les prédicats peuvent ensuite être combinés avec d'autres prédicats pour créer des conditions conditionnelles imbriquées expressions à la manière d'un arbre. Le Listing 9-3 réécrit la construction de la prédication partie de l'exemple de la liste 9-2 en utilisant la méthode conjonction (). Notez comment chaque instruction conditionnelle est combinée avec son prédécesseur à l'aide d'un appel and ().

Liste 9-3. Construction de prédicat à l'aide de la conjonction

```
Critères de prédicat = cb.conjunction ();
if (nom! = null) {
      ParameterExpression <String> p =
            cb.parameter (String.class, "nom");
      critères = cb.and (critères, cb.equal (emp.get ("nom"), p));
}
if (deptName! = null) {
      ParameterExpression <String> p =
            cb.parameter (String.class, "dept");
      critères = cb.et (critères,
                              cb.equal (emp.get ("dept"). get ("name"), p));
}
```

Épisode 403


```
if (projectName! = null) {
      ParameterExpression <String> p =
            cb.parameter (String.class, "projet");
      critères = cb.and (critères, cb.equal (project.get ("nom"), p));
}
if (ville! = null) {
      ParameterExpression <String> p =
            cb.parameter (String.class, "ville");
      critères = cb.et (critères,
                              cb.equal (emp.get ("adresse"). get ("ville"), p));
}

if (critère.getExpressions (). taille () == 0) {
      lancer une nouvelle RuntimeException ("aucun critère");
}
```

En ce qui concerne les autres préoccupations liées aux prédicats, dans le tableau 9-2, il convient de noter que il existe deux ensembles de méthodes disponibles pour les comparaisons relatives. Par exemple, il y a GreaterThan () et gt (). Les formes à deux lettres sont spécifiques aux valeurs numériques et sont fortement typé pour travailler avec les types de nombres. Les formulaires longs doivent être utilisés pour tous les autres cas.

Littéraux

Les valeurs littérales peuvent nécessiter un traitement spécial lorsqu'elles sont exprimées avec l'API Criteria. Dans tout les cas rencontrés jusqu'à présent, les méthodes sont surchargées pour fonctionner avec les deux Expression objets et littéraux Java. Cependant, il peut y avoir des cas où seule une expression object est accepté (dans les cas où il est supposé que vous ne passeriez jamais une valeur littérale ou lorsque l'un quelconque d'un certain nombre de types serait acceptable). Si vous rencontrez cette situation

puis, pour utiliser ces expressions avec des littéraux Java, les littéraux doivent être encapsulés à l'aide de méthode literal (). Les littéraux NULL sont créés à partir de la méthode nullLiteral (), qui accepte un paramètre de classe et produit une version typée de NULL pour correspondre au passé classe. Cela est nécessaire pour étendre le typage fort de l'API aux valeurs NULL.

Épisode 404

Paramètres

La gestion des paramètres pour les requêtes d'API Criteria est différente de JP QL. Alors que les chaînes JP QL préfixez simplement les noms de chaîne par deux points pour désigner un alias de paramètre, cette technique ne fonctionne pas dans l'API Criteria. Au lieu de cela, nous devons créer explicitement une ParameterExpression de le type correct qui peut être utilisé dans les expressions conditionnelles. Ceci est réalisé grâce au paramètre () de l'interface CriteriaBuilder. Cette méthode nécessite un type de classe (pour définir le type de l'objet ParameterExpression) et un nom facultatif à utiliser avec paramètres nommés. Référencement9-4 illustre cette méthode.

Liste 9-4. Création d'expressions de paramètres

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
c.select (emp);
ParameterExpression <String> deptName =
        cb.parameter (String.class, "deptName");
c.where (cb.equal (emp.get ("dept"). get ("name"), deptName));
```

Si le paramètre ne sera pas réutilisé dans d'autres parties de la requête, il peut être intégré directement dans l'expression de prédicat pour rendre la définition globale de la requête plus concise. Le code suivant révise le Listing 9-4 pour utiliser cette technique:

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
c.select (emp)
  .where (cb.equal (emp.get ("dept"). get ("nom"),
                          cb.parameter (String.class, "deptName")));
```

Sous-requêtes

L'interface AbstractQuery fournit la méthode subquery () pour la création de sous-requêtes. Les sous-requêtes peuvent être corrélées (ce qui signifie qu'elles référencent une racine, un chemin ou join à partir de la requête parente) ou non corrélé. L'API Criteria prend en charge les deux et des sous-requêtes non corrélées, en utilisant à nouveau des racines de requête pour lier les différentes clauses et expressions ensemble. L'argument de subquery () est une instance de classe représentant le type de résultat de la sous-requête. La valeur de retour est une instance de Subquery, qui est elle-même un extension de AbstractQuery. À l'exception des méthodes restreintes de construction

Épisode 405

clauses, l'instance Subquery est une définition de requête complète comme CriteriaQuery qui peut être utilisé pour créer des requêtes simples et complexes.

Pour illustrer l'utilisation des sous-requêtes, examinons un exemple plus significatif, en modifiant Référencement 9-2 pour utiliser des sous-requêtes au lieu de la méthode distinct () pour éliminer les doublons. Selon le modèle de données illustré à la figure 8-1 , l'entité Employé a des relations avec quatre autres entités: relations à valeur unique avec le service et l'adresse, et relations valorisées par la collection avec Phone et Project. Chaque fois que nous nous joignons à travers un relation de valeur de collection, nous avons le potentiel de renvoyer des lignes en double; donc, nous devons changer l'expression des critères pour que Project utilise une sous-requête. Référencement9-5 montre le fragment de code requis pour effectuer cette modification.

Liste 9-5. Recherche d'employés modifiée avec des sous-requêtes

```
@Apatride
classe publique SearchService {
    @PersistenceContext (unitName = "EmployeeHR")
    EntityManager em;

    Liste publique <Employee> findEmployees (String name, String deptName,
                                                String projectName, String city) {
        CriteriaBuilder cb = em.getCriteriaBuilder ();
        CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
        Racine <Employee> emp = c.from (Employee.class);
        c.select (emp);

        // ...

        if (projectName! = null) {
            Sous-requête <Employee> sq = c.subquery (Employee.class);
            Projet <Projet> racine = sq.from (Project.class);
            Rejoindre <Projet, Employé> sqEmp = project.join ("employés");
            sq.select (sqEmp)
              .where (cb.equal (project.get ("nom"),
                                cb.parameter (String.class, "project")));
            critère.add (cb.in (emp) .valeur (sq));
        }

        // ...
}
```

Épisode 406

Référencement 9-5 contient quelques changements importants par rapport à l'exemple présenté en premier dans le listing 9-2 . Tout d'abord, l'appel de la méthode distinct () a été supprimé ainsi que la jointure à l'entité Projet. Nous avons également introduit une nouvelle sous-requête non corrélée par rapport à Project. Parce que la sous-requête de Listing 9-5 déclare sa propre racine et ne fait pas référence quoi que ce soit de la requête parente, il s'exécute indépendamment et n'est donc pas corrélé. La requête JP QL équivalente avec uniquement des critères de projet serait:

```
SÉLECTIONNER e
DE Employé e
WHERE e IN (SELECT emp
            FROM Projet p REJOINDRE p. Employés emp
            WHERE p.name =: projet)
```

Chaque fois que nous écrivons des requêtes qui utilisent des sous-requêtes, il existe souvent plusieurs façons de atteindre un résultat particulier. Par exemple, nous pourrions réécrire l'exemple précédent pour utiliser EXISTS au lieu de IN et déplacez l'expression conditionnelle dans la clause WHERE du

sous-requête.

```
if (projectName! = null) {
        Sous-requête <Projet> sq = c.subquery (Project.class);
        Projet <Projet> racine = sq.from (Project.class);
        Rejoindre <Projet, Employé> sqEmp = project.join ("employés");
        sq.select (projet)
            .where (cb.equal (sqEmp, emp),
                        cb.equal (project.get ("nom"),
                                    cb.parameter (String.class, "project")));
        critères.add (cb.exists (sq));
}
```

En référençant la racine Employee à partir de la requête parent dans la clause WHERE de la sous-requête, nous avons maintenant une sous-requête corrélée. Cette fois, la requête prend les éléments suivants formulaire en JP QL:

```
SÉLECTIONNER e
DE Employé e
O EXISTE (SELECT p
                    FROM Projet p REJOINDRE p. Employés emp
                    WHERE emp = e AND p.name =: nom)
```

Épisode 407

Nous pouvons encore pousser cet exemple plus loin et réduire l'espace de recherche pour la sous-requête en déplaçant la référence à la racine Employee vers la clause FROM de la sous-requête et rejoindre directement la liste des projets spécifiques à cet employé. Dans JP QL, nous écririons ceci comme suit:

```
SÉLECTIONNER e
DE Employé e
O EXISTE (SELECT p
                    DE e.projects p
                    WHERE p.name =: nom)
```

Afin de recréer cette requête à l'aide de l'API Criteria, nous sommes confrontés à un dilemme. Nous devons baser la requête sur l'objet racine de la requête parent mais le from () n'accepte qu'un type de classe persistant. La solution est le corrélat () méthode à partir de l'interface de sous-requête. Il remplit une fonction similaire à from () méthode de l'interface AbstractQuery, mais le fait avec les objets Root et Join du requête parent. L'exemple suivant montre comment utiliser correlate () dans ce cas:

```
if (projectName! = null) {
        Sous-requête <Projet> sq = c.subquery (Project.class);
        Racine <Employé> sqEmp = sq.correlate (emp);
        Rejoignez <Employé, Projet> project = sqEmp.join ("projets");
        sq.select (projet)
            .where (cb.equal (project.get ("nom"),
                                cb.parameter (String.class, "project")));
        critères.add (cb.exists (sq));
}
```

Avant de laisser les sous-requêtes dans l'API Criteria, il y a un autre cas de coin avec sous-requêtes corrélées à explorer: référencement d'une expression de jointure à partir de la requête parente dans la clause FROM d'une sous-requête. Prenons l'exemple suivant qui renvoie des projets contenant des gestionnaires avec des subordonnés directs gagnant un salaire moyen plus élevé qu'un utilisateur-seuil défini:

```
SÉLECTIONNER p
FROM Projet p REJOINDRE p. Employés e
WHERE TYPE (p) = DesignProject ET
        e.directs N'EST PAS VIDE ET
```


Épisode 408


```
    (SELECT AVG (d. Salaire)
      FROM e.directs d)> =: valeur
```

Lors de la création de la définition de requête de l'API Criteria pour cette requête, nous devons mettre en corrélation les Attribut employés de Project, puis joignez-le aux subordonnés directs afin de calculer le salaire moyen. Cet exemple illustre également l'utilisation de la méthode type () du Interface de chemin pour faire une comparaison polymorphe des types:

```
CriteriaQuery <Projet> c = cb.createQuery (Project.class);
Racine <Projet> project = c.from (Project.class);
Rejoindre <Projet, Employé> emp = project.join ("employés");
Sous-requête <Number> sq = c.subquery (Number.class);
Rejoignez <Projet, Employé> sqEmp = sq.correlate (emp);
Rejoindre <Employé, Employé> dirige = sqEmp.join ("dirige");
c.select (projet)
  .where (cb.equal (project.type (), DesignProject.class),
          cb.isNotEmpty (emp. <Collection> get ("dirige")),
          cb.ge (sq.select (cb.avg (directs.get ("salaire"))),
                  cb.parameter (Number.class, "value")));
```

IN Expressions

Contrairement aux autres opérateurs, l'opérateur IN nécessite un traitement spécial dans les critères API. La méthode in () de l'interface CriteriaBuilder n'accepte qu'un seul argument, expression à valeur unique qui sera testée par rapport aux valeurs de l'expression IN. Dans Afin de définir les valeurs de l'expression IN, nous devons utiliser l'objet CriteriaBuilder.In renvoyé par la méthode in (). Considérez la requête JP QL suivante:

```
SÉLECTIONNER e
DE Employé e
WHERE e.address.state IN ('NY', 'CA')
```

Pour convertir cette requête en l'API Criteria, nous devons appeler la méthode value () de la CriteriaBuilder.In interface pour définir les identifiants d'état que nous souhaitons interroger, ainsi:

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
```


Épisode 409


```
c.select (emp)
  .where (cb.in (emp.get ("adresse")
                  .get ("état")). valeur ("NY"). valeur ("CA"));
```

Notez l'invocation chaînée de la méthode value () afin de définir plusieurs valeurs dans l'expression IN. L'argument de in () est l'expression à rechercher par rapport au liste des valeurs fournies via la méthode value ().

Dans les cas où il y a un grand nombre d'appels value () à enchaîner qui sont tous du même type, l'interface Expression propose un raccourci pour créer des expressions IN. Les méthodes in () de cette interface permettent de définir une ou plusieurs valeurs en un seul appel.

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
c.select (emp)
  .where (emp.get ("adresse")
                .get ("état"). in ("NY", "CA"));
```

Dans ce cas, l'appel à in () est suffixé à l'expression plutôt que préfixé comme c'était le cas dans l'exemple précédent. Notez la différence de type d'argument entre les Versions d'interface CriteriaBuilder et Expression de in (). La version Expression of in () accepte les valeurs à rechercher, pas l'expression à rechercher. Le dans () La méthode de l'interface CriteriaBuilder permet plus d'options de saisie, mais pour la plupart c'est en grande partie un cas de préférence personnelle au moment de décider de l'approche à utiliser.

Les expressions IN qui utilisent des sous-requêtes sont écrites en utilisant une approche similaire. Pour plus exemple complexe, dans le chapitre précédent, nous avons présenté une requête JP QL en utilisant un IN expression dans laquelle le service d'un employé est testé par rapport à une liste générée à partir d'une sous-requête. L'exemple est reproduit ici.

```
SÉLECTIONNER e
DE Employé e
O e.department DANS
   (SELECT DISTINCT d
    FROM Département d REJOINDRE d.employés de REJOINDRE de.projet p
    WHERE p.name LIKE 'QA%')
```

Nous pouvons convertir cet exemple en l'API Criteria, comme indiqué dans le Listing 9-6 .

Annonce 9-6. Expression IN utilisant une sous-requête

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
Sous-requête <Department> sq = c.subquery (Department.class);
Racine <Department> dept = sq.from (Department.class);
Rejoindre le projet <Employé, Projet> =
      dept.join ("employés"). join ("projets");
sq.select (dept. <Integer> get ("id"))
   .distinct (vrai)
   .where (cb.like (projet. <String> get ("nom"), "QA%"));
c.select (emp)
  .where (cb.in (emp.get ("dept"). get ("id")). value (sq));
```

La sous-requête est créée séparément puis transmise à la méthode value () en tant que expression pour rechercher l'entité Department. Cet exemple illustre également l'utilisation d'un expression d'attribut en tant que valeur dans la liste de recherche.

Expressions de cas

Comme l'expression IN, la création d'expressions CASE avec l'API Criteria nécessite l'utilisation

d'une interface d'assistance. Dans cet exemple, nous convertissons les exemples utilisés dans le chapitre 8 en Criteria API, démontrant des expressions de cas générales et simples, ainsi que COALESCE.

Astuce, bien que les instructions CASE soient requises par les fournisseurs Jpa, elles peuvent ne pas être pris en charge par toutes les bases de données. l'utilisation d'une instruction CASE sur une plateforme de base de données qui ne prend pas en charge les expressions CASE est indéfini.

Nous commençons par la forme générale de l'expression CASE, la plus puissante mais aussi le plus complexe.

```
SELECT p. Nom,
        CASE WHEN TYPE (p) = DesignProject ALORS 'Développement'
             WHEN TYPE (p) = QualityProject ALORS 'QA'
             ELSE 'Non-développement'
        FIN
FROM Projet p
O les employés ne sont pas vides
```

396

Épisode 411

La méthode selectCase () de l'interface CriteriaBuilder est utilisée pour créer l'expression CASE. Pour la forme générale, il ne prend aucun argument et renvoie un CriteriaBuilder.Case objet que nous pouvons utiliser pour ajouter les expressions conditionnelles au Instruction CASE. L'exemple suivant illustre cette approche:

```
CriteriaQuery <Object []> c = cb.createQuery (Object []. Class);
Racine <Projet> project = c.from (Project.class);
c.multiselect (project.get ("nom"),
               cb.selectCase ()
                  .when (cb.equal (project.type (), DesignProject.class),
                         "Développement")
                  .when (cb.equal (project.type (), QualityProject.class),
                         "QA")
                  .otherwise ("Non-développement"))
  .where (cb.isNotEmpty (project. <List <Employee>> get ("Employees")));
```

Les méthodes when () et else () correspondent aux mots clés WHEN et ELSE de JP QL. Malheureusement, «else» est déjà un mot-clé en Java, donc «autrement» doit être utilisé comme substitut.

L'exemple suivant simplifie l'exemple précédent jusqu'à la forme simple du Instruction CASE.

```
SELECT p. Nom,
        TYPE DE CAS (p)
             QUAND DesignProject PUIS 'Développement'
             QUAND QualityProject ALORS «QA»
             ELSE 'Non-développement'
        FIN
FROM Projet p
O les employés ne sont pas vides
```

Dans ce cas, nous passons l'expression primaire à tester à selectCase () et utilisez les méthodes when () et else () de CriteriaBuilder.
Interface SimpleCase. Plutôt qu'un prédicat ou une expression booléenne, ces méthodes acceptent désormais les expressions à valeur unique comparées à l'expression de base du Instruction CASE.

Épisode 412


```
CriteriaQuery <Object []> c = cb.createQuery (Object []. Class);
Racine <Projet> project = c.from (Project.class);
c.multiselect (project.get ("nom"),
                cb.selectCase (project.type ())
                    .when (DesignProject.class, "Développement")
                    .when (QualityProject.class, "QA")
                    .otherwise ("Non-développement"))
  .where (cb.isNotEmpty (project. <List <Employee>> ("Employees")));
```

Le dernier exemple que nous couvrons dans cette section concerne l'expression JP QL COALESCE.

```
SELECT COALESCE (d.name, d.id)
DU Département d
```

La création d'une expression COALESCE avec l'API Criteria nécessite une interface d'assistance comme les autres exemples que nous avons examinés dans cette section, mais il est plus proche de la forme IN que les expressions CASE. Ici, nous invoquons la méthode coalesce () sans arguments pour récupérer un objet CriteriaBuilder.Coalesce que nous utilisons ensuite la méthode value () de pour ajouter des valeurs à l'expression COALESCE. L'exemple suivant démontre cette approche:

```
CriteriaQuery <Objet> c = cb.createQuery ();
Racine <Department> dept = c.from (Department.class);
c.select (cb.coalesce ()
                .value (dept.get ("nom"))
                .value (dept.get ("id")));
```

Des versions pratiques de la méthode coalesce () existent également pour le cas où deux expressions sont comparées.

```
CriteriaQuery <Objet> c = cb.createQuery ();
Racine <Department> dept = c.from (Department.class);
c.select (cb.coalesce (dept.get ("nom"),
                       dept.get ("id")));
```

Une dernière remarque concernant les expressions de cas est qu'elles constituent une autre exception à la règle selon laquelle les méthodes CriteriaBuilder ne sont pas mutantes. Chaque méthode when () provoque une autre expression conditionnelle à ajouter de manière incrémentielle à l'expression de cas, et chaque value () ajoute une valeur supplémentaire à la liste de fusion.

Épisode 413

Expressions de fonction

À ne pas confondre avec les fonctions intégrées de JP QL, les expressions de fonction de critères sont l'équivalent API Criteria du mot clé FUNCTION dans JP QL. Ils permettent le SQL natif fonctions stockées à mélanger avec d'autres expressions de l'API Criteria. Ils sont destinés à les cas où une quantité limitée de SQL natif est requise pour satisfaire certaines exigences mais vous ne voulez pas convertir la requête entière en SQL.

Les expressions de fonction sont créées avec la méthode function () du
Interface CriteriaBuilder. Il nécessite comme arguments le nom de la fonction de base de données, le
le type de retour attendu, et une liste variable d'arguments, le cas échéant, à passer au
une fonction. Le type de retour est une expression, il peut donc être utilisé dans de nombreux autres endroits dans
la requête. L'exemple suivant appelle une fonction de base de données pour mettre en majuscule la première lettre
de chaque mot dans un nom de département:

```
CriteriaQuery <String> c = cb.createQuery (String.class);
Racine <Department> dept = c.from (Department.class);
c.select (cb.function ("initcap", String.class, dept.get ("nom")));
```

Comme toujours, les développeurs souhaitant maximiser la portabilité de leurs applications
doit être prudent en utilisant des expressions de fonction. Contrairement aux requêtes SQL natives, qui sont
clairement marquées, les expressions de fonction sont une petite partie de ce qui ressemble autrement à un
requête JPA portable normale qui est en fait liée à un comportement spécifique à la base de données.

Abattu

JPA 2.1 a introduit la prise en charge du downcasting de type lors de l'interrogation sur une entité
hiérarchie d'héritage via l'opération TREAT dans JP QL. Dans l'API Criteria, cette opération
est exposé via la méthode traiter () de l'interface CriteriaBuilder. Il peut être utilisé
lors de la construction de jointures pour limiter les résultats à une sous-classe spécifique ou dans le cadre de critères généraux
expressions afin d'accéder aux champs d'état à partir de sous-classes spécifiques.

> Notez que la requête ne sera pas valide si le type de cible n'est pas un sous-type du
> type statique du premier argument.

Épisode 414

La méthode traiter () a été surchargée pour renvoyer des objets Join ou Path
selon le type d'arguments. Rappelez-vous l'exemple du chapitre8 qui ont démontré
en utilisant traiter () pour limiter une requête Employé aux seuls employés qui travaillent sur un
QualityProject dont la cote de qualité est supérieure à cinq:

```
SELECT e FROM Employee a JOIN TREAT (e.projects AS QualityProject) qp
WHERE qp.qualityRating> 5
```

L'équivalent Critères de cette requête est le suivant:

```
CriteriaQuery <Employee> q = cb.createQuery (Employee.class); Racine <Employé>
emp = q.from (classe.employé); Rejoignez le projet <Employee, QualityProject> =
cb.treat (emp.join (emp.get ("projets"), QualityProject.class); q.select (emp)
    .where (cb.gt (projet. <Integer> get ("qualityRating"), 5));
```

Dans cet exemple, traiter () accepte un objet Join et renvoie un objet Join et peut
être référencé dans la clause WHERE car il est enregistré dans une variable. Lorsqu'il est utilisé directement
dans une expression de critère, comme dans une clause WHERE, il accepte un chemin ou un objet racine
et renvoie un chemin qui correspond à la sous-classe demandée. Par exemple, nous pourrions
utiliser la méthode traiter () pour accéder à l'évaluation de la qualité ou à la phase de conception d'une requête
à travers les projets. En général, l'opérateur TREAT peut également avoir pour effet de filtrer
type spécifié ainsi que l'exécution de la descente.

```
CriteriaQuery <Projet> q = cb.createQuery (Project.class); Racine <Projet>
project = q.from (Project.class); q.select (projet)
```

```
    .where (cb.ou (
                        cb.gt (cb.treat (projet, QualityProject.class).
<Integer> get ("qualityRating"), 5),
                        cb.gt (cb.treat (projet, DesignProject.class).
<Integer> get ("designPhase"), 3)));
```

Remarque La prise en charge du downcasting vers un sous-type a été ajoutée dans Jpa 2.1. Aucun changement ont été ajoutés dans la version Jpa 2.2.

Épisode 415

Critères de jointure externe

Plus tôt dans le chapitre, nous avons montré comment utiliser l'API Criteria pour créer des jointures entre entités. Nous avons également montré comment utiliser la méthode where () du Interface AbstractQuery et le cadre d'expression de l'API Criteria à construire les conditions de filtrage qui limiteront l'ensemble de résultats. Cela fonctionne bien pour les jointures internes, mais comme nous l'avons vu dans le dernier chapitre, les jointures externes nécessitent un support supplémentaire pour établir des critères de filtrage qui préservent toujours le caractère facultatif des entités jointes. Comme le où () méthode de l'interface AbstractQuery, l'interface Join prend en charge la méthode on () pour spécifier les critères de jointure externe qui doivent être évalués lors de la création du jeu de résultats. Cela équivaut à la condition ON de l'expression JP QL JOIN. Nous avons vu ce qui suit exemple de retour au chapitre 8 qui a démontré en utilisant le mot-clé ON dans JP QL:

```
SELECT e FROM Employee e JOIN e.projects p ON p.name = 'Zooby'
```

La méthode on () prend un seul objet prédicat pour représenter la condition de jointure. le les critères équivalents à cette requête JP QL seraient exprimés comme suit:

```
CriteriaQuery <Employé> q = cb.createQuery (Employee.class); Racine <Employé>
emp = q.from (classe.employé); Rejoignez <Employee, Project> project = emp.
join ("projets", JoinType.LEFT)
                    .on (cb.equal (project.get ("nom"), "Zooby"));
q.select (emp);
```

Remarque La prise en charge de la condition ON des requêtes de jointure externe a été ajoutée dans Jpa 2.1. Non des changements ont été ajoutés dans la version 2.2 de Jpa.

La clause ORDER BY

La méthode orderBy () de l'interface CriteriaQuery définit le classement d'une requête définition. Cette méthode accepte un ou plusieurs objets Order, qui sont créés par asc () et desc () de l'interface CriteriaBuilder, pour les méthodes ascendantes et descendantes commande, respectivement. L'exemple suivant illustre la méthode orderBy ():

```
CriteriaQuery <Tuple> c = cb.createQuery (Tuple.class);
Racine <Employee> emp = c.from (Employee.class);
```

Épisode 416


```
Rejoindre <Employé, Département> dept = emp.join ("dept");
c.multiselect (dept.get ("nom"), emp.get ("nom"));
c.orderBy (cb.desc (dept.get ("nom")),
                cb.asc (emp.get ("nom")));
```

L'ordre des requêtes via l'API Criteria est toujours soumis aux mêmes contraintes que JP QL. Les arguments de asc () et desc () doivent être des expressions à valeur unique, généralement formé à partir du champ d'état d'une entité. L'ordre dans lequel les arguments sont passés à la méthode orderBy () détermine la génération de SQL. L'équivalent JP QL pour le La requête présentée dans l'exemple précédent est la suivante:

```
SELECT d.name, e.name
FROM Employé e REJOINDRE e.dept d
ORDER BY d.name DESC, e.name
```

Les clauses GROUP BY et HAVING

Les méthodes groupBy () et having () de l'interface AbstractQuery sont les critères API équivalent des clauses GROUP BY et HAVING de JP QL, respectivement. Tous les deux Les arguments acceptent une ou plusieurs expressions utilisées pour regrouper et filtrer les données.

Voici comment la clause GROUP BY est définie:

```
groupby_clause :: = GROUP BY groupby_item {, groupby_item} *
groupby_item :: = single_valued_path_expression | identification_variable
```

Voici comment la clause HAVING est définie:

```
had_clause :: = HAVING expression_conditionnelle
```

À ce stade du chapitre, le modèle d'utilisation de ces méthodes devrait être plus intuitif pour vous. Prenons l'exemple suivant du chapitre précédent:

```
SELECT e, COUNT (p)
FROM Employé e REJOINDRE e.projects p
GROUPE PAR e
COMPTE (p)> = 2
```


Épisode 417

Pour recréer cet exemple avec l'API Criteria, nous devons utiliser à la fois l'agrégat les fonctions et les méthodes de regroupement. L'exemple suivant illustre cela conversion:

```
CriteriaQuery <Object []> c = cb.createQuery (Object []. Class);
Racine <Employee> emp = c.from (Employee.class);
Rejoignez <Employé, Projet> project = emp.join ("projets");
c.multiselect (emp, cb.count (projet))
  .groupBy (emp)
  .having (cb.ge (cb.count (projet), 2));
```

Mise à jour et suppression en masse

L'interface CriteriaBuilder n'est pas limitée aux requêtes de rapport. Mise à jour groupée et les requêtes de suppression peuvent également être créées à l'aide de createCriteriaUpdate () et createCriteriaDelete (), respectivement.

Une requête de mise à jour en bloc est construite à l'aide de la création et de la modification d'un javax. objet persistence.criteria.CriteriaUpdate.

Comme leurs homologues de requête de rapport, les versions CriteriaBuilder de la mise à jour en masse et les requêtes de suppression utilisent les mêmes méthodes pour générer les clauses FROM et WHERE de la requête. Les méthodes spécifiques aux opérations de mise à jour et de suppression en bloc sont encapsulées dans les interfaces CriteraUpdate et CriteriaDelete.

> Notez que bien que des mises à jour et des suppressions en masse existent dans Jp QL depuis Jpa 1.0, la prise en charge de l'exécution de mises à jour et de suppressions groupées à l'aide de l'API Criteria n'était pas ajouté jusqu'à Jpa 2.1. Aucune modification n'a été ajoutée dans la version 2.2 de Jpa.

Les requêtes de mise à jour en masse dans JP QL utilisent l'expression SET pour modifier l'état des entités. En miroir de cette fonctionnalité, la méthode set () de la requête CriteriaUpdate permet développeur d'attribuer des expressions construites à partir de l'API Criteria pour modifier champs d'état d'entité.

Un objet CriteriaUpdate est créé à l'aide de l'un des createCriteriaUpdate méthodes de l'interface CriteriaBuilder.

Épisode 418

Un objet CriteriaUpdate:

- Est tapé en fonction du type d'entité qui est la cible de la mise à jour.
- Possède une seule racine, qui est l'entité en cours de mise à jour.

Au chapitre 8, nous avons démontré la requête suivante pour attribuer une augmentation aux employés associé à un projet spécifique:

```
MISE À JOUR Employé e
SET e.salary = e.salary + 5000
O EXISTE (SELECT p
                DE e.projects p
                WHERE p.name = 'Release2')
```

À l'aide de l'API Criteria, la même requête serait construite comme suit:

```
CriteriaUpdate <Employee> q = cb.createCriteriaUpdate (Employee.class);
Racine <Employee> e = q.from (Employee.class); Sous-requête <Projet>
sq = c.subquery (Project.class);
Racine <Employé> sqEmp = sq.correlate (emp);
Rejoignez <Employé, Projet> project = sqEmp.join ("projets");
sq.select (projet) .where (cb.equal (project.get ("nom"), "Release2"));
q.set (emp.get ("salaire"), cb.sum (emp.get ("salaire"), 5000)) .where (cb.
existe (sq));
```

La méthode set () peut être appelée plusieurs fois pour la même requête. Ça aussi prend en charge la même capacité de chaînage d'appel que celle utilisée par d'autres méthodes de requête dans l'API Criteria.

Les opérations de suppression groupée sont très simples avec l'API Criteria. Le suivant

exemple convertit l'exemple de requête JP QL DELETE du chapitre 8 pour utiliser les critères API:

```
CriteriaDelete <Employee> q = cb.createCriteriaDelete (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
q.where (cb.isNull (emp.get ("dept"));
```

S'il est exécuté, cet exemple supprimera tous les employés qui ne sont affectés à aucun département.

Épisode 419

Définitions de requêtes fortement typées

Tout au long de ce chapitre, nous avons présenté les définitions de requête de l'API Criteria construit à l'aide de noms de chaîne pour faire référence aux attributs dans les expressions de chemin. Ce sous-ensemble de l'API Criteria est appelée API basée sur des chaînes. Nous l'avons mentionné à quelques reprises, cependant, qu'il existait également une approche alternative pour la construction d'expressions de chemin proposait une approche plus typée. Dans les sections suivantes, nous établissons quelques théorie autour du métamodèle qui sous-tend l'approche fortement typée puis démontrer son utilisation avec l'API Criteria.

L'API Metamodel

Avant d'examiner les définitions de requêtes fortement typées, nous devons d'abord préparer le terrain pour notre discussion avec une courte digression dans le métamodèle des unités de persistance dans un JPA application. Le *métamodèle* d'une unité de persistance est une description du type persistant, l'état et les relations des entités, des éléments incorporables et des classes gérées. Avec ça, on peut interroger le moteur d'exécution du fournisseur de persistance pour trouver des informations sur les classes dans une unité de persistance. Une grande variété d'informations, des noms aux types en passant par les relations, est stocké par le fournisseur de persistance et rendu accessible via l'API du métamodèle.

L'API du métamodèle est exposée via la méthode getMetamodel () du Interface EntityManager. Cette méthode retourne un objet implémentant le métamodèle interface, que nous pouvons ensuite utiliser pour commencer à naviguer dans le métamodèle. Le métamodèle l'interface peut être utilisée pour lister les classes persistantes dans une unité de persistance ou pour récupérer des informations sur un type persistant spécifique.

Par exemple, pour obtenir des informations sur la classe Employee, nous avons été pour démontrer dans ce chapitre, nous utiliserions la méthode entity ().

```
Métamodèle mm = em.getMetamodel ();
EntityType <Employee> emp_ = mm.entity (Employee.class);
```

Les méthodes équivalentes pour les éléments intégrés et les classes gérées sont embeddable () et managedType (), respectivement. Il est important de noter que l'appel à entity () dans ce exemple ne crée pas une instance de l'interface EntityType. Il s'agit plutôt de récupérer un objet métamodèle que le fournisseur de persistance aurait initialisé lorsque le EntityManagerFactory pour l'unité de persistance a été créé. Avait l'argument de classe pour entity () n'était pas un type persistant préexistant, une exception IllegalArgumentException ont été jetés.

Épisode 420

L'interface EntityType est l'une des nombreuses interfaces de l'API du métamodèle qui contiennent des informations sur les types et attributs persistants. Figure 9-2 montre le relations entre les interfaces qui composent l'API du métamodèle.

Figure 9-2. *Interfaces de métamodèle*

Pour développer davantage cet exemple, considérez un outil qui inspecte une unité persistante et imprime des informations récapitulatives sur la console. Pour énumérer tous les attributs et leurs types, nous pourrions utiliser le code suivant:

```
public <T> void listAttributes (type EntityType <T>) {
    for (Attribut <? super T,?> attr: type.getAttributes ()) {
        System.out.println (attr.getName () + "" +
                            attr.getJavaType (). getName () + "" +
                            attr.getPersistentAttributeType ());
    }
}
```

Épisode 421

Pour l'entité Employé, cela se traduirait par ce qui suit:

```
id int BASIC
nom java.lang.String BASIC
salaire flottant BASIC
département com.acme.Department MANY_TO_ONE
adresse com.acme.Address MANY_TO_ONE
dirige com.acme.Employee ONE_TO_MANY
téléphones com.acme.Phone ONE_TO_MANY
projets com.acme.Project ONE_TO_MANY
```

En quelques appels de méthode, nous avons découvert de nombreuses informations sur le Entité des employés. Nous avons répertorié tous les attributs et pour chacun d'entre eux que nous connaissons maintenant le nom de l'attribut, le type de classe et le type persistant. Les collections ont été déballées pour révéler le type d'attribut sous-jacent, et les différents types de relations sont clairement

marqué.

Du point de vue de l'API Criteria et en fournissant des requêtes fortement typées, nous sommes principalement intéressé par les informations de type exposées par les interfaces API du métamodèle. Dans la section suivante, nous montrons comment il peut être utilisé dans ce contexte.

Le métamodèle d'une unité de persistance n'est pas un nouveau concept. Versions précédentes de JPA ont toujours maintenu des structures similaires en interne pour une utilisation au moment de l'exécution, mais uniquement avec JPA 2.0 était ce genre de métamodèle exposé directement aux développeurs. Utilisation directe du les classes de métamodèles sont quelque peu spécialisées, mais, pour les développeurs d'outils ou les applications besoin de travailler avec une unité de persistance de manière complètement générique, le métamodèle est un source utile d'informations sur les types persistants.

Présentation de l'API fortement typée

L'API basée sur des chaînes de l'API Criteria se concentre sur la construction du chemin expressions: join (), fetch () et get () acceptent tous les arguments de chaîne. Le fortement l'API typée dans l'API Criteria prend également en charge les expressions de chemin en étendant la même méthodes, mais est également présente dans tous les aspects de l'interface de CriteriaBuilder, simplifiant la saisie et l'application de la sécurité des types lorsque cela est possible.

L'API fortement typée tire ses informations de type des classes d'API du métamodèle nous l'avons présenté dans la section précédente. Par exemple, la méthode join () est surchargée pour accepter des arguments de type SingularAttribute, CollectionAttribute, SetAttribute,

Épisode 422

ListAttribute et MapAttribute. Chaque version surchargée utilise les informations de type associé à l'interface d'attribut pour créer le type de retour approprié tel que MapJoin pour les arguments de type MapAttribute.

Pour démontrer ce comportement, nous revisitons un exemple du début du chapitre où nous avons été obligés d'utiliser joinMap () avec l'API basée sur des chaînes afin d'accéder au Objet MapJoin. Cette fois, nous utilisons l'API du métamodèle pour obtenir le type d'entité et d'attribut informations et transmettez-les aux méthodes de l'API Criteria.

```
CriteriaQuery <Objet> c = cb.createQuery ();
Racine <Employee> emp = c.from (Employee.class);
EntityType <Employee> emp_ = emp.getModel ();
MapJoin <Employé, Chaîne, Téléphone> phone =
      emp.join (emp_.getMap ("téléphones", String.class, Phone.class));
c.multiselect (emp.get (emp_.getSingularAttribute ("nom", String.class)),
                        phone.key (), phone.value ());
```

Il y a plusieurs choses à noter à propos de cet exemple. Le premier est l'utilisation de getModel (). Cette méthode existe sur de nombreuses interfaces de l'API Criteria en tant que raccourci vers le sous-jacent objet métamodèle. Nous l'assignons à une variable, emp_, et ajoutons un trait de soulignement par convention pour aider à le désigner comme un type de métamodèle. Deuxièmement, les deux appels aux méthodes du Interface EntityType. L'appel getMap () renvoie l'objet MapAttribute pour l'attribut phones, tandis que l'invocation getSingularAttribute () renvoie le Objet SingularAttribute pour l'attribut de nom. Encore une fois, nous devons fournir le type informations sur l'attribut, en partie pour satisfaire les exigences de type générique de invocation de méthode mais aussi comme mécanisme de vérification de type. Avait l'un des arguments été incorrecte, une exception aurait été levée. Notez également que la jointure () La méthode ne qualifie plus le type de collection mais renvoie l'instance MapJoin correcte. La méthode join () est surchargée pour se comporter correctement en présence des différents interfaces d'attribut de collection de l'API du métamodèle.

Le potentiel d'erreur dans l'utilisation des objets de métamodèle est en fait le cœur de ce le rend fortement typé. En faisant en sorte que les informations de type soient disponibles, les critères

L'API peut garantir que non seulement les objets appropriés sont transmis en tant que méthode
arguments mais aussi que des types compatibles sont utilisés dans diverses méthodes de construction d'expression.

Il ne fait cependant aucun doute que c'est une manière beaucoup plus verbeuse de construire
une requête. Heureusement, la spécification JPA 2.0 définit également une présentation alternative de
le métamodèle d'unité de persistance qui est conçu pour faire de la programmation fortement typée
Plus facile. Nous discutons de ce modèle dans la section suivante.

Épisode 423

Le métamodèle canonique

Jusqu'à présent, notre utilisation de l'API du métamodèle a ouvert les portes à une vérification de type solide
mais au détriment de la lisibilité et de la complexité accrue. Les API du métamodèle sont
pas complexes, mais ils sont verbeux. Pour simplifier leur utilisation, JPA fournit également une *canonique métamodèle* pour une unité de persistance.

Le métamodèle canonique se compose de classes dédiées, généralement générées une par
classe persistante, qui contient les déclarations statiques des objets métamodèles associés
avec cette classe persistante. Cela vous permet d'accéder aux mêmes informations exposées
via l'API du métamodèle, mais sous une forme qui s'applique directement à vos classes persistantes.
Le Listing 9-7 montre un exemple de classe de métamodèle canonique.

Annonce 9-7. La classe de métamodèle canonique pour les employés

```
@StaticMetamodel (Employee.class)
public class Employee_ {
    public static volatile SingularAttribute <Employee, Integer> id;
    public static volatile SingularAttribute <Employee, String> nom;
    public statique volatile SingularAttribute <Employee, String> salaire;
    public static volatile SingularAttribute <Employee, Department> dept;
    adresse publique statique volatile SingularAttribute <Employee, Address>;
    projet CollectionAttribute <Employee, Project> volatile statique public;
    téléphones MapAttribute <Employee, String, Phone> statiques et volatils publics;
}
```

Chaque classe de métamodèle canonique contient une série de champs statiques, un pour chaque
attribut dans la classe persistante. Chaque champ de la classe de métamodèle canonique est de la
type de métamodèle qui correspond au type du champ ou de la propriété du même nom dans
la classe persistante. Si un champ ou une propriété persistante dans une entité est de type primitif
ou une relation à valeur unique, puis le champ du même nom dans le métamodèle canonique
La classe sera de type SingularAttribute. De même, si un champ ou une propriété persistante
a une valeur de collection, alors le champ de la classe de métamodèle canonique sera de type
ListAttribute, SetAttribute, MapAttribute ou CollectionAttribute, selon
le type de collection. De plus, chaque classe de métamodèle canonique est annotée avec @
StaticMetamodel, qui identifie la classe persistante qu'il modélise.

Épisode 424

Une classe de métamodèle canonique est générée dans le même package que sa classe associée
classe persistante et porte le même nom, mais avec un suffixe de soulignement supplémentaire.

Les classes de métamodèles non canoniques peuvent être générées dans d'autres packages et avec des noms différents s'il y a lieu de le faire. Certains outils de génération peuvent fournir ces types d'options. L'annotation @StaticMetamodel fournit la liaison entre les classe de métamodèle et l'entité, pas le nom ou le package, il n'y a donc pas de méthode standard de lecture dans ces métamodèles. Si un outil fournisseur peut générer le métamodèle concret classes sous une forme non canonique, alors ce runtime peut être nécessaire pour reconnaître ou les détecter également.

Utilisation du métamodèle canonique

Nous sommes maintenant en mesure de tirer parti du métamodèle sans réellement en utilisant l'API du métamodèle. A titre d'exemple, nous pouvons convertir l'exemple du précédent section pour utiliser les classes canoniques générées statiquement.

```
CriteriaQuery <Objet> c = cb.createQuery ();
Racine <Employee> emp = c.from (Employee.class);
MapJoin <Employee, String, Phone> phone = emp.join (Employee_.phones);
c.multiselect (emp.get (nom_employé), phone.key (), phone.value ());
```

Il s'agit d'une approche beaucoup plus concise de l'utilisation des objets de métamodèle que interfaces dont nous avons discuté précédemment tout en offrant exactement les mêmes avantages. Couplé avec un environnement de développement qui a de bonnes fonctionnalités de complétion de code, vous pouvez le trouver plus pratique à développer en utilisant le métamodèle canonique qu'avec l'API basée sur des chaînes.

Nous pouvons convertir un exemple plus complexe pour illustrer en utilisant le canonique métamodèle. Le Listing 9-8 convertit l'exemple en Listing9-6, montrant une expression IN qui utilise une sous-requête, de l'utilisation des noms d'attributs basés sur des chaînes à l'utilisation approche typée.

Annonce 9-8. Requête fortement typée

```
CriteriaQuery <Employee> c = cb.createQuery (Employee.class);
Racine <Employee> emp = c.from (Employee.class);
Sous-requête <Department> sq = c.subquery (Department.class);
Racine <Department> dept = sq.from (Department.class);
Rejoindre le projet <Employé, Projet> =
    dept.join (Department_.employees) .join (Employee_.projects);
```



```
sq.select (dept.get (Department_.id))
    .distinct (vrai)
    .where (cb.like (project.get (Project_.name), "QA%"));
c.select (emp)
  .where (cb.in (emp.get (Employee_.dept) .get (Department_.id)). value (sq));
```

Notez qu'il existe deux différences principales entre cet exemple et l'extrait 9-6 . Premier, l'utilisation de champs statiques de métamodèle typés pour indiquer les attributs d'entité permet d'éviter le typage erreurs dans les chaînes d'attributs. Deuxièmement, il n'est plus nécessaire de faire de la saisie en ligne pour convertir le Path <Object>, renvoyé par la méthode get (), en un type plus étroit. le un typage plus fort résout ce problème pour nous.

Tous les exemples de ce chapitre peuvent être facilement convertis pour utiliser le canonique classes de métamodèles en changeant simplement les attributs basés sur des chaînes en champs statiques des classes de métamodèles générées. Par exemple, emp.get ("nom") peut être remplacé par emp.get (Employee_.name), et ainsi de suite.

Génération du métamodèle canonique

Si vous choisissez d'utiliser le métamodèle généré dans vos requêtes, vous devez être conscient de

certains des détails du processus de développement en cas d'incohérence ou de configuration
des problèmes surgissent. Les classes de métamodèles canoniques devront être mises à jour ou régénérées
lorsque certains changements d'entités se sont produits au cours du développement. Par exemple,
changer le nom d'un champ ou d'une propriété, ou changer sa forme, nécessiterait un
classe de métamodèle canonique mise à jour pour cette entité.

Les outils de génération proposés par les fournisseurs peuvent varier considérablement en fonction ou en
opération. La génération peut impliquer la lecture du fichier persistence.xml, ainsi que
accéder aux annotations sur les entités et aux fichiers de mappage XML pour déterminer ce que
Les classes de métamodèles devraient ressembler à. Étant donné que la spécification ne nécessite pas de tels outils
pour même exister, un fournisseur peut choisir de ne pas le prendre en charge du tout, en espérant que si les développeurs
veulent utiliser les classes de métamodèles canoniques, ils les coderont à la main. La plupart des fournisseurs
offrent cependant une sorte d'outil de génération; c'est juste une question de comprendre comment
cet outil spécifique au fournisseur fonctionne. Il peut fonctionner de manière statique comme un outil de ligne de commande distinct,
ou il peut utiliser le hook du compilateur proposé dans le JDK (à partir de Java SE 6) pour examiner le
entités et générer les classes au moment de la compilation. Par exemple, pour exécuter la commande-
en mode ligne de l'outil livré avec l'implémentation de référence EclipseLink, vous
pourrait définir les options javac -processor et -proc: only. Ces deux options indiquent

Épisode 426

le processeur de code / d'annotation EclipseLink[1] pour que le compilateur invoque et demande au
compilateur pour appeler uniquement le processeur mais ne pas faire de compilation réelle.

```
javac -processor org.eclipse.persistence.internal.jpa.modelgen.
CanonicalModelProcessor
        -proc: uniquement
        -classpath lib / *. jar; punit
        *.Java
```

Les options sont sur des lignes séparées pour les rendre plus faciles à voir. On suppose que le
lib contient le JAR EclipseLink et le JAR d'interface JPA nécessaires, et que le
META-INF / persistence.xml se trouve dans le répertoire punit.

Les outils de génération de métamodèles fonctionneront également généralement dans un IDE, et il y aura probablement
être une configuration spécifique à l'IDE nécessaire pour diriger le compilateur incrémentiel pour qu'il utilise
processeur d'annotation de l'outil. Dans certains IDE, il doit y avoir un code / annotation supplémentaire
processeur JAR à configurer. Les classes de métamodèles générées doivent aller dans un
répertoire et être sur le chemin de classe de construction afin que les requêtes de critères qui les référencent puissent
compiler. Consultez les fichiers d'aide de l'IDE pour savoir comment les processeurs d'annotations ou APT sont pris en charge,
ainsi que la documentation du fournisseur sur ce qu'il faut configurer pour activer
génération dans un IDE donné.

Choisir le bon type de requête

Maintenant que vous êtes familiarisé avec les requêtes de critères, vous êtes bien armé avec deux des trois
langages distincts dans lesquels créer des requêtes: JP QL, SQL natif et l'API Criteria. nous
ont démontré que l'API Criteria est relativement facile à comprendre et à utiliser, mais
quand doit-il être utilisé? La réponse est une combinaison de vos propres préférences en tant que
développeur et les capacités des différentes approches de requête.

Les requêtes SQL natives sont un choix facile à faire: soit vous devez accéder à un fournisseur.
caractéristique spécifique ou vous ne le faites pas. Si vous le faites, il n'y a qu'une seule option disponible si vous ne pouvez pas
contourner la dépendance. Vous avez découvert dans ce chapitre que JP QL et les critères

[1] Ceci est souvent appelé un outil de traitement des annotations, ou APT, car il s'agissait d'un support
seul outil livré avec le JDK et utilisé uniquement pour le traitement des annotations. Depuis Java SE 6,

c'est en fait un hook généralisé au moment de la compilation pour effectuer tout type de prétraitement ou de code génération.

Épisode 427

Les API sont presque complètement équivalentes en fonctionnalités. Quand devriez-vous utiliser un texte définition de requête sur une créée à partir des API de programmation?

L'API de programmation s'écoule facilement de l'application. Il peut être fortement typé pour réduire les risques d'erreurs et avec les fonctionnalités de complétion de code de la plupart des les environnements de développement, il peut être relativement rapide à mettre en place. Il est également idéal pour les cas où la définition de la requête ne peut pas être entièrement construite avec l'entrée d'un utilisateur.

JP QL, en revanche, est plus concis et familier à toute personne expérimentée SQL. Il peut également être intégré aux annotations d'application ou aux descripteurs XML et maintenue indépendamment du processus de programmation régulier. Quelques développements Les environnements offrent également des constructeurs visuels pour les requêtes JP QL, ce qui en fait une partie intégrante du processus de développement.

Il n'y a pas de bonne réponse quand il s'agit de choisir entre JP QL et les critères API. Ils peuvent être mélangés et assortis dans une application comme bon vous semble. En général, cependant, nous encourageons toujours les développeurs à utiliser autant que possible les requêtes nommées, et, pour cela, JP QL est le choix idéal.

Résumé

Dans ce chapitre, nous avons visité l'API Criteria. Nous avons commencé par un aperçu et exemple axé sur la création de requêtes dynamiques.

En examinant l'API Criteria, nous avons commencé avec les concepts JP QL et avons cherché à établissez des parallèles entre le langage de requête et l'API Criteria. Nous avons regardé comment formuler chaque clause d'une requête avec l'API Criteria et adresser certains des problèmes complexes rencontrés par les développeurs.

Nous avons présenté l'API du métamodèle et montré comment elle peut être utilisée pour créer fortement requêtes typées avec l'API Criteria. Nous avons examiné la programmation avec le métamodèle API utilisant à la fois les interfaces d'exécution et via les classes générées d'un canonical mise en œuvre du métamodèle.

Enfin, nous avons discuté des avantages des différentes approches de création de requêtes en utilisant JP QL, SQL natif et l'API Criteria. Nous avons souligné certaines des forces de les langages de requête et a présenté des conseils sur le moment où il pourrait être plus avantageux. Nous avons réitéré que le bon langage de requête pour vous dépendra principalement du style et Préférence personnelle.

Épisode 428

Au cours de ce chapitre, nous avons essayé de démontrer que bien que le forme de l'API Criteria est très différente du langage de requête JP QL, le sous-jacent la sémantique est presque identique et la commutation entre les deux est relativement facile une fois maîtrisez les modèles de codage de l'API.

Dans le chapitre suivant, nous revenons au mappage objet-relationnel et couvrons

concepts tels que l'héritage, les clés primaires composites et les associations, et mappages de table.

Épisode 429

CHAPITRE 10

Objet avancé Cartographie relationnelle

Chaque application est différente et, bien que la plupart aient des éléments de complexité eux, les parties difficiles dans une application auront tendance à être différentes de celles des autres types d'applications. Quelle que soit l'application sur laquelle vous travaillez, il y a de fortes chances que à tout moment, il faudra utiliser au moins une fonctionnalité avancée de l'API. Ce chapitre présente et explique certaines de ces fonctionnalités ORM plus avancées.

La plupart des fonctionnalités de ce chapitre sont destinées aux applications qui doivent réconcilier les différences entre un modèle de données existant et un modèle objet. Pour exemple, lorsque les données d'une table d'entités seraient mieux décomposées dans l'objet modèle en tant qu'entité et objet dépendant référencé par l'entité, l'infrastructure de cartographie devrait pouvoir prendre en charge cela. De même, lorsque l'entité les données sont réparties sur plusieurs tables, la couche de mappage doit permettre ce type de

configuration à spécifier.

Dans ce livre, les discussions sur la manière dont les entités de JPA sont juste des classes Java régulières et non des objets spéciaux qui sont nécessaires pour étendre un sous-classe ou implémenter des méthodes spéciales. L'un des avantages des entités étant Java standard classes, c'est qu'ils peuvent adhérer aux concepts et pratiques déjà établis qui existent systèmes orientés objet. L'une des innovations traditionnelles orientées objet est l'utilisation de l'héritage et la création d'objets dans une hiérarchie afin d'hériter de l'état et du comportement.

Ce chapitre décrit certaines des fonctionnalités de cartographie les plus avancées et explore dans les diverses possibilités offertes par l'API et la couche de cartographie. On voit aussi comment l'héritage fonctionne dans le cadre de l'API Java Persistence et comment il affecte le modèle.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_10

Épisode 430

Noms de table et de colonne

Dans les sections précédentes, nous avons montré les noms des tables et des colonnes en majuscules identifiants. Nous l'avons fait, d'abord, car cela permet de les différencier des identifiants Java et, deuxièmement, parce que la norme SQL définit que les identifiants de base de données non délimités ne respectez la casse, et la plupart ont tendance à les afficher en majuscules.

Partout où un nom de table ou de colonne est spécifié ou est défini par défaut, la chaîne d'identificateur est transmis au pilote JDBC exactement comme il est spécifié ou défini par défaut. Par exemple, lorsqu'aucun nom de table n'est spécifié pour l'entité Employee, alors le nom de la table assumé et utilisé par le fournisseur sera Employee, qui par définition SQL n'est pas différent de EMPLOYÉ. Le fournisseur n'est ni obligé ni censé faire quoi que ce soit pour essayez d'ajuster les identifiants avant de les transmettre au pilote de base de données.

Les annotations suivantes devraient donc être équivalentes en ce qu'elles renvoient aux même table dans une base de données conforme à la norme SQL:

```
@Table (nom = "employé")
@Table (nom = "Employé")
@Table (nom = "EMPLOYÉ")
```

Certains noms de base de données sont destinés à être spécifiques à la casse et doivent être explicitement délimité. Par exemple, une table peut s'appeler EmployeeSales, mais sans cas la distinction deviendrait EMPLOYEESALES, nettement moins lisible et plus difficile à déterminer sa signification. Bien que ce ne soit en aucun cas une pratique courante ou une bonne pratique, une base de données en théorie pourrait avoir une table EMPLOYEE ainsi qu'une table Employee. Celles-ci devraient être délimité afin de faire la distinction entre les deux. La méthode de délimitation est l'utilisation d'un deuxième ensemble de guillemets doubles, qui doivent être échappés, autour de l'identifiant. le Le mécanisme d'échappement est la barre oblique inverse (le caractère \), ce qui provoquerait ce qui suit annotations pour faire référence à différents tableaux:

```
@Table (nom = "\" Employé \ "")
@Table (nom = "\" EMPLOYÉ \ "")
```

Notez que l'ensemble extérieur de guillemets doubles n'est que le délimiteur habituel des chaînes dans les éléments d'annotation, mais les guillemets doubles internes sont précédés de la barre oblique inverse pour les faire échapper, indiquant qu'ils font partie de la chaîne et non de la chaîne terminateurs.

Épisode 431

Lors de l'utilisation d'un fichier de mappage XML, l'identifiant est également délimité par des guillemets dans le nom de l'identifiant. Par exemple, les deux éléments suivants représentent des Colonnes:

<nom de la colonne = "& quot; ID & quot;" />
<nom de la colonne = "& quot; Id & quot;" />

La méthode d'échappement XML est différente de celle utilisée en Java. Au lieu de en utilisant la barre oblique inverse, XML s'échappe avec une esperluette (&) suivie d'un mot décrivant l'élément spécifique échappé (dans ce cas, "quot") et enfin un caractère point-virgule (;).

Conseil Certains fournisseurs prennent en charge des fonctionnalités pour normaliser la casse des identifiants sont stockés et transmis dans les deux sens entre le fournisseur et le pilote jdbC. cela fonctionne autour de certains pilotes jdbC qui, par exemple, acceptent les majuscules identificateurs dans l'instruction SELECT de la requête SQl native, mais les renvoyer mappés aux identifiants en minuscules.

Parfois, la base de données est configurée pour utiliser des identificateurs spécifiques au cas, et elle deviendrait plutôt fastidieux (et a l'air extrêmement moche) d'avoir à mettre des guillemets supplémentaires sur chaque simple nom de la table et de la colonne. Si vous vous trouvez dans cette situation, il y a une commodité dans le fichier de mappage XML qui vous sera utile.

En général, un fichier XML de mappage objet-relationnel contiendra le mappage informations pour les classes qui y sont répertoriées. Les fichiers de mappage XML sont traités au chapitre 13 .

En incluant l'élément vide delimited-identifiers dans le fichier de mappage XML, tous les identifiants de l'unité de persistance seront traités comme délimités et les guillemets seront ajouté à eux lorsqu'ils sont transmis au pilote. Le seul hic, c'est qu'il n'y a aucun moyen pour remplacer ce paramètre. Une fois que l'indicateur des identifiants délimités est activé, tous les identifiants doivent être spécifiés exactement tels qu'ils existent dans la base de données. De plus, si vous décidez de activez l'option identificateurs délimités, assurez-vous de supprimer tous les guillemets échappés dans vos noms d'identifiant ou vous constaterez qu'ils seront inclus dans le nom. En utilisant échapper en plus de l'option des identificateurs délimités prendra les guillemets et enveloppez-les avec d'autres guillemets, en faisant les échappés faire partie de l'identifiant.

Épisode 432

Conversion de l'état de l'entité

Retour au chapitre 4 nous avons montré certaines des façons dont certains types de données, tels que les types énumérés, les types temporels et les objets volumineux peuvent être stockés dans la base de données. Cependant, il ne serait pas pratique de définir un soutien ciblé pour chaque type de des données qui ne sont pas primitives, notamment parce qu'il y aura toujours une certaine application qui souhaite stocker les données d'une manière nouvelle et différente, ou est contraint de le faire en raison d'un base de données héritée. En outre, une application peut définir son propre ensemble de types que le la spécification ne pourrait clairement jamais prévoir. Une solution plus flexible et évolutive, donc,

est de concevoir un moyen permettant à l'application de définir sa propre stratégie de conversion pour quel que soit le type qu'il juge nécessaire de convertir. Ceci est rendu possible grâce à l'utilisation de convertisseurs.

Remarque Des convertisseurs ont été ajoutés à la spécification jpa 2.1. aucun changement n'a été ajouté à la version 2.2 de jpa.

Créer un convertisseur

Un convertisseur est une classe spécifiée par l'application qui implémente le Interface AttributeConverter <X, Y>. Il peut être appliqué de manière déclarative à persistante attributs dans n'importe quelle entité, superclasse mappée ou classe intégrable pour contrôler la façon dont l'état de l'attribut est stocké dans la base de données. Il définit de la même manière comment l'état est converti de la base de données dans l'objet. Ces deux opérations de conversion comprennent le deux méthodes de l'interface AttributeConverter <X, Y>, illustrées dans le Listing 10-1.

Liste 10-1. Interface AttributeConverter

```
interface publique AttributeConverter <X, Y> {

    public Y convertToDatabaseColumn (attribut X);

    public X convertToEntityAttribute (Y dbData);
}
```

Notez que l'interface doit être définie avec deux paramètres de type. Le premier type le paramètre représente le type de l'attribut d'entité tandis que le second représente le Type JDBC à utiliser lors du stockage des données dans la colonne de base de données. Un convertisseur d'échantillons

la classe d'implémentation est affichée dans la liste 10-2 . Il illustre un simple convertisseur nécessaire pour stocker un attribut d'entité booléenne sous forme d'entier dans la base de données.

Notez qu'il n'y a actuellement qu'une prise en charge pour convertir un attribut d'entité en un seul colonne dans la base de données. la possibilité de le stocker sur plusieurs colonnes peut être normalisé dans une prochaine version.

Liste 10-2. Convertisseur booléen en entier

```
@Convertisseur
La classe publique BooleanToIntegerConverter implémente AttributeConverter <Boole
an, Entier> {

    public Integer convertToDatabaseColumn (attribut booléen) {
        return (attrib? 1: 0);
    }

    public Boolean convertToEntityAttribute (Integer dbData) {
        return (dbData> 0)
    }
}
```

La classe de convertisseur est annotée afin que le conteneur puisse la détecter et la valider. Les méthodes de conversion sont triviales dans ce cas car le processus de conversion est un tel simple, mais la logique de conversion pourrait être plus étendue s'il en fallait davantage.

Les convertisseurs peuvent être utilisés pour la conversion explicite ou automatique de l'état de base,

les sections suivantes décrivent.

Remarque Les convertisseurs sont des *classes gérées* , par conséquent, lors de l'exécution en java Se, chaque la classe de convertisseur doit être incluse dans un élément de classe dans le fichier persistence.xml descripteur. les schémas persistence.xml et orm.xml ont été mis à jour dans jpa version 2.2.

Épisode 434

Conversion d'attributs déclaratifs

Un convertisseur peut être utilisé explicitement pour convertir un attribut d'entité en annotant le attribut avec @Convert et définissant l'élément convertisseur sur la classe de convertisseur. Par exemple, si nous avions un attribut booléen nommé «bonded» dans notre entité Employee, il pourrait être converti en un entier dans la colonne de la base de données mappée si nous annotions l'attribut avec @Convert et spécifie la classe de convertisseur déclarée dans le précédent section:

```
@Convert (convertisseur = BooleanToIntegerConverter.class)
private Boolean lié;
```

L'attribut converti doit être du type correct. Depuis le premier le type paramétré du convertisseur est booléen, l'attribut que nous convertissons doit être de type booléen. Cependant, les types wrapper et primitifs sont automatiquement renvoyés pendant conversion, de sorte que l'attribut aurait pu également être de type booléen.

Conversion d'attributs incorporés

Si l'attribut à convertir fait partie d'un type intégrable et que nous convertissons il à partir d'une entité de référence, puis nous utilisons l'élément attributeName pour spécifier l'attribut à convertir. Par exemple, si l'attribut lié était contenu dans un L'objet intégrable SecurityInfo et l'entité Employee avaient un attribut securityInfo, alors nous pourrions le convertir comme indiqué dans la liste 10-3 .

Liste 10-3. Conversion d'un attribut incorporé

```
@Entité
Employé de classe publique {
    // ...
    @Convert (convertisseur = BooleanToIntegerConverter.class,
    attributeName = "lié")
    private SecurityInfo securityInfo;
    // ...
}

@Embeddable
public class SecurityInfo {
```

Épisode 435


```
    private Boolean lié;
    // ...
}
```

Dans la section «Objets intégrés complexes» plus loin dans ce chapitre, nous décrivons des utilisations plus avancées des éléments embarqués; en particulier, la notation par points est représentée par un signifie remplacer les éléments incorporables imbriqués. Cette même notation peut également être utilisée dans le attribut Nom de l'annotation @Convert pour référencer un attribut incorporé imbriqué.

Conversion de collections

Au chapitre 5 , nous avons montré que les collections peuvent être mappées en tant que collections d'éléments si le les valeurs sont de type basique ou intégrable, ou mappées comme un-à-plusieurs ou plusieurs-à-plusieurs relations si les valeurs sont un type d'entité. Les convertisseurs peuvent être appliqués à l'élément mais comme les convertisseurs ne convertissent pas les instances d'entités, ils ne peuvent pas, en général, être appliqué aux mappages de relations[1] .

Les collections d'éléments de types de base sont simples à convertir. Ils sont annotés le de la même manière que les attributs de base qui ne sont pas des collections. Ainsi, si nous avions un attribut qui était de type List <Boolean>, nous l'annoterions comme nous l'avons fait avec notre attribut lié, et toutes les valeurs booléennes de la collection seraient converties:

```
@ElementCollection
@Convert (convertisseur = BooleanToIntegerConverter.class)
liste privée <Boolean> securityClearances;
```

Si une collection d'éléments est une carte avec des valeurs de type basique, alors les valeurs de la carte sera convertie. Pour effectuer la conversion sur les clés, l'attributName L'élément doit être utilisé avec une valeur spéciale de «clé» pour indiquer que les clés de la carte doivent être convertis à la place des valeurs.

Rappel de la section «Remplacer les attributs intégrables» du chapitre 5 que si nous avions une collection d'éléments qui était une carte, et elle contenait des éléments incorporables, puis nous préfixé l'élément de nom de @AttributeOverride avec «clé». ou «valeur». C'était pour distinguer si nous remplaçions la colonne pour une clé ou une valeur intégrable de la carte. De même, si nous voulons convertir un attribut intégrable dans une carte, alors nous préfixe l'élément attributeName avec «clé». ou "valeur", suivi du nom

[1] Sauf dans le cas décrit plus loin dans cette section.

Épisode 436

de l'attribut intégrable à convertir. Comme cas particulier, nous pouvons utiliser la "clé" ou "clé." préfixe sur l'une des relations de collection un-à-plusieurs ou plusieurs-à-plusieurs pour mapper les clés s'il s'agit de types basiques ou intégrables, respectivement. Utilisation du domaine modèle dans la liste 5-17 , si nous voulions convertir le nom de famille de l'employé à stocker sous caractères majuscules (en supposant que nous ayons défini la classe de convertisseur correspondante), nous annoterait l'attribut comme indiqué dans la liste 10-4 .

Annonce 10-4. Conversion d'une clé d'attribut intégrable dans une carte de relations

```
@Entité
Département de classe publique {
    // ...
    @Plusieurs à plusieurs
```

```
    @MapKey (nom = "empName")
    @Convert (convertisseur = UpperCaseConverter.class, attributeName = "clé.
    nom de famille")
    Carte privée <EmployeeName, Employee> employés;
    // ...
}
```

Limites

Il y a quelques restrictions imposées aux convertisseurs, principalement pour empêcher les utilisateurs de faire des choses qui leur causeraient des ennuis. Par exemple, les convertisseurs ne peuvent pas être utilisés sur les attributs d'identifiant, les attributs de version ou les attributs de relation (sauf si vous êtes conversion de la partie clé d'une carte de relations, comme dans l'extrait 10-4). Espérons que ce sera ne vous surprend pas, car dans la plupart des cas, la conversion de ces types d'attributs serait sans doute une très mauvaise idée. Ces attributs sont fortement utilisés par le fournisseur comme il gère les entités; changer leur forme ou même leur valeur pourrait causer incohérences ou résultats incorrects.

Les convertisseurs ne peuvent pas non plus être utilisés sur les attributs annotés avec @Enumerated ou @Temporel, mais cela ne signifie pas que vous ne pouvez pas convertir les types énumérés ou temporels les types. Cela signifie simplement que si vous utilisez les annotations standard @Enumerated ou @Temporal pour mapper ces types, vous ne pouvez pas non plus utiliser de convertisseurs personnalisés. Si vous utilisez un classe de convertisseur personnalisée, vous prenez alors le contrôle de la valeur qui est stockée et vous n'avez pas besoin d'utiliser l'une de ces annotations pour demander au fournisseur JPA de faire le

Épisode 437

conversion pour vous. En termes simples, si vous effectuez une conversion personnalisée à l'aide de convertisseurs sur types énumérés ou temporels, laissez simplement les annotations @Enumerated ou @Temporal désactivées.

Conversion automatique

Lorsque nous avons défini un convertisseur pour convertir un booléen en entier, nous avions probablement à l'esprit qu'il serait utilisé dans des endroits très spécifiques, sur un ou peut-être quelques attributs. Tu ne veulent généralement pas convertir chaque attribut booléen de votre domaine en un entier. Toutefois, si vous utilisez fréquemment un type de données plus sémantiquement riche, tel que l'URL class, vous souhaiterez peut-être que chaque attribut de ce type soit converti. Tu peux le faire en définissant l'option autoApply sur l'annotation @Converter. Dans la liste10-5 , une URL convertisseur est déclaré avec l'option autoApply activée. Cela entraînera chaque persistant attribut de type URL dans l'unité de persistance à convertir en chaîne lorsque l'entité qui le contient est écrit dans la base de données.

> Notez qu'il n'est pas défini si deux convertisseurs sont déclarés être appliqués automatiquement au même type d'attribut.

Annonce 10-5. Convertisseur URL-chaîne

```
@Converter (autoApply = true)
La classe publique URLConverter implémente AttributeConverter <URL, String> {

    public String convertToDatabaseColumn (attribut URL) {
        return attrib.toString ();
    }

    URL publique convertToEntityAttribute (String dbData) {
```

```
        essayez {return nouvelle URL (dbData); }
            catch (MalformedURLException ex) {
            lancer une nouvelle exception IllegalArgumentException ("DB data:" + dbData + "est
pas une URL légale ");}
    }
}
```


Épisode 438

Nous pouvons remplacer la conversion par attribut. Si, au lieu de l'auto-convertisseur appliqué, nous voulons utiliser un convertisseur différent pour un attribut donné, puis nous peut annoter l'attribut avec l'annotation @Convert et spécifier la classe de convertisseur nous voulons utiliser. Alternativement, si nous voulons désactiver complètement la conversion et laisser le fournisseur revient à la sérialisation de l'URL, alors nous pouvons utiliser la disableConversion attribut:

```
@Convert (disableConversion = true)
URL homePage;
```

Convertisseurs et requêtes

Définir les convertisseurs et configurer les attributs à convertir, c'est à peu près tout vous devez faire pour que la conversion fonctionne. Mais il y a quelques points supplémentaires lié à la conversion dont vous devez être conscient lors de l'interrogation.

Le premier est que le processeur de requêtes appliquera le convertisseur aux deux attributs ciblés pour la conversion ainsi que les littéraux auxquels ils sont comparés dans une requête. Cependant, les opérateurs ne sont pas modifiés. Cela signifie que seule une certaine comparaison les opérateurs fonctionneront une fois la requête convertie. Pour illustrer ce point, considérons le requête suivante:

```
SELECT e FROM Employé e WHERE e.bonded = true
```

Cette requête fonctionnera correctement si bonded est configuré pour être converti de booléen en entier. le SQL généré aura converti à la fois l'attribut lié et le littéral true en entier correspondant en appelant la méthode convertToDatabaseColumn () dessus et L'opérateur égal fonctionnera aussi bien sur les entiers que sur les booléens.

Cependant, nous pouvons souhaiter interroger tous les employés qui ne sont pas liés:

```
SÉLECTIONNEZ e FROM Employé e O PAS e.bonded
```

Si nous essayons d'exécuter cette requête, l'analyseur n'aura aucun problème avec elle, mais quand il vient le temps de l'exécuter le SQL résultant contiendra un NOT et la valeur de e.bonded aura été converti en un entier. Cela provoquera généralement une exception de base de données car l'opération NOT ne peut pas être appliquée à un entier.

Épisode 439

Il est possible que vous rencontriez un problème ou deux si vous
interrogation sur les attributs convertis. Alors que vous pouvez généralement compter sur la conversion de littéraux et les paramètres d'entrée utilisés en comparaison, s'ils sont contenus dans une fonction, comme UPPER () ou MOD (), ils ne seront probablement pas convertis. Même certains des plus les opérations de comparaison avancées, telles que LIKE, peuvent ne pas appliquer la conversion au littéral opérande. La morale est d'essayer de ne pas utiliser d'attributs convertis dans les requêtes, et si vous le faites, jouez autour et faites quelques essais pour vous assurer que vos requêtes fonctionnent comme prévu.

Objets intégrés complexes

Dans le chapitre 4 , nous avons examiné l'incorporation d'objets dans des entités et la façon dont une l'objet devient une partie de, et dépend de, l'entité qui l'intègre. Nous expliquons maintenant comment faire plus avec les objets incorporés, et comment ils peuvent contenir plus que juste mappages de base.

Mappages intégrés avancés

Les objets incorporés peuvent incorporer d'autres objets, avoir des collections d'éléments de base ou types intégrables et avoir des relations avec des entités. Tout cela est possible sous le hypothèse selon laquelle les objets incorporés dans d'autres objets incorporés sont toujours dépendants sur l'entité d'intégration. De même, lorsque des relations bidirectionnelles existent dans un objet incorporé, ils sont traités comme s'ils existaient dans l'entité propriétaire, et le l'entité cible pointe vers l'entité propriétaire, pas vers l'objet incorporé.

À titre d'exemple, ramenons nos objets Employé et Adresse incorporés de Chapitre 4 et mettez un peu à jour le modèle. Insérer un objet ContactInfo, contenant l'adresse ainsi que les informations de téléphone, dans chaque employé. Au lieu d'avoir un attribut d'adresse, notre entité Employee aurait maintenant un attribut nommé contactInfo, de type ContactInfo, annoté avec @Embedded. Le modèle est illustré à la figure 10-1.

Épisode 440

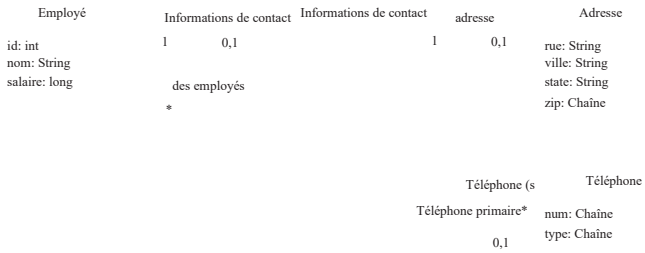

Figure 10-1. *Embeddables imbriqués avec relations*

La classe ContactInfo contient un objet incorporé, ainsi que certaines relations, et serait annoté comme indiqué dans l'extrait 10-6.

Annonce 10-6. Classe ContactInfo intégrable

```
@Embeddable @Access (AccessType.FIELD)
public class ContactInfo {
    @Embedded
    Adresse privée de résidence;

    @ManyToOne
    @JoinColumn (nom = "PRI_NUM")
    téléphone privé primaryPhone;

    @ManyToMany @MapKey (nom = "type")
    @JoinTable (nom = "EMP_PHONES")
    téléphones Map <String, Phone> privés;
    // ...
}
```

La classe Address reste la même que dans l'extrait 4-26 du chapitre 4 , mais nous avons ajouté plus de profondeur à nos informations de contact. Dans le module intégrable ContactInfo, nous avons l'adresse en tant qu'objet incorporé imbriqué, mais nous avons également un relation unidirectionnelle avec le numéro de téléphone servant de numéro de contact principal. Une relation plusieurs-à-plusieurs bidirectionnelle avec les numéros de téléphone de l'employé avoir une table de jointure par défaut nommée EMPLOYEE_PHONE, et du côté du téléphone la relation

Épisode 441

L'attribut ferait référence à une liste d'instances Employee, l'élément mappedBy étant le nom qualifié de l'attribut de relation incorporé. Par qualifié, nous entendons que il doit d'abord contenir l'attribut dans Employee qui contient également l'incorporable en tant que caractère séparateur de points (.) et l'attribut de relation dans l'incorporable. Référencement 10-7 montre la classe Phone et son mappage vers l'entité Employee.

Annonce 10-7. Classe de téléphone faisant référence à l'attribut intégré

```
@Entité
Téléphone public class {
    @Id private String num;

    @ManyToMany (mappedBy = "contactInfo.phones")
    Liste privée des employés <Employee>;

    type de chaîne privé;
    // ...
}
```

Une condition concernant les types incorporables est que si un objet incorporé fait partie d'un collection d'éléments, l'objet incorporé dans la collection ne peut inclure que mappages où la clé étrangère est stockée dans la table source. Il peut contenir des relations, telles que un-à-un et plusieurs-à-un, mais il ne peut pas contenir un-à-plusieurs ou des relations plusieurs-à-plusieurs où la clé étrangère est dans la table cible ou dans un table de jointure. De même, il ne peut pas contenir d'autres mappages basés sur une table de collection comme element collections.

Remplacement des relations intégrées

Quand nous avons introduit les éléments intégrés pour la première fois dans Chapter 4 , nous avons montré à quel point les types peuvent être réutilisés en étant incorporés dans plusieurs classes d'entités. Même si l'état est mappé dans l'incorporable, l'entité d'intégration peut les remplacer mappages en utilisant @AttributeOverride pour redéfinir la façon dont l'état incorporé est mappé dans cette table d'entité particulière. Maintenant que nous utilisons des relations à l'intérieur embeddables, @AttributeOverride ne suffit pas. Pour remplacer la façon dont une relation est

mappé, nous devons utiliser @AssociationOverride, qui nous permet de
remplacer les colonnes de jointure de relation et joindre des tables.

Épisode 442

Avant d'examiner un exemple de remplacement d'un élément intégrable avec une relation, pensons d'abord à la réutilisabilité d'un tel objet. Si une relation entre l'entité A et l'entité B est définie dans l'incorporable de type E, alors soit la relation appartient par A et la clé étrangère est dans la table correspondant à A (ou dans une table de jointure appartenant à A), ou il appartient à B et la clé étrangère va être dans la table de B (ou une table de jointure appartenant à B). Si elle appartient à B, alors la clé étrangère sera à la table de A, et il n'y aurait aucun moyen de utilisez E dans n'importe quelle autre entité car la clé étrangère serait dans la mauvaise table. De même, si la relation était bidirectionnelle, l'attribut en B serait de type A (ou collection de A) et ne pouvait pas faire référence à un objet d'un autre type. On peut comprendre, par conséquent, que seuls les éléments intégrables avec des relations appartenant à l'entité source, A, et qui sont unidirectionnels, peuvent être réutilisés dans d'autres entités.

Supposons que la relation plusieurs-à-plusieurs dans ContactInfo était unidirectionnelle et Phone n'avait pas de référence à l'employé qui a intégré ContactInfo. Nous pourrions souhaitez également intégrer des instances de ContactInfo dans une entité client. Le consommateur table, cependant, peut avoir une colonne de clé étrangère PRI_CONTACT au lieu de PRI_NUM, et de Bien sûr, nous ne pourrions pas partager la même table de jointure pour l'employé et le client relations avec le téléphone. La classe Customer résultante est affichée dans la liste10-8.

Annonce 10-8. ContactInfo d'intégration de classe client

```
@Entité
Client de classe publique {
    @Id int id;

    @Embedded
    @AttributeOverride (nom = "adresse.zip", colonne = @ Colonne (nom = "ZIP"))
    @AssociationOverrides ({
        @AssociationOverride (nom = "primaryPhone",
                              joinColumns = @ JoinColumn (nom = "EMERG_PHONE")),
        @AssociationOverride (nom = "téléphones",
                              joinTable = @ JoinTable (name = "CUST_PHONE"))})
    contactInfo privé contactInfo;

    // ...
}
```


Épisode 443

Nous pouvons remplacer l'attribut zip dans l'adresse qui est intégrée dans contactInfo en utilisant @AttributeOverride et en accédant à l'attribut dans le objet Address intégré.

Parce que nous remplaçons deux associations, nous devons utiliser la variante plurielle de

@AssociationOverrides. Notez que s'il n'y avait pas eu de table de jointure explicitement spécifiée
pour l'attribut téléphones, le nom de la table de jointure par défaut aurait été différent
en fonction de l'entité intégrant ContactInfo. Depuis le nom par défaut
se compose en partie du nom de l'entité propriétaire, la table rejoignant le salarié
L'entité à l'entité Phone aurait été définie par défaut sur EMPLOYEE_PHONE, alors que dans Customer
la table de jointure aurait été définie par défaut sur CUSTOMER_PHONE.

Astuce, il n'existe actuellement aucun moyen de remplacer la table de collection pour un élément collection dans un intégrable.

Clés primaires composées

Dans certains cas, une entité doit avoir une clé primaire ou un identifiant composé de
plusieurs champs, ou du point de vue de la base de données, la clé primaire de sa table est constituée
de plusieurs colonnes. Ceci est plus courant pour les bases de données héritées et se produit également lorsqu'un
La clé primaire est composée d'une relation, un sujet que nous aborderons plus loin dans ce chapitre.

Il existe deux options disponibles pour avoir des clés primaires composées dans une entité,
en fonction de la structure de la classe d'entité. Les deux nécessitent l'utilisation d'un
classe séparée contenant les champs de clé primaire appelés classe de clé primaire; la différence
entre les deux options est déterminé par ce que contient la classe d'entité.

Les classes de clé primaire doivent inclure des définitions de méthode pour equals () et hashCode ()
afin de pouvoir être stockés et saisis par le fournisseur de persistance, et leurs champs
ou les propriétés doivent faire partie de l'ensemble des types d'identificateurs valides répertoriés dans le chapitre précédent.
Ils doivent également être publics, implémenter Serializable et avoir un constructeur sans argument.

À titre d'exemple de clé primaire composée, nous regardons à nouveau l'entité Employee,
cette fois seulement, le numéro d'employé est spécifique au pays où il travaille. Deux
les employés de différents pays peuvent avoir le même numéro d'employé, mais un seul peut
être utilisé dans un pays donné. La figure 10-2 montre le tableau EMPLOYÉ structuré avec
une clé primaire composée pour capturer cette exigence. Compte tenu de cette définition de table, nous allons
regardez maintenant comment mapper l'entité Employee en utilisant les deux styles différents de clé primaire
classe.

Épisode 444

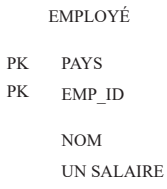

Figure 10-2. *Table EMPLOYEE avec une clé primaire composée*

Classe d'identification

Le premier et le plus élémentaire type de classe de clé primaire est une classe ID. Chaque champ de l'entité
qui constitue la clé primaire est marquée de l'annotation @Id. La classe de clé primaire
est défini séparément et associé à l'entité en utilisant l'annotation @IdClass sur
la définition de la classe d'entité. Le listing 10-9 montre une entité avec un primaire composé
clé qui utilise une classe ID. La classe d'ID d'accompagnement est affichée dans la liste10-5 .

Annonce 10-9. Utilisation d'une classe d'identification

```
@Entité
@IdClass (EmployeeId.class)
```

```
Employé de classe publique {
        @Id private String country;
        @Id
        @Column (nom = "EMP_ID")
        id int privé;
        nom de chaîne privé;
        long salaire privé;
        // ...
}
```

La classe de clé primaire doit contenir des champs ou des propriétés qui correspondent au primaire attributs clés de l'entité à la fois en nom et en type. Référencement10-10 montre le EmployeeId classe de clé primaire. Il comporte deux champs, l'un pour représenter le pays et l'autre pour représenter le numéro d'employé. Nous avons également fourni les méthodes equals () et hashCode () pour permettre la classe à utiliser dans les opérations de tri et de hachage.

Épisode 445

Annonce 10-10. La classe d'identifiant EmployeeId

```
Public class EmployeeId implémente Serializable {
        pays String privé;
        id int privé;

        public EmployeeId () {}
        public EmployeeId (String country, int id) {
            this.country = pays;
            this.id = id;
        }

        public String getCountry () {pays de retour; }
        public int getId () {identifiant de retour; }

        public boolean equals (Object o) {
                return ((o instanceof EmployeeId) &&
                                country.equals (((EmployeeId) o) .getCountry ()) &&
                                id == ((EmployeeId) o) .getId ());
        }

        public int hashCode () {
                return country.hashCode () + id;
        }
}
```

Notez qu'il n'existe aucune méthode setter sur la classe EmployeeId. Une fois qu'il a été construit à l'aide des valeurs de clé primaire, il ne peut pas être modifié. Nous faisons cela pour appliquer le notion qu'une valeur de clé primaire ne peut pas être modifiée, même lorsqu'elle est composée de plusieurs des champs. Étant donné que l'annotation @Id a été placée sur les champs de l'entité, le fournisseur utilisez également l'accès aux champs lorsqu'il a besoin de travailler avec la classe de clé primaire.

La classe ID est utile en tant qu'objet structuré qui encapsule tous les principaux information clé. Par exemple, lorsque vous effectuez une requête basée sur la clé primaire, comme méthode find () de l'interface EntityManager, une instance de la classe ID peut être utilisé comme argument au lieu d'une collection non structurée et non ordonnée de données de clé primaire. Le listing 10-11 montre un extrait de code qui recherche un employé avec un nom de pays et un numéro d'employé donnés. Une nouvelle instance de la classe EmployeeId est construit à l'aide des arguments de la méthode puis utilisé comme argument de find () méthode.

Épisode 446

Annonce 10-11. Appel d'une requête de clé primaire sur une entité avec une classe d'ID

```
EmployeeId id = new EmployeeId (pays, id);
Employé emp = em.find (Employee.class, id);
```

Astuce car l'argument de find () est de type Object, les fournisseurs peuvent prendre en charge transmettre des tableaux simples ou des collections d'informations de clé primaire. qui passe les arguments qui ne sont pas des classes de clé primaire ne sont pas portables.

Classe d'identification intégrée

Une entité qui contient un seul champ du même type que la classe de clé primaire est dite pour utiliser une classe d'ID intégrée. La classe ID intégrée n'est qu'un objet intégré qui se trouve être composé des composants clés primaires. Nous utilisons un @EmbeddedId annotation pour indiquer qu'il ne s'agit pas simplement d'un objet incorporé normal, mais également d'un classe clé. Lorsque nous utilisons cette approche, il n'y a pas d'annotations @Id sur la classe, ni l'annotation @IdClass utilisée. Vous pouvez considérer @EmbeddedId comme l'équivalent logique de mettre à la fois @Id et @Embedded sur le terrain.

Comme les autres objets incorporés, la classe d'ID incorporée doit être annotée avec @Embeddable, mais le type d'accès peut différer de celui de l'entité qui l'utilise. Référencement 10-12 montre à nouveau la classe EmployeeId, cette fois en tant que clé primaire intégrable classe. Les méthodes getter, implémentations equals () et hashCode (), sont les mêmes que la version précédente du Listing 10-10.

Annonce 10-12. Classe de clé primaire intégrable

```
@Embeddable
public class EmployeeId {
    pays String privé;
    @Column (nom = "EMP_ID")
    id int privé;

    public EmployeeId () {}
    public EmployeeId (String country, int id) {
        this.country = pays;
```


Épisode 447


```
        this.id = id;
    }

    // ...
}
```

L'utilisation de la classe de clé primaire intégrée n'est pas différente de l'utilisation d'une classe de clé primaire intégrée type, sauf que l'annotation utilisée sur l'attribut est @EmbeddedId au lieu de @Embedded. Le Listing 10-13 montre l'entité Employee ajustée pour utiliser le

version de la classe EmployeeId. Notez que puisque les mappages de colonnes sont présents sur le Embedded type, nous ne spécifions pas le mappage pour EMP_ID comme cela a été fait dans le cas du Classe d'identification. Si la classe de clé primaire incorporée est utilisée par plusieurs entités, alors le L'annotation @AttributeOverride peut être utilisée pour personnaliser les mappages comme vous le feriez pour un type intégré standard. Pour renvoyer les attributs country et id de la clé primaire à partir des méthodes getter, nous devons déléguer à l'objet ID intégré pour obtenir les valeurs.

Annonce 10-13. Utilisation d'une classe d'ID intégrée

```
@Entité
Employé de classe publique {
    @EmbeddedId identifiant privé EmployeeId;
    nom de chaîne privé;
    long salaire privé;

    employé public () {}
    employé public (chaîne country, int id) {
        this.id = new EmployeeId (pays, id);
    }

    public String getCountry () {return id.getCountry (); }
    public int getId () {return id.getId (); }
    // ...
}
```

Nous pouvons créer une instance de EmployeeId et la passer à la méthode find () tout comme nous fait pour l'exemple de classe ID, mais, si nous voulons créer la même requête en utilisant JP QL et référencer la clé primaire, nous devons parcourir explicitement la classe d'ID intégrée. Référencement 10-14 montre cette technique. Même si l'id n'est pas une relation, nous traversons toujours il en utilisant la notation par points afin d'accéder aux membres de la classe incorporée.

Épisode 448

Annonce 10-14. Référencement d'une classe d'ID intégrée dans une requête

```
public Employee findEmployee (String country, int id) {
    retour (employé)
        em.createQuery ("SELECT e" +
                        "FROM Employé e" +
                        "O e.id.country =? 1 ET e.id.id =? 2")
          .setParameter (1, pays)
          .setParameter (2, id)
          .getSingleResult ();
}
```

La décision d'utiliser un seul attribut d'identifiant intégré ou un groupe d'identifiant attributs, chacun mappé séparément dans la classe d'entité, se résume principalement à préférence. Certaines personnes aiment encapsuler les composants de l'identifiant dans un seul attribut d'entité du type de classe d'identificateur intégré. Le compromis est qu'il fait déréférencer une partie de l'identifiant un peu plus longtemps dans le code ou dans JP QL, bien que avoir des méthodes d'aide, comme celles de la liste 10-13 , peut vous aider.

Si vous accédez ou définissez des parties de l'identifiant individuellement, cela peut rendre plus sens de créer un attribut d'entité distinct pour chacune des parties d'identificateur constitutif. Ceci présente un modèle et une interface plus représentatifs pour l'identifiant séparé Composants. Cependant, si la plupart du temps vous référencez et transmettez identifiant en tant qu'objet, vous feriez peut-être mieux d'utiliser un identifiant intégré qui crée et stocke une seule instance de l'identifiant composite.

Identifiants dérivés

Lorsqu'un identifiant dans une entité inclut une clé étrangère vers une autre entité, on l'appelle un identifiant dérivé. Parce que l'entité contenant l'identifiant dérivé dépend de une autre entité pour son identité, la première est appelée l'entité dépendante. L'entité qui cela dépend de la cible d'une relation plusieurs-à-un ou un-à-un du entité dépendante, et s'appelle l'entité parente. La figure 10-3 montre un exemple de données modèle pour les deux types d'entités, avec la table DEPARTMENT représentant l'entité mère et table PROJECT représentant l'entité dépendante. Notez que dans cet exemple, il est une colonne de clé primaire de nom supplémentaire dans PROJECT, ce qui signifie que le L'entité de projet a un attribut d'identificateur qui ne fait pas partie de sa relation avec le service.

Épisode 449

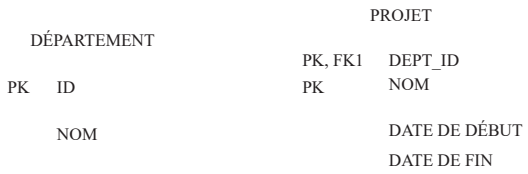

Figure 10-3. *Tables d'entités dépendantes et parentes*

L'objet dépendant ne peut pas exister sans clé primaire, et depuis cette clé primaire se compose de la clé étrangère de l'entité mère, il doit être clair qu'une nouvelle l'entité ne peut pas être persistante sans que la relation avec l'entité mère ne soit établie. Il n'est pas défini de modifier la clé primaire d'une entité existante, donc le one-to-one ou la relation plusieurs-à-un qui fait partie d'un identifiant dérivé est également immuable et ne doit pas être réaffecté à une nouvelle entité une fois que l'entité dépendante a été conservée ou existe déjà.

Nous avons passé les dernières sections à discuter de différents types d'identifiants, et vous pourrait repenser à ce que vous avez appris et réaliser qu'il existe un certain nombre de paramètres susceptibles d'affecter la manière dont un identificateur dérivé peut être configuré. Par exemple, l'identifiant dans l'une ou l'autre des entités peut être composé d'un ou de plusieurs les attributs. La relation entre l'entité dépendante et l'entité parente peut faire jusqu'à l'identifiant dérivé entier, ou, comme dans la figure 10-3 , il peut y avoir un état supplémentaire dans l'entité dépendante qui y contribue. L'une des entités pourrait avoir un simple ou clé primaire composée, et dans le cas composé peut avoir une classe ID ou un classe d'identification intégrée. Tous ces facteurs se combinent pour produire une multitude de scénarios, dont chacun nécessite des configurations légèrement différentes. Les règles de base pour les les identifiants sont décrits en premier, avec des descriptions plus détaillées dans ce qui suit sections.

Règles de base pour les identificateurs dérivés

La plupart des règles relatives aux identificateurs dérivés peuvent être résumées en quelques déclarations générales, même si l'application des règles ensemble n'est peut-être pas aussi simple. Nous passons par certains des cas plus tard pour les expliquer, et même montrer un cas d'exception ou deux pour le garder

Épisode 450

intéressant, mais pour jeter les bases de ces cas d'utilisation, les règles peuvent être définies comme suit:

- Une entité dépendante peut avoir plusieurs entités parentes (c.-à-d. identifiant peut inclure plusieurs clés étrangères).
- Une entité dépendante doit avoir toutes ses relations avec les entités parentes défini avant de pouvoir être conservé.
- Si une classe d'entité a plusieurs attributs d'ID, non seulement elle doit utiliser une classe d'ID, mais il doit également y avoir un attribut correspondant du même nom dans la classe ID que chacun des attributs ID de l'entité.
- Les attributs d'ID d'une entité peuvent être d'un type simple ou d'une entité type qui est la cible d'une relation plusieurs-à-un ou un-à-un.
- Si un attribut ID dans une entité est de type simple, alors le type de l'attribut correspondant dans la classe ID doit être du même type simple.
- Si un attribut ID dans une entité est une relation, alors le type de l'attribut correspondant dans la classe ID est du même type que l'attribut principal type de clé de l'entité cible dans la relation (si le le type de clé est un type simple, une classe d'ID ou une classe d'ID intégrée).
- Si l'identifiant dérivé d'une entité dépendante est sous la forme une classe d'ID intégrée, puis chaque attribut de cette classe d'ID qui représente une relation doit être référencée par un @MapsId annotation sur l'attribut de relation correspondant.

Les sections suivantes décrivent comment ces règles peuvent être appliquées.

Clé primaire partagée

Un cas simple, quoique un peu moins courant, est celui où l'identifiant dérivé est composé de un attribut unique qui est la clé étrangère de relation. À titre d'exemple, supposons qu'il y ait eu un relation bidirectionnelle un-à-un entre les entités Employee et EmployeeHistory. Puisqu'il n'y a qu'un seul historique d'employé par employé, nous pourrions décider de partager la clé primaire. Dans le Listing 10-15 , si EmployeeHistory est l'entité dépendante, puis on indique que la relation clé étrangère est l'identifiant en annotant le relation avec @Id.

Épisode 451

Liste 10-15. Identifiant dérivé avec un seul attribut

```
@Entité
public class EmployeeHistory {
    // ...
    @Id
    @Un par un
    @JoinColumn (nom = "EMP_ID")
    employé privé;
    // ...
}
```

Le type de clé primaire de EmployeeHistory sera du même type
en tant qu'employé, donc si l'employé a un identifiant entier simple, l'identifiant de
EmployeeHistory sera également un entier. Si l'employé a une clé primaire composée,
soit avec une classe d'ID ou une classe d'ID intégrée, alors EmployeeHistory va partager
la même classe d'ID (et doit également être annotée avec l'annotation @IdClass). le
le problème est que cela trébuche sur la règle de classe ID selon laquelle il devrait y avoir un attribut correspondant
dans l'entité pour chaque attribut de sa classe ID. C'est l'exception à la règle, en raison de
le fait même que la classe ID soit partagée entre les entités parent et dépendantes.

Parfois, quelqu'un peut souhaiter que l'entité contienne un attribut de clé primaire comme
ainsi que l'attribut de relation, avec les deux attributs mappés sur la même clé étrangère
colonne du tableau. Même si l'attribut de clé primaire n'est pas nécessaire dans l'entité,
certaines personnes voudront peut-être le définir séparément pour un accès plus facile. Malgré le fait que le
deux attributs correspondent à la même colonne de clé étrangère (qui est également la colonne de clé primaire),
le mappage n'a pas à être dupliqué aux deux endroits. L'annotation @Id est placée
sur l'attribut identifiant et @MapsId annote l'attribut de relation pour indiquer que
il mappe également l'attribut ID. Ceci est montré dans la liste10-16. Notez que physique
les annotations de mappage (par exemple, @Column) ne doivent pas être spécifiées sur l'attribut empId car
@MapsId indique que l'attribut de relation est l'endroit où le mappage se produit.

Annonce 10-16. Identificateur dérivé avec mappages partagés

```
@Entité
public class EmployeeHistory {
      // ...
      @Id
      int empId;
```

Épisode 452


```
      @MapsId
      @Un par un
      @JoinColumn (nom = "EMP_ID")
      employé privé;
      // ...
}
```

Il y a quelques points supplémentaires à mentionner à propos de @MapsId, avant que nous
passez aux identificateurs dérivés avec plusieurs attributs mappés.

Le premier point est en réalité une suite logique au fait que la relation annotée
with @MapsId définit également le mappage pour l'attribut identificateur. Si il n'y a pas
remplacer l'annotation @JoinColumn sur l'attribut de relation, puis la colonne de jointure
sera défini par défaut selon les règles de défaut habituelles. Si tel est le cas, alors le
L'attribut identifiant sera également mappé à cette même valeur par défaut. Par exemple, si le
L'annotation @JoinColumn a été supprimée du listing 10-16, puis l'employé et
les attributs empId seraient mappés à la colonne de clé étrangère EMPLOYEE_ID par défaut
(en supposant que la colonne de clé primaire dans la table EMPLOYEE était ID).

Deuxièmement, même si l'attribut identifier partage le mappage de base de données défini
sur l'attribut de relation, du point de vue de l'attribut identificateur, c'est vraiment un
mappage en lecture seule. Les mises à jour ou les insertions dans la colonne de clé étrangère de la base de données ne seront jamais
se produisent via l'attribut de relation. C'est l'une des raisons pour lesquelles vous devez toujours
n'oubliez pas de définir les relations parentes avant d'essayer de conserver une entité dépendante.

Remarque: n'essayez pas de définir uniquement l'attribut identifiant (et non la relation
attribut) comme moyen de raccourcir la persistance d'une entité dépendante. Certains fournisseurs
peut avoir un support spécial pour faire cela, mais cela ne causera pas
clé à écrire dans la base de données.

L'attribut identifiant sera renseigné automatiquement par le fournisseur lorsqu'un
l'instance d'entité est lue à partir de la base de données, ou supprimée / validée. Cependant, il ne peut pas être
supposé être présent lors du premier appel de persist () sur une instance sauf si l'utilisateur le définit
explicitement.

Épisode 453

Attributs mappés multiples

Un cas plus courant est probablement celui dans lequel l'entité dépendante a un identifiant
cela inclut non seulement une relation, mais aussi un état qui lui est propre. Nous utilisons l'exemple
illustré à la Figure 10-3, où un projet a un identifiant composé composé d'un nom
et une clé étrangère du département qui la gère. L'identifiant unique étant le
combinaison de son nom et de son département, aucun département ne serait autorisé à créer
plusieurs projets portant le même nom. Cependant, deux départements différents peuvent
choisissez le même nom pour leurs propres projets. Référencement10-17 illustre le mappage trivial
de l'identificateur de projet en utilisant @Id sur les attributs name et dept.

Annonce 10-17. Projet avec identifiant dépendant

```
@Entité
@IdClass (ProjectId.class)
Projet de classe publique {

    @Id nom de chaîne privé;

    @Id
    @ManyToOne
    @JoinColumns ({
        @JoinColumn (nom = "DEPT_NUM",
                     referencedColumnName = "NUM"),
        @JoinColumn (nom = "DEPT_CTY",
                     referencedColumnName = "COUNTRY")})
    département privé;
    // ...
}
```

L'identifiant composé signifie que nous devons également spécifier la classe de clé primaire
en utilisant l'annotation @IdClass. Rappelez-vous notre règle selon laquelle les classes de clé primaire doivent avoir un
correspondant à l'attribut nommé pour chacun des attributs d'ID sur l'entité, et généralement le
les attributs doivent également être du même type. Cependant, cette règle ne s'applique que lorsque le
les attributs sont de types simples et non de types d'entités. Si @Id annote une relation, alors
cet attribut de relation sera d'un type d'entité cible et l'extension de règle
est que l'attribut de classe de clé primaire doit être du même type que la clé primaire du

Épisode 454

entité cible. Cela signifie que la classe ProjectId spécifiée comme classe ID pour Project dans le Listing 10-17 doit avoir un attribut nommé name, de type String, et un autre nommé dept qui sera du même type que la clé primaire du département. Si le ministère a un clé primaire entière simple, alors l'attribut dept dans ProjectId sera de type int, mais si Department a une clé primaire composée avec sa propre classe de clé primaire, par exemple DeptId, alors l'attribut dept dans ProjectId serait de type DeptId, comme indiqué dans la liste 10-18 .

Annonce 10-18. Classes d'ID ProjectId et DeptId

```
La classe publique ProjectId implémente Serializable {
        nom de chaîne privé;
        DeptId privé;

        public ProjectId () {}
        Public ProjectId (DeptId deptId, String name) {
                this.dept = deptId;
                this.name = nom;
        }
       // ...
}

La classe publique DeptId implémente Serializable {
        numéro int privé;
        pays String privé;

        public DeptId () {}
        public DeptId (nombre entier, chaîne de pays) {
                this.number = nombre;
                this.country = pays;
        }
        // ...
}
```

Utilisation d'EmbeddedId

Il est également possible d'avoir un identifiant dérivé lorsque l'un ou l'autre (ou les deux) entités utilise @EmbeddedId. Lorsque la classe ID est intégrée, la non-relation les attributs d'identifiant sont mappés dans la classe d'ID intégrable, comme d'habitude, mais le

les attributs de la classe d'ID intégrés qui correspondent aux relations sont mappés par les attributs de relation dans l'entité. Le Listing 10-19 montre comment l'identifiant dérivé est mappé dans la classe Project lorsqu'une classe d'ID incorporée est utilisée. Nous annotons l'attribut de relation avec @MapsId ("dept"), indiquant qu'il spécifie également le mappage de l'attribut dept de la classe d'ID intégrée. L'attribut dept de ProjectId est du même type de clé primaire que Department dans le Listing 10-20.

Notez que nous avons utilisé plusieurs colonnes de jointure sur la relation de service parce que Department a une clé primaire composée. Le mappage d'identifiants en plusieurs parties est expliqué plus en détail dans la section "Colonnes de jointure composées" plus loin dans le chapitre.

Annonce 10-19. Project et classe ProjectId intégrée

```
@Entité
Projet de classe publique {

        @EmbeddedId ID de ProjectId privé;
```

```
    @MapsId ("dept")
    @ManyToOne
    @JoinColumns ({
        @JoinColumn (name = "DEPT_NUM", referencedColumnName = "NUM"),
        @JoinColumn (name = "DEPT_CTRY", referencedColumnName = "CTRY")})
    département du département privé;
    // ...
}

@Embeddable
La classe publique ProjectId implémente Serializable {
    @Column (nom = "P_NAME")
    nom de chaîne privé;
    @Embedded
    DeptId privé;

    // ...
}
```

L'entité Department a un identifiant intégré, mais ce n'est pas un identifiant dérivé car la classe DeptId ID n'a aucun attribut correspondant à la relation attributs dans Department. Les annotations @Column dans la classe DeptId mappent l'identificateur

Épisode 456

champs dans l'entité Department, mais lorsque DeptId est incorporé dans ProjectId, ces les mappages de colonnes ne s'appliquent pas. Une fois l'attribut dept mappé par le département relation dans Project, les annotations @JoinColumn sur cette relation sont utilisées comme les mappages de colonnes pour la table PROJECT.

Liste 10-20. Département et classe DeptId intégrée

```
@Entité
Département de classe publique {
    @EmbeddedId ID privé DeptId;

    @OneToMany (mappedBy = "département")
    Projets <Project> de liste privée;

    // ...
}

@Embeddable
La classe publique DeptId implémente Serializable {
    @Column (nom = "NUM")
    numéro int privé;
    @Column (nom = "CTRY")
    pays String privé;
    // ...
}
```

Si la classe Department avait une clé primaire simple, par exemple un long au lieu d'un ID class, alors l'attribut dept dans ProjectId serait simplement le type de clé primaire simple de Department (le type long), et il n'y aurait qu'une seule colonne de jointure sur le plusieurs-à-un attribut de service dans Project.

ALTERNATIVE AUX IDENTIFIANTS DÉRIVÉS

L'annotation @MapsId et la possibilité d'appliquer @Id aux attributs de relation ont été introduites

dans jpa 2.0 pour améliorer la situation qui existait dans jpa 1.0. à ce moment-là, seul le one-to-one
Le scénario de clé primaire partagée a été spécifié à l'aide de l'annotation @PrimaryKeyJoinColumn
(l'utilisation de l'annotation @Id est la méthode préférée et recommandée à l'avenir).

Épisode 457

bien qu'il n'y ait pas de moyen spécifié de résoudre le cas général de l'inclusion d'une clé étrangère dans un identifiant, il était généralement soutenu par la pratique consistant à ajouter un ou plusieurs (redondants) vers l'entité dépendante. chaque champ ajouté contiendrait une clé étrangère vers le entité associée, et, car le champ ajouté et la relation seraient mappés au même (s) colonne (s) de jointure, l'une ou l'autre des deux devrait être marquée comme lecture seule (voir section "Mappages en lecture seule"), ou non actualisable ou insérable. l'exemple suivant montre comment le listing 10-17 serait fait en utilisant jpa 1.0. la classe id serait la même. Depuis le Les attributs deptNumber et deptCountry sont des attributs d'identifiant et ne peuvent pas être modifiés dans la base de données, il n'est pas nécessaire de définir leur possibilité de mise à jour sur false.

```
@Entité
@IdClass (ProjectId.class)
Projet de classe publique {
    @Id nom de chaîne privé;

    @Id
    @Column (nom = "DEPT_NUM", insérable = faux)
    private int deptNumber;

    @Id
    @Column (nom = "DEPT_CTRY", insertable = false)
    private String deptCountry;

    @ManyToOne
    @JoinColumns ({
        @JoinColumn (name = "DEPT_NUM", referencedColumnName = "NUM"),
        @JoinColumn (name = "DEPT_CTRY", referencedColumnName = "CTRY")})
    département du département privé;
    // ...
}
```

Éléments de mappage avancés

Des éléments supplémentaires peuvent être spécifiés sur les annotations @Column et @JoinColumn (et leurs parents @MapKeyColumn, @MapKeyJoinColumn et @OrderColumn), certains qui s'appliquent à la génération de schémas abordée au chapitre 14. D'autres pièces que nous pouvons décrire séparément comme s'appliquant aux colonnes et joindre des colonnes dans les sections suivantes.

Épisode 458

Mappages en lecture seule

JPA ne définit vraiment aucun type d'entité en lecture seule, même si elle apparaîtra probablement

dans une prochaine version. L'API définit cependant des options pour définir des mappages individuels
être en lecture seule à l'aide des éléments insérables et actualisables de @Column et
Annotations @JoinColumn. Ces deux paramètres par défaut sont true mais peuvent être définis sur false si
nous voulons nous assurer que le fournisseur n'insérera ni ne mettra à jour les informations dans le tableau de
réponse aux modifications de l'instance d'entité. Si les données de la table mappée existent déjà
et nous voulons nous assurer qu'il ne sera pas modifié au moment de l'exécution, alors l'insertable et
les éléments pouvant être mis à jour peuvent être définis sur false, empêchant ainsi le fournisseur de faire
autre chose que la lecture de l'entité à partir de la base de données. Référencement10-21 démontre le
Entité d'employé avec mappages en lecture seule.

Annonce 10-21. Rendre une entité en lecture seule

```
@Entité
Employé de classe publique {
    @Id
    @Colonne (insérable = faux)
    id int privé;
    @Column (insertable = false, updatable = false)
    nom de chaîne privé;
    @Column (insertable = false, updatable = false)
    long salaire privé;

    @ManyToOne
    @JoinColumn (name = "DEPT_ID", insertable = false, updatable = false)
    département du département privé;
    // ...
}
```

Nous n'avons pas à nous soucier de la modification du mappage d'identifiant, car il est
illégal de modifier les identifiants. Les autres mappages, cependant, sont marqués comme ne pouvant pas
être inséré ou mis à jour, donc nous supposons qu'il y a déjà des entités dans la base de données
à lire et à utiliser. Aucune nouvelle entité ne sera conservée et les entités existantes ne le seront jamais
être mis à jour.

Épisode 459

Notez que cela ne garantit pas que l'état de l'entité ne changera pas en mémoire.
Les instances d'employés peuvent toujours être modifiées à l'intérieur ou à l'extérieur d'une transaction, mais à
l'heure de validation de la transaction ou chaque fois que les entités sont vidées dans la base de données, cet état
ne sera pas enregistré et le fournisseur ne lèvera probablement pas d'exception pour l'indiquer. Être
modifiez avec soin les mappages en lecture seule en mémoire, car la modification des entités peut
les amener à devenir incompatibles avec l'état de la base de données et pourraient faire des ravages
sur un cache spécifique au fournisseur.

Même si tous ces mappages ne peuvent pas être mis à jour, l'entité dans son ensemble pourrait
être toujours supprimé. Une fonction de lecture seule appropriée résoudra ce problème une fois pour toutes
une version future, mais en attendant, certains fournisseurs prennent en charge la notion de lecture seule
entités, et peut en optimiser le traitement dans leurs caches et leur contexte de persistance
implémentations.

Optionalité

Comme vous le verrez au chapitre 14 lorsque nous parlerons de génération de schémas, il existe des métadonnées
qui autorise les colonnes de la base de données à être nulles ou les oblige à avoir des valeurs.
Bien que ce paramètre affecte le schéma de base de données physique, il existe également des paramètres sur
certains des mappages logiques qui permettent un mappage de base ou une association à valeur unique

le mappage doit être laissé vide ou doit être spécifié dans le modèle d'objet. L'élément qui requiert ou autorise un tel comportement est l'élément facultatif dans @Basic, Annotations @ManyToOne et @OneToOne.

Lorsque l'élément facultatif est spécifié comme faux, il indique au fournisseur que le mappage de champ ou de propriété peut ne pas être nul. L'API ne définit pas réellement ce que Le comportement est dans le cas où la valeur est nulle, mais le fournisseur peut choisir de lancer un exception ou simplement faire autre chose. Pour les mappages de base, ce n'est qu'un indice et peut être complètement ignoré. L'élément facultatif peut également être utilisé par le fournisseur lors de l'exécution génération de schéma, car si facultatif est défini sur true, la colonne de la base de données doit également être nullable.

Parce que l'API n'entre dans aucun détail sur l'ordinalité du modèle objet, il y a un certain degré de non-portabilité associé à son utilisation. Un exemple de définir le gestionnaire comme un attribut obligatoire est indiqué dans la liste 10-22 . Le défaut value for optional est true, ce qui oblige à ne spécifier que si une valeur false est nécessaire.

Épisode 460

Annonce 10-22. Utilisation de mappages facultatifs

```
@Entité
Employé de classe publique {
    // ...
    @ManyToOne (facultatif = false)
    @JoinColumn (name = "DEPT_ID", insertable = false, updatable = false)
    département du département privé;
    // ...
}
```

Relations avancées

Si vous êtes en mesure de démarrer à partir d'une application Java et de créer un schéma de base de données, vous avez alors un contrôle complet sur ce à quoi le schéma ressemble et comment vous mappez les classes à la base de données. Dans ce cas, il est probable que vous n'aurez pas besoin pour utiliser de très nombreuses fonctionnalités de relation avancées proposées par l'API. le la flexibilité d'être en mesure de définir un modèle de données rend généralement configuration du mappage. Cependant, si vous êtes dans la situation malheureuse de cartographier un Modèle Java à une base de données existante, puis pour contourner le schéma de données, vous peut avoir besoin d'accéder à plus de mappages que ceux dont nous avons discuté jusqu'à présent. Les cartographies décrits dans les sections suivantes concernent principalement le mappage vers des bases de données héritées, et seront le plus souvent utilisés car ils sont la seule option. Une exception notable est le fonction de suppression des orphelins, utilisée pour modéliser une relation parent-enfant.

Utilisation des tables de jointure

Nous avons déjà vu des mappages tels que le plusieurs-à-plusieurs et unidirectionnel un-à-de nombreux mappages qui utilisent des tables de jointure. Parfois, un schéma de base de données utilise une table de jointure pour associer deux types d'entités, même si la cardinalité de l'entité cible dans la relation est une. Une relation un-à-un ou plusieurs-à-un n'a normalement pas besoin d'une table de jointure car la cible ne sera jamais qu'une seule entité et la clé étrangère peut être stockée dans la table d'entités source. Mais si la table de jointure existe déjà pour une relation plusieurs-à-un, alors, bien sûr, nous devons mapper la relation en utilisant cette table de jointure. Pour ce faire, nous avons seulement besoin ajoutez l'annotation @JoinTable au mappage de relations.

Épisode 461

Que la relation soit unidirectionnelle ou bidirectionnelle, l'annotation @JoinTable est une annotation physique et doit être définie du côté propriétaire de la relation, juste comme avec tous les autres mappages. Cependant, parce qu'une table de jointure n'est pas la configuration par défaut pour les mappages qui ne sont pas plusieurs-à-plusieurs ou unidirectionnels un-à-plusieurs, nous devons spécifiez l'annotation lorsque nous voulons qu'une table de jointure soit utilisée. Les éléments de la L'annotation @JoinTable peut toujours être utilisée pour remplacer les différents noms de schéma.

Dans la liste 10-23, nous voyons une table de jointure utilisée pour une relation plusieurs-à-un d'un employé au service. La relation peut être unidirectionnelle ou peut être bidirectionnel, avec une relation un-à-plusieurs du service à l'employé, mais dans les deux cas, le côté «plusieurs» doit toujours être le propriétaire. La raison en est que même si c'était bidirectionnel, le côté @ManyToOne ne pouvait pas être le propriétaire car il y aurait aucun moyen pour l'attribut @ManyToOne de faire référence au côté d'attribut @OneToMany propriétaire. Il n'y a pas d'élément mappedBy dans la définition d'annotation @ManyToOne.

Comme avec la plupart des autres mappages, le côté non propriétaire d'une relation bidirectionnelle ne change pas selon que la relation est mappée à l'aide d'une table de jointure ou ne pas. Il fait simplement référence à la relation propriétaire et lui permet de correspondre aux tables physiques / colonnes en conséquence.

Annonce 10-23. Mappage plusieurs à un à l'aide d'une table de jointure

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;

    @ManyToOne
    @JoinTable (nom = "EMP_DEPT")
    département du département privé;
    // ...
}
```

Éviter les tables de jointure

Jusqu'à présent, nous avons discuté d'un mappage unidirectionnel un-à-plusieurs dans le contexte d'utilisation d'une table de jointure, mais il est également possible de mapper un mappage unidirectionnel sans utiliser de table de jointure. Il nécessite que la clé étrangère soit dans la table cible, ou "plusieurs"

Épisode 462

côté de la relation, même si l'objet cible ne fait aucune référence à le côté «un». C'est ce qu'on appelle un mappage de clé étrangère cible un-à-plusieurs unidirectionnel, car la clé étrangère se trouve dans la table cible au lieu d'une table de jointure.

Pour utiliser ce mappage, nous indiquons d'abord que la relation un-à-plusieurs est unidirectionnel en ne spécifiant aucun élément mappedBy dans l'annotation. Ensuite nous spécifiez une annotation @JoinColumn sur l'attribut un-à-plusieurs pour indiquer le colonne clé. Le hic, c'est que la colonne de jointure que nous spécifions s'applique à la table

de l'objet cible, et non de l'objet source dans lequel l'annotation apparaît.

Annonce 10-24. Mappage unidirectionnel un à plusieurs à l'aide d'une clé étrangère cible

```
@Entité
Département de classe publique {
    @Id id int privé;

    @OneToMany
    @JoinColumn (nom = "DEPT_ID")
    les employés de la collection privée <Employee>;
    // ...
}
```

L'exemple du Listing 10-24 montre à quel point il est simple de mapper un unidirectionnel - mappage vers plusieurs à l'aide d'une clé étrangère cible. La colonne DEPT_ID fait référence à la table mappé par Employee et est une clé étrangère vers la table DEPARTMENT, même si le L'entité d'employé n'a aucun attribut de relation avec le service.

Avant d'utiliser ce mappage, vous devez comprendre les implications de cette opération, car ils peuvent être assez négatifs, tant du point de vue de la modélisation que de la performance perspective. Chaque ligne de la table EMPLOYEE correspond à une instance Employee, avec chaque colonne correspondant à un état ou une relation dans l'instance. Quand il y a un changement dans la ligne, il y a l'hypothèse qu'une sorte de changement s'est produite Employé correspondant, mais dans ce cas, cela ne suit pas nécessairement. L'employé peut-être juste été changé pour un département différent, et parce qu'il n'y avait pas référence au ministère de la part de l'employé, il n'y a eu aucun changement pour l'employé.

Du point de vue des performances, pensez au cas où à la fois l'état d'un L'employé est changé et le service auquel il appartient est changé. Lors de l'écriture l'état de l'employé, la clé étrangère du service n'est pas connue car l'employé l'entité n'y fait aucune référence. Dans ce cas, l'employé devra peut-être être

Épisode 463

écrit deux fois, une fois pour l'état modifié de l'employé et une deuxième fois lorsque les modifications de l'entité Department sont écrites et la clé étrangère de Employee vers Le département doit être mis à jour pour pointer vers le département qui y fait référence.

Colonnes de jointure composées

Maintenant que nous avons expliqué comment créer des entités avec des clés primaires composées, il est pas loin pour comprendre que, dès que nous avons une relation avec une entité avec un identifiant composé, nous aurons besoin d'un moyen d'étendre la façon dont nous le référencons actuellement.

Jusqu'à présent, nous avons traité le mappage des relations physiques uniquement comme une jointure colonne, mais, si la clé primaire que nous référencons est composée de plusieurs champs, alors nous aurons besoin de plusieurs colonnes de jointure. C'est pourquoi nous avons le pluriel @JoinColumns annotation qui peut contenir autant de colonnes de jointure que nous en avons besoin.

Il n'y a pas de valeurs par défaut pour les noms de colonne de jointure lorsque nous avons plusieurs jointures Colonnes. La réponse la plus simple est de demander à l'utilisateur de les attribuer, donc, lorsque plusieurs les colonnes de jointure sont utilisées, à la fois l'élément name et l'élément referencedColumnName, qui indique le nom de la colonne de clé primaire dans la table cible, doit être spécifié.

Maintenant que nous entrons dans des scénarios plus complexes, ajoutons un plus intéressant relation au mix. Disons que les employés ont des managers et que chaque manager a un certain nombre d'employés qui travaillent pour lui. Vous ne trouverez peut-être pas cela très intéressant jusqu'à ce que vous vous rendiez compte que les managers sont eux-mêmes des employés, les colonnes de jointure sont en fait auto-référentiels, c'est-à-dire faisant référence à la même table dans laquelle ils sont stockés. Figure 10-4

montre la table EMPLOYEE avec cette relation.

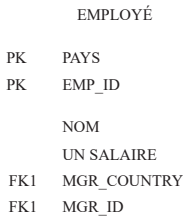

Figure 10-4. *Table EMPLOYEE avec clé étrangère composée à auto-référencement*

Épisode 464

Référencement 10-25 montre une version de l'entité Employé qui a une relation de gestionnaire, qui est plusieurs à un de chacun des employés gérés au gestionnaire, et un to-many dirige la relation du gestionnaire vers ses employés gérés.

Liste 10-25. Relations composées auto-référencées

```
@Entité
@IdClass (EmployeeId.class)
Employé de classe publique {
    @Id private String country;
    @Id
    @Column (nom = "EMP_ID")
    id int privé;

    @ManyToOne
    @JoinColumns ({
        @JoinColumn (name = "MGR_COUNTRY", referencedColumnName = "COUNTRY"),
        @JoinColumn (name = "MGR_ID", referencedColumnName = "EMP_ID")
    })
    gestionnaire d'employé privé;

    @OneToMany (mappedBy = "manager")
    Collection privée <Employee> dirige;
    // ...
}
```

N'importe quel nombre de colonnes de jointure peut être spécifié, bien qu'en pratique le soit très rarement il y en a plus de deux. La forme plurielle de @JoinColumns peut être utilisée sur plusieurs-à-un ou relations un à un ou plus généralement chaque fois que l'annotation @JoinColumn unique est valable.

Un autre exemple à considérer est la table de jointure d'une relation plusieurs-à-plusieurs. Nous pouvons revoir la relation Employé et Projet décrite au chapitre 4 à prendre en compte notre clé primaire composée dans Employee. La nouvelle structure de table pour cela relation est montrée dans la figure 10-5.

Épisode 465

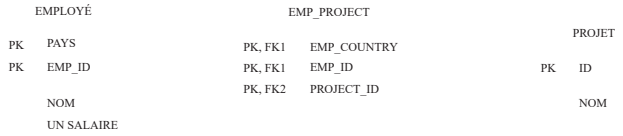

Figure 10-5. *Joindre la table avec une clé primaire composée*

Si nous conservons l'entité Employee en tant que propriétaire, où la table de jointure est définie, alors le le mappage pour cette relation sera comme indiqué dans la liste 10-26 .

Annonce 10-26. Joindre une table avec des colonnes de jointure composées

```
@Entité
@IdClass (EmployeeId.class)
Employé de classe publique {
    @Id private String country;
    @Id
    @Column (nom = "EMP_ID")
    id int privé;

    @Plusieurs à plusieurs
    @JoinTable (
        name = "EMP_PROJECT",
        joinColumns = {
            @JoinColumn (name = "EMP_COUNTRY", referencedColumnName = "COUNTRY"),
            @JoinColumn (name = "EMP_ID", referencedColumnName = "EMP_ID")},
        inverseJoinColumns = @ JoinColumn (name = "PROJECT_ID"))
    projets <Projet> de collection privée;
    // ...
}
```

Suppression des orphelins

L'élément orphanRemoval fournit un moyen pratique de modéliser parent-enfant relations, ou plus spécifiquement des relations privées. Nous différencions ces deux parce que la propriété privée est une variété particulière de parent-enfant dans laquelle l'enfant l'entité ne peut être qu'un enfant d'une entité mère, et ne peut jamais appartenir à une autre

Épisode 466

parent. Alors que certaines relations parent-enfant permettent à l'enfant de migrer d'un parent à un autre, dans un mappage privé, l'entité détenue a été créée pour appartenir au parent et ne peut jamais être migré. Une fois qu'il est supprimé du parent, il est considéré orphelin et supprimé par le fournisseur.

Seules les relations avec une cardinalité unique du côté source peuvent activer l'orphelin suppression, c'est pourquoi l'option orphanRemoval est définie sur @OneToOne et Annotations de relation @OneToMany, mais sur aucun des @ManyToOne ou @ManyToMany annotations.

Lorsqu'il est spécifié, l'élément orphanRemoval entraîne la suppression de l'entité enfant lorsque la relation entre le parent et l'enfant est rompue. Ceci peut être fait

soit en définissant sur null l'attribut qui contient l'entité associée, soit en plus dans le cas un-à-plusieurs en supprimant l'entité enfant de la collection. Le fournisseur est alors responsable, au moment du vidage ou de la validation (selon la première éventualité), de la suppression de l'orphelin entité enfant.

Dans une relation parent-enfant, l'enfant dépend de l'existence du parent. Si le parent est supprimé, l'enfant devient par définition orphelin et doit également être retiré. Cette deuxième caractéristique du comportement de suppression des orphelins est exactement équivalente à un fonctionnalité que nous avons couverte dans le chapitre 6 appelé *cascade* , dans lequel il est possible de tout sous-ensemble d'un ensemble défini d'opérations dans une relation. Activation de la suppression des orphelins une relation entraîne automatiquement la relation avec l'option d'opération REMOVE ajouté à sa liste en cascade, il n'est donc pas nécessaire de l'ajouter explicitement. Le faire est tout simplement redondant. Il est impossible de désactiver la suppression en cascade d'une relation marquée pour élimination des orphelins puisque sa définition même exige qu'un tel comportement soit présent.

Dans la liste 10-27, la classe Employee définit une relation un-à-plusieurs avec sa liste des évaluations annuelles. Peu importe que la relation soit unidirectionnelle ou bidirectionnel, la configuration et la sémantique sont les mêmes, nous n'avons donc pas besoin de montrer le l'autre côté de la relation.

Annonce 10-27. Classe d'employés avec suppression orpheline d'entités d'évaluation

```
@Entité
Employé de classe publique {
    @Id id int privé;
    @OneToMany (orphanRemoval = vrai)
    Evals <Evaluation> de liste privée;
    // ...
}
```

Épisode 467

Supposons qu'un employé reçoive une évaluation injuste d'un responsable. L'employé peut aller au gestionnaire pour corriger les informations et l'évaluation pourrait être modifié, ou l'employé pourrait devoir faire appel de l'évaluation, et en cas de succès l'évaluation peut simplement être supprimée du dossier de l'employé. Cela le ferait être supprimé de la base de données également. Si l'employé décide de quitter l'entreprise, alors lorsque l'employé est retiré du système, ses évaluations seront automatiquement enlevé avec lui.

Si la collection dans la relation était une carte, saisie par un type d'entité différent, alors la suppression des orphelins ne s'appliquerait qu'aux valeurs d'entité de la carte, pas aux clés. Cette signifie que les clés d'entité ne sont jamais de propriété privée.

Enfin, si l'objet orphelin n'est pas actuellement géré dans le contexte de persistance, soit parce qu'il a été créé en mémoire et pas encore persisté, soit parce qu'il est simplement détaché du contexte de persistance, la suppression des orphelins ne sera pas appliquée. De même, s'il a déjà été supprimé dans le contexte de persistance actuel orphelin la suppression ne sera pas appliquée.

État de la relation de mappage

Il y a des moments où une relation est associée à un état. Par exemple, disons que nous voulons conserver la date à laquelle un employé a été affecté à un projet. Le stockage de l'état sur l'employé est possible mais moins utile, car la date est vraiment liée à la relation de l'employé avec un projet particulier (une seule entrée dans le association plusieurs-à-plusieurs). Retirer un employé d'un projet devrait vraiment la date d'affectation pour partir, donc la stocker dans le cadre de l'employé signifie que nous avons pour s'assurer que les deux sont cohérents l'un avec l'autre, ce qui peut être gênant. En UML, nous montrerions ce type de relation en utilisant une classe d'association. Figure 10-6 montre un exemple de cette technique.

Épisode 468

Figure 10-6. *Modélisation de l'état d'une relation à l'aide d'une classe d'association*

Dans la base de données, tout est rose car on peut simplement ajouter une colonne à la jointure table. Le modèle de données fournit un support naturel pour l'état des relations. Figure 10-7 spectacles la relation plusieurs-à-plusieurs entre EMPLOYÉ et PROJECT avec une jointure élargie table.

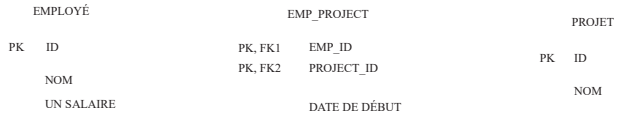

Figure 10-7. *Table de jointure avec état supplémentaire*

Lorsque nous arrivons au modèle objet, cependant, cela devient beaucoup plus problématique. Le problème est que Java n'a pas de prise en charge inhérente de l'état des relations. Les relations sont juste des références d'objet ou des pointeurs, donc aucun état ne peut jamais exister sur eux. L'état existe le objets uniquement et les relations ne sont pas des objets de première classe.

La solution Java consiste à transformer la relation en une entité qui contient le state et mappez la nouvelle entité à ce qui était auparavant la table de jointure. La nouvelle entité va ont une relation plusieurs-à-un avec chacun des types d'entités existants, et chacun des les types d'entités auront une relation un-à-plusieurs avec la nouvelle entité représentant la relation. La clé primaire de la nouvelle entité sera la combinaison des deux relations avec les deux types d'entités. Référencement10-28 montre tous les participants au Relation employé-projet.

Épisode 469

Annonce 10-28. Mappage de l'état de relation avec une entité intermédiaire

```
@Entité
Employé de classe publique {
    @Id id int privé;
    // ...
    @OneToMany (mappedBy = "employé")
    affectations de collection privée <ProjectAssignment>;
    // ...
}

@Entité
Projet de classe publique {
    @Id id int privé;
    // ...
    @OneToMany (mappedBy = "projet")
    affectations de collection privée <ProjectAssignment>;
    // ...
}

@Entité
@Table (nom = "EMP_PROJECT")
@IdClass (ProjectAssignmentId.class)
public class ProjectAssignment {
    @Id
    @ManyToOne
    @JoinColumn (nom = "EMP_ID")
    employé privé;

    @Id
    @ManyToOne
    @JoinColumn (nom = "PROJECT_ID")
    projet de projet privé;

    @Temporal (TemporalType.DATE)
    @Column (name = "START_DATE", pouvant être mis à jour = false)
    private Date startDate;
    // ...
}
```


Épisode 470


```
La classe publique ProjectAssignmentId implémente Serializable {
    employé int privé;
    projet int privé;
    // ...
}
```

Ici, nous avons la clé primaire entièrement composée de relations, avec les deux colonnes de clé étrangère constituant la clé primaire dans la table de jointure EMP_PROJECT. le la date à laquelle l'affectation a été effectuée peut être définie manuellement lorsque l'affectation est créé, ou il peut être associé à un déclencheur qui le déclenche lorsque le l'affectation est créée dans la base de données. Notez que si un déclencheur a été utilisé, l'entité aurait besoin d'être actualisé à partir de la base de données afin de renseigner la date d'affectation champ dans l'objet Java.

Tables multiples

Les scénarios de cartographie les plus courants sont ceux que l'on appelle la variété des rencontres du milieu. Cela signifie que le modèle de données et le modèle objet existent déjà ou, si ce n'est pas le cas existe, puis il est créé indépendamment de l'autre modèle. Ceci est pertinent car il sont un certain nombre de fonctionnalités de l'API Java Persistence qui tentent de résoudre les problèmes qui se posent dans ce cas.

Jusqu'à présent, nous avons supposé qu'une entité était mappée sur une seule table et qu'une seule ligne de cette table représente une entité. Dans un modèle de données existant ou hérité, il était en fait assez courant de diffuser des données, même des données étroitement couplées, entre plusieurs tables. Cela a été fait pour différentes tâches administratives et de performance raisons, dont l'une était de réduire les conflits de table lorsque des sous-ensembles spécifiques de données ont été consultés ou modifiés.

Pour en tenir compte, les entités peuvent être mappées sur plusieurs tables à l'aide de la Annotation @SecondaryTable et sa forme @SecondaryTables au pluriel. La table par défaut ou la table définie par l'annotation @Table est appelée la table primaire, et tout les autres sont appelés tables secondaires. Nous pouvons ensuite distribuer les données dans une entité à travers les lignes de la table principale et des tables secondaires simplement en définissant le tables secondaires comme annotations sur l'entité, puis spécifiant quand nous mappons chacune champ ou propriété de la table dans laquelle se trouve la colonne. Pour ce faire, spécifiez le nom de la table dans l'élément de table dans @Column ou @JoinColumn. Nous n'avons pas eu besoin de l'utiliser élément antérieur, car la valeur par défaut de table est le nom de la table primaire.

Épisode 471

Il ne reste plus qu'à spécifier comment joindre la ou les tables secondaires au table primaire. Nous avons vu au chapitre4 Comment la colonne de jointure de clé primaire est un cas particulier d'une colonne de jointure où la colonne de jointure n'est que la colonne de clé primaire (ou les colonnes de le cas des clés primaires composites). Prise en charge de la jonction de tables secondaires à la table primaire est limitée aux colonnes de jointure de clé primaire et est spécifiée comme un Annotation @PrimaryKeyJoinColumn dans le cadre de l'annotation @SecondaryTable.

Pour illustrer l'utilisation d'une table secondaire, considérez le modèle de données illustré dans Figure 10-8. Il existe une relation de clé primaire entre les tables EMP et EMP_ADDRESS. La table EMP stocke les informations de l'employé principal, tandis que les informations d'adresse a été déplacé vers la table EMP_ADDRESS.

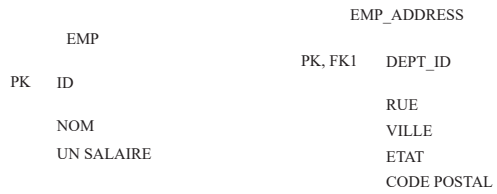

Figure 10-8. *Tables EMP et EMP_ADDRESS*

Pour mapper cette structure de table à l'entité Employee, nous devons déclarer EMP_ADDRESS comme un table secondaire et utilisez l'élément table de l'annotation @Column pour chaque attribut stocké dans cette table. Le listing 10-29 montre l'entité mappée. La clé primaire de l'EMP_ La table ADDRESS se trouve dans la colonne EMP_ID. S'il avait été nommé ID, alors nous n'aurions pas nécessaire pour utiliser l'élément name dans l'annotation @PrimaryKeyJoinColumn. Il est par défaut au nom de la colonne de clé primaire dans la table primaire.

Annonce 10-29. Mappage d'une entité sur deux tables

```
@Entité
@Table (nom = "EMP")
```

```
@SecondaryTable (nom = "EMP_ADDRESS",
    pkJoinColumns = @ PrimaryKeyJoinColumn (nom = "EMP_ID"))
    Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
```


Épisode 472


```
    long salaire privé;
    @Column (table = "EMP_ADDRESS")
    rue privée String;
    @Column (table = "EMP_ADDRESS")
    ville privée de String;
    @Column (table = "EMP_ADDRESS")
    état de chaîne privé;
    @Column (name = "ZIP_CODE", table = "EMP_ADDRESS")
    zip de chaîne privé;
    // ...
}
```

Au chapitre 4, nous avons appris à utiliser les éléments de schéma ou de catalogue dans @Table pour qualifiez la table primaire comme faisant partie d'un schéma ou d'un catalogue de base de données particulier. C'est aussi valide dans l'annotation @SecondaryTable.

Auparavant, lors de la discussion des objets incorporés, nous avons mappé les champs d'adresse du Entité d'employé dans un type incorporé d'adresse. Avec les données d'adresse dans un secondaire table, il est toujours possible de le faire en spécifiant le nom de la table mappée dans le cadre du informations de colonne dans l'annotation @AttributeOverride. Le listing 10-30 démontre cette approche. Notez que nous devons énumérer tous les champs du type incorporé même si les noms de colonne peuvent correspondre aux valeurs par défaut correctes.

Liste 10-30. Mappage d'un type incorporé à une table secondaire

```
@Entité
@Table (nom = "EMP")
@SecondaryTable (nom = "EMP_ADDRESS",
                    pkJoinColumns = @ PrimaryKeyJoinColumn (nom = "EMP_ID"))
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    long salaire privé;
    @Embedded
    @AttributeOverrides ({
        @AttributeOverride (nom = "rue", colonne = @ Colonne (table = "EMP_
        ADRESSE")),
```


Épisode 473


```
        @AttributeOverride (nom = "ville", colonne = @ Colonne (table = "EMP_
        ADRESSE")),
```

```
        @AttributeOverride (nom = "état", colonne = @ Colonne (table = "EMP_
        ADRESSE")),
        @AttributeOverride (nom = "zip",
                                column = @ Column (name = "ZIP_CODE", table = "EMP_
                                ADRESSE"))
    })
    adresse d'adresse privée;
    // ...
}
```

Prenons un exemple plus complexe impliquant plusieurs tables et composés clés primaires. Figure 10-9 montre la structure de table que nous souhaitons mapper. En plus de EMPLOYEE, il existe deux tables secondaires, ORG_STRUCTURE et EMP_LOB. L'ORG_ Le tableau STRUCTURE stocke les informations de reporting des employés et des responsables. La table EMP_LOB stocke des objets volumineux rarement récupérés lors des options de requête normales. En mouvement objets volumineux vers une table secondaire est une technique de conception courante dans de nombreuses bases de données schémas.

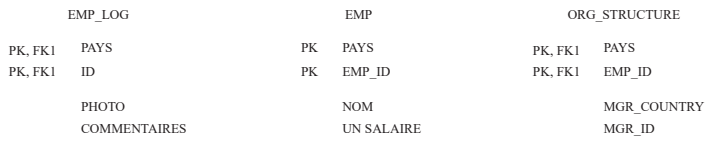

Figure 10-9. *Tables secondaires avec relations de clé primaire composées*

Référencement 10-31 montre l'entité Employé mappée à cette structure de table. le La classe d'ID EmployeeId du Listing 10-10 a été réutilisée dans cet exemple.

Annonce 10-31. Mappage d'une entité avec plusieurs tables secondaires

```
@Entité
@IdClass (EmployeeId.class)
@SecondaryTables ({
    @SecondaryTable (nom = "ORG_STRUCTURE", pkJoinColumns = {
```

Épisode 474


```
        @PrimaryKeyJoinColumn (name = "COUNTRY", referencedColumnName = "COUNTRY"),
        @PrimaryKeyJoinColumn (name = "EMP_ID", referencedColumnName = "EMP_ID")}),
        @SecondaryTable (nom = "EMP_LOB", pkJoinColumns = {
        @PrimaryKeyJoinColumn (name = "COUNTRY", referencedColumnName = "COUNTRY"),
        @PrimaryKeyJoinColumn (name = "ID", referencedColumnName = "EMP_ID")})
})
Employé de classe publique {
    @Id private String country;
    @Id
    @Column (nom = "EMP_ID")
    id int privé;

    @Basic (fetch = FetchType.LAZY)
    @Lob
    @Column (table = "EMP_LOB")
    octet privé [] photo;

    @Basic (fetch = FetchType.LAZY)
    @Lob
```

```
    @Column (table = "EMP_LOB")
    les commentaires de char privé [];

    @ManyToOne
    @JoinColumns ({
        @JoinColumn (name = "MGR_COUNTRY", referencedColumnName = "COUNTRY",
                    table = "ORG_STRUCTURE"),
        @JoinColumn (name = "MGR_ID", referencedColumnName = "EMP_ID",
                    table = "ORG_STRUCTURE")
    })
    gestionnaire d'employé privé;
    // ...
}
```


Épisode 475

Nous avons jeté quelques courbes dans cet exemple pour le rendre plus intéressant. le
Tout d'abord, nous avons défini Employee comme ayant une clé primaire composite. Cela nécessite
informations supplémentaires à fournir pour la table EMP_LOB, car sa clé primaire
n'est pas nommé de la même manière que la table primaire. La prochaine différence est que nous stockons
une relation dans la table secondaire ORG_STRUCTURE. Le MGR_COUNTRY et le MGR_ID
les colonnes se combinent pour référencer l'ID du responsable de cet employé. Depuis le
l'employé a une clé primaire composite, la relation gestionnaire doit également spécifier un ensemble
de colonnes de jointure au lieu d'une seule, et les éléments referencedColumnName dans ceux
les colonnes de jointure font référence aux colonnes de clé primaire COUNTRY et EMP_ID de l'entité
table primaire EMPLOYÉ.

Héritage

L'une des erreurs courantes commises par les développeurs novices orientés objet est qu'ils
se convertir au principe de la réutilisation, mais le pousser trop loin. Il est trop facile de se faire prendre
dans la quête de la réutilisation et de créer des hiérarchies d'héritage complexes, le tout dans l'intérêt de
partager quelques méthodes. Ces types de hiérarchies à plusieurs niveaux conduiront souvent à de la douleur et
difficultés sur la route car l'application devient difficile à déboguer et un défi à
maintenir.

La plupart des applications bénéficient des avantages d'au moins un héritage dans l'objet
modèle. Comme pour la plupart des choses, cependant, la modération doit être appliquée, en particulier lorsqu'elle
vient de mapper les classes sur des bases de données relationnelles. Les grandes hiérarchies peuvent souvent conduire
à une réduction significative des performances, et il se peut que le coût de la réutilisation du code soit plus élevé
que vous voudrez peut-être payer.

Dans les sections suivantes, nous expliquons le support qui existe dans l'API pour mapper
hiérarchies d'héritage et décrivez certaines des répercussions.

Hiérarchies de classe

Parce qu'il s'agit d'un livre sur l'API Java Persistence, le premier et le plus évident
pour commencer à parler d'héritage est dans le modèle d'objet Java. Les entités sont des objets, après
tous, et devrait pouvoir hériter de l'état et du comportement d'autres entités. Ce n'est pas seulement
attendu mais également indispensable pour le développement d'applications orientées objet.

Épisode 476

Qu'est-ce que cela signifie lorsqu'une entité hérite de l'état de sa superclasse d'entités? Ça peut impliquent des choses différentes dans le modèle de données, mais dans le modèle Java, cela signifie simplement que lorsque une entité de sous-classe est instanciée, elle a sa propre version ou copie de ses deux définies localement état et son état hérité, qui sont tous persistants. Bien que ce principe de base ne soit pas du tout surprenant, cela ouvre la question moins évidente de ce qui se passe lorsqu'une entité hérite de quelque chose d'autre qu'une autre entité. Quelles classes une entité est-elle autorisée à étendre, et que se passe-t-il quand il le fait?

Considérez la hiérarchie des classes illustrée dans la figure 10-10 . Comme nous l'avons vu au chapitre 1 , il y a un certain nombre de façons dont l'héritage de classe peut être représenté dans la base de données. Dans l'objet modèle, il peut même y avoir un certain nombre de façons différentes de mettre en œuvre une hiérarchie, certaines qui peuvent inclure des classes non-entité. Nous utilisons cet exemple pour explorer les moyens de persister hiérarchies d'héritage dans les sections suivantes.

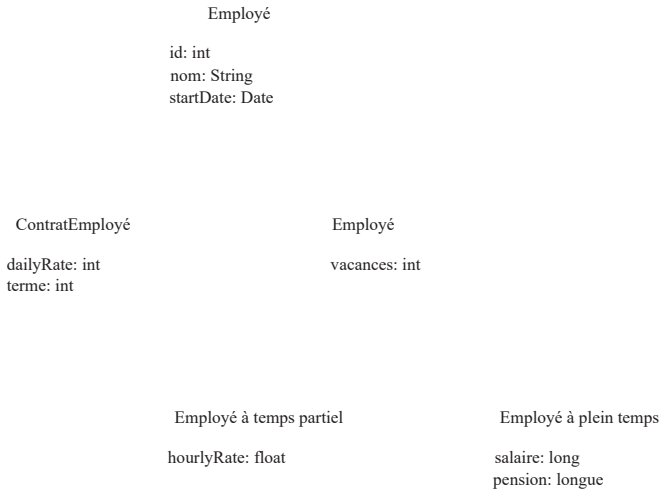

Figure 10-10. *Hiérarchie des classes d'héritage*

Nous différencions une hiérarchie de classes générale, qui est un ensemble de différents types des classes Java qui s'étendent les unes les autres dans un arbre et une hiérarchie d'entités, qui est un arbre composé de classes d'entités persistantes entrecoupées de classes non-entité. Une entité La hiérarchie est enracinée dans la première classe d'entité de la hiérarchie.

Épisode 477

Superclasses mappées

L'API Java Persistence définit un type spécial de classe appelé une superclasse mappée qui est assez utile comme superclasse pour les entités. Une superclasse mappée fournit un classe dans laquelle stocker l'état et le comportement partagés dont les entités peuvent hériter, mais n'est pas en soi une classe persistante et ne peut pas agir en qualité d'entité. Ça ne peut pas être

interrogé et ne peut pas être la cible d'une relation. Les annotations telles que @Table sont non autorisé sur les superclasses mappées car l'état qui y est défini ne s'applique qu'à ses sous-classes d'entités.

Les superclasses mappées peuvent être comparées aux entités de la même manière que une classe abstraite est comparée à une classe concrète; ils peuvent contenir un état et un comportement mais ne peut tout simplement pas être instancié en tant qu'entités persistantes. Une classe abstraite n'est utile que dans relation à ses sous-classes concrètes, et une superclasse mappée n'est utile que comme état et comportement hérité par les sous-classes d'entités qui l'étendent. Ils ne jouent pas de rôle dans une hiérarchie d'héritage d'entité autre que la contribution de cet état et de ce comportement au entités qui en héritent.

Les superclasses mappées peuvent ou non être définies comme abstraites dans leurs définitions de classe, mais il est bon d'en faire de véritables classes Java abstraites. Nous n'en connaissons aucun de bons cas d'utilisation pour en créer des instances Java concrètes sans jamais pouvoir les persister, et il y a de fortes chances que, si vous en trouvez un, vous voulez probablement le mappé la superclasse pour être une entité.

Toutes les règles de mappage par défaut qui s'appliquent aux entités s'appliquent également aux état des relations dans les superclasses mappées. Le plus grand avantage de l'utilisation de mappé superclasses est capable de définir un état partagé partiel auquel il ne faut pas accéder son propre sans l'état supplémentaire que ses sous-classes d'entité lui ajoutent. Si tu n'es pas Assurez-vous de faire d'une classe une entité ou une superclasse mappée, il vous suffit de demander vous-même si vous avez besoin d'interroger ou d'accéder à une instance uniquement exposée en tant qu'instance de cette classe mappée. Cela inclut également les relations, car un la superclasse ne peut pas être utilisée comme cible d'une relation. Si vous répondez oui à une variante de cette question, alors vous devriez probablement en faire une entité de première classe.

Regard sur la figure 10-10 , nous pourrions éventuellement traiter la classe CompanyEmployee en tant que superclasse mappée au lieu d'une entité. Il définit l'état partagé, mais peut-être avons-nous aucune raison d'interroger dessus.

Une classe est indiquée comme étant une superclasse mappée en l'annotant avec le Annotation @MappedSuperclass. Les fragments de classe de Listing10-32 montrent comment le hiérarchie serait mappée avec CompanyEmployee en tant que superclasse mappée.

Épisode 478

Annonce 10-32. Entités héritant d'une superclasse mappée

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    @Temporal (TemporalType.DATE)
    @Column (nom = "S_DATE")
    private Date startDate;
    // ...
}

@Entité
Public class ContractEmployee étend Employee {
    @Column (nom = "D_RATE")
    private int dailyRate;
    terme int privé;
    // ...
}

@MappedSuperclass
classe abstraite publique CompanyEmployee étend Employee {
    vacances int privé;
```

```
    // ...
}

@Entité
public class FullTimeEmployee étend CompanyEmployee {
    long salaire privé;
    pension longue privée;
    // ...
}

@Entité
public class PartTimeEmployee étend CompanyEmployee {
    @Column (nom = "H_RATE")
    flotteur privé hourlyRate;
    // ...
}
```


Épisode 479

Classes transitoires dans la hiérarchie

Les classes d'une hiérarchie d'entités, qui ne sont pas des entités ou des superclasses mappées, sont appelées classes transitoires. Les entités peuvent étendre les classes transitoires directement ou indirectement à travers une superclasse mappée. Lorsqu'une entité hérite d'une classe transitoire, l'état défini dans la classe transitoire est toujours hérité dans l'entité, mais il n'est pas persistant. Dans en d'autres termes, l'entité aura un espace alloué pour l'état hérité, selon le les règles Java habituelles, mais cet état ne sera pas géré par le fournisseur de persistance. Ce sera effectivement ignoré pendant le cycle de vie de l'entité. L'entité peut gérer cet état manuellement via l'utilisation des méthodes de rappel de cycle de vie que nous décrivons au chapitre 12 , ou d'autres approches, mais l'état ne sera pas conservé dans le cadre de la gestion gérée par le fournisseur cycle de vie de l'entité.

On pourrait concevoir d'avoir une hiérarchie composée d'une entité qui a une sous-classe transitoire, qui à son tour a une ou plusieurs sous-classes d'entités. Alors que cette affaire n'est pas vraiment courante, elle est néanmoins possible et peut être réalisée dans de rares circonstances dans lesquelles un état transitoire partagé ou un comportement commun est souhaité. Il serait normalement plus pratique, cependant, de déclarer l'état transitoire ou le comportement dans la superclasse d'entité que de créer une classe transitoire intermédiaire. Référencement10-33 spectacles une entité qui hérite d'une superclasse qui définit l'état transitoire qui est le temps l'entité a été créée en mémoire.

Annonce 10-33. Entité héritant d'une superclasse transitoire

```
classe abstraite publique CachedEntity {
    createTime long privé;

    public CachedEntity () {createTime = System.currentTimeMillis (); }

    public long getCacheAge () {retourne System.currentTimeMillis () -
    créer du temps; }
}

@Entité
Public class Employee étend CachedEntity {
    employé public () {super (); }
    // ...
}
```

Épisode 480

Dans cet exemple, nous avons déplacé l'état transitoire de la classe d'entité vers un état transitoire superclass, mais le résultat final est vraiment le même. L'exemple précédent pourrait avoir été un peu plus soigné sans la classe supplémentaire, mais cet exemple nous permet de partager le transitoire état et comportement sur un nombre quelconque d'entités qui n'ont besoin que d'étendre CachedEntity.

Classes abstraites et concrètes

Nous avons évoqué la notion de classes abstraites versus concrètes dans le contexte de mappé des superclasses, mais nous ne sommes pas entrés plus en détail sur les entités et les transitoires Des classes. La plupart des gens, selon leur philosophie, pourraient s'attendre à ce que tous les non-feuilles les classes dans une hiérarchie d'objets doivent être abstraites, ou à tout le moins que certaines serait. Une restriction selon laquelle les entités doivent toujours être des classes concrètes gâcherait cela assez facilement, et heureusement ce n'est pas le cas. C'est parfaitement acceptable pour les entités, superclasses mappées, ou classes transitoires pour être abstraites ou concrètes à n'importe quel niveau de l'arbre d'héritage. Comme pour les superclasses mappées, rendre les classes transitoires concrètes la hiérarchie ne sert vraiment aucun but, et en règle générale devrait être évitée pour éviter les erreurs de développement accidentelles et les abus.

Le cas dont nous n'avons pas parlé est celui où une entité est un abstrait classe. La seule différence entre une entité qui est une classe abstraite et une qui est une la classe concrète est la règle Java qui interdit l'instanciation des classes abstraites. Ils peuvent toujours définir un état et un comportement persistants qui seront hérités par le béton les sous-classes d'entités en dessous. Ils peuvent être interrogés, dont le résultat sera composé d'instances de sous-classes d'entités concrètes. Ils peuvent également supporter le mappage d'héritage métadonnées pour la hiérarchie.

Notre hiérarchie dans la figure 10-10 avait une classe Employee qui était une classe concrète. Nous ne voudrions pas que les utilisateurs instancient accidentellement cette classe, puis tentent de conserver un employé partiellement défini. Nous pourrions nous protéger contre cela en le définissant comme abstrait. Nous finirions alors avec toutes nos classes non-feuilles étant abstraites et les classes feuilles être persévérant.

Modèles d'héritage

JPA prend en charge trois représentations de données différentes. L'utilisation de deux d'entre eux est assez répandu, alors que le troisième est moins courant et n'a pas besoin d'être pris en charge, bien qu'il soit encore entièrement défini avec l'intention que les fournisseurs puissent être tenus de soutenez-le à l'avenir.

Épisode 481

Lorsqu'une hiérarchie d'entités existe, elle est toujours enracinée dans une classe d'entités. Rappeler que les superclasses mappées ne comptent pas comme des niveaux dans la hiérarchie car elles contribuent seulement aux entités en dessous d'eux. La classe d'entité racine doit signifier l'héritage hiérarchie en étant annoté avec l'annotation @Inheritance. Cette annotation indique la stratégie à utiliser pour la cartographie et doit être l'une des trois stratégies décrites dans les sections suivantes.

Chaque entité de la hiérarchie doit définir ou hériter de son identifiant, ce qui signifie que l'identifiant doit être défini soit dans l'entité racine, soit dans une superclasse mappée Au dessus de. Une superclasse mappée peut être plus haut dans la hiérarchie des classes que là où le

l'identifiant est défini.

Stratégie de table unique

Le moyen le plus courant et le plus performant de stocker l'état de plusieurs classes est de définir une table unique pour contenir un sur-ensemble de tous les états possibles dans l'une des classes d'entités. Cette Cette approche est appelée, sans surprise, une stratégie à table unique. Cela a pour conséquence que, pour toute ligne de table donnée représentant une instance d'une classe concrète, il peut y avoir des colonnes qui n'ont pas de valeurs car elles s'appliquent uniquement à une classe sœur dans la hiérarchie.

À partir de la figure 10-10, nous voyons que l'id est situé dans la classe d'entité Employee racine et est partagé par le reste des classes de persistance. Toutes les entités persistantes dans un héritage tree doit utiliser le même type d'identifiant. Nous n'avons pas besoin d'y penser très longtemps avant nous voyons pourquoi cela a du sens aux deux niveaux. Dans la couche d'objet, ce ne serait pas possible pour lancer une opération polymorphe find () sur une superclasse s'il n'y avait pas de type d'identifiant que nous pourrions transmettre. De même, au niveau de la table, nous aurions besoin de plusieurs colonnes de clé primaire mais sans pouvoir toutes les remplir lors d'une insertion donnée de une instance qui n'a utilisé que l'un d'entre eux.

La table doit contenir suffisamment de colonnes pour stocker tout l'état dans toutes les classes. Un ligne individuelle stocke l'état d'une instance d'entité d'un type d'entité concret, qui impliquerait normalement qu'il y aurait des colonnes non remplies dans chaque ligne. De Bien sûr, cela conduit à la conclusion que les colonnes mappées à l'état de sous-classe concret devrait être nullable, ce qui n'est normalement pas un gros problème mais pourrait être un problème pour certains administrateurs de bases de données.

En général, l'approche à table unique a tendance à gaspiller davantage la base de données tablespace, mais il offre des performances de pointe pour les requêtes polymorphes et l'écriture opérations. Le SQL nécessaire pour exécuter ces opérations est simple, optimisé et ne nécessite pas d'adhésion.

Épisode 482

Pour spécifier la stratégie de table unique pour la hiérarchie d'héritage, la classe d'entité racine est annoté avec l'annotation @Inheritance avec sa stratégie définie sur SINGLE_TABLE.
Dans notre modèle précédent, cela signifierait annoter la classe Employee comme suit:

```
@Entité
@Inheritance (stratégie = InheritanceType.SINGLE_TABLE)
Classe abstraite publique Employé {...}
```

En fin de compte, cependant, la stratégie à table unique est la stratégie par défaut, nous ne le ferions donc pas strictement même besoin d'inclure l'élément de stratégie du tout. Un @Inheritance vide l'annotation ferait tout aussi bien l'affaire.

En chiffres 10-11 , nous voyons la représentation à table unique de notre hiérarchie d'employés modèle. En termes de structure de table et d'architecture de schéma pour la stratégie de table unique, cela ne fait aucune différence que CompanyEmployee soit une superclasse mappée ou une entité.

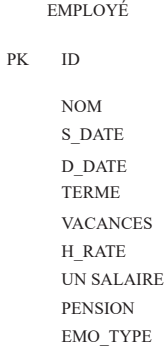

Figure 10-11. *Un modèle de données d'héritage à table unique*

Colonne du discriminateur

Vous avez peut-être remarqué une colonne supplémentaire nommée EMP_TYPE dans la figure 10-11 c'était non mappé à un champ dans l'une des classes de la figure 10-10. Ce champ a un spécial et est obligatoire lors de l'utilisation d'une seule table pour modéliser l'héritage. Cela s'appelle un discriminateur et est mappé à l'aide de l'annotation @DiscriminatorColumn dans en conjonction avec l'annotation @Inheritance que nous avons déjà apprise. Le nom élément de cette annotation spécifie le nom de la colonne qui doit être utilisé comme discriminateur, et s'il n'est pas spécifié, il sera défini par défaut sur une colonne nommée DTYPE.

Épisode 483

Un élément discriminatorType dicte le type de la colonne discriminator. Certains les applications préfèrent utiliser des chaînes pour distinguer les types d'entités, tandis que d'autres comme utiliser des valeurs entières pour indiquer la classe. Le type de la colonne discriminante peut être l'un des trois types de colonne discriminateurs prédéfinis: INTEGER, STRING ou CHAR. Si l'élément discriminatorType n'est pas spécifié, alors le type par défaut de STRING sera assumé.

Valeur discriminante

Chaque ligne du tableau aura une valeur dans la colonne discriminante appelée valeur de discriminateur, ou un indicateur de classe, pour indiquer le type d'entité qui est stocké dans cette rangée. Chaque entité concrète de la hiérarchie d'héritage a donc besoin d'un valeur discriminante spécifique à ce type d'entité afin que le fournisseur puisse traiter ou affectez le type d'entité correct lors du chargement et du stockage de la ligne. La façon dont cela est fait consiste à utiliser une annotation @DiscriminatorValue sur chaque classe d'entité concrète. La ficelle valeur dans l'annotation spécifie la valeur du discriminateur que les instances de la classe seront attribués lorsqu'ils seront insérés dans la base de données. Cela permettra au fournisseur pour reconnaître les instances de la classe lorsqu'elle émet des requêtes. Cette valeur doit être du même type que celui spécifié ou défini par défaut comme élément discriminatorType dans @ Annotation DiscriminatorColumn.

Si aucune annotation @DiscriminatorValue n'est spécifiée, le fournisseur utilisera un manière spécifique au fournisseur d'obtenir la valeur. Si le discriminatorType était STRING, alors le fournisseur utilisera simplement le nom de l'entité comme chaîne d'indicateur de classe. Si la discriminatorType est INTEGER, alors nous devrions soit spécifier le discriminateur valeurs pour chaque classe d'entité ou aucune d'entre elles. Si nous devions en spécifier certains mais pas d'autres, alors nous ne pouvions pas garantir qu'une valeur générée par un fournisseur ne chevaucherait pas une que nous avons spécifié.

Référencement 10-34 montre comment notre hiérarchie d'employés est mappée à une stratégie à table unique.

Annonce 10-34. Hiérarchie d'entités mappée à l'aide d'une stratégie de table unique

```
@Entité
@Table (nom = "EMP")
@Héritage
@DiscriminatorColumn (nom = "EMP_TYPE")
Classe abstraite publique Employé {...}
```

Épisode 484


```
@Entité
Public class ContractEmployee étend Employee {...}

@MappedSuperclass
classe abstraite publique CompanyEmployee étend Employee {...}

@Entité
@DiscriminatorValue ("FTEmp")
public class FullTimeEmployee étend CompanyEmployee {...}

@Entity (nom = "PTEmp")
classe publique PartTimeEmployee étend CompanyEmployee {...}
```

La classe Employee est la classe racine, elle établit donc la stratégie d'héritage et colonne discriminateur. Nous avons supposé la stratégie par défaut de SINGLE_TABLE et type de discriminateur STRING.

Ni les classes Employé ni EntrepriseEmployé n'ont de valeurs discriminantes, car les valeurs de discriminateur ne doivent pas être spécifiées pour les classes d'entités abstraites, mappées superclasses, classes transitoires ou toute classe abstraite d'ailleurs. Seulement du béton les classes d'entités utilisent des valeurs discriminantes car ce sont les seules qui obtiennent réellement stockés et extraits de la base de données.

L'entité ContractEmployee n'utilise pas d'annotation @DiscriminatorValue, car la chaîne par défaut "ContractEmployee", qui est le nom d'entité par défaut est donné à la classe, c'est exactement ce que nous voulons. La classe FullTimeEmployee répertorie explicitement sa valeur discriminante est "FTEmp", c'est donc ce qui est stocké dans chaque ligne pour les instances de FullTimeEmployee. Pendant ce temps, la classe PartTimeEmployee obtiendra "PTEmp" comme son discriminateur car il a défini son nom d'entité sur "PTEmp", et le nom d'entité obtient utilisé comme valeur discriminante quand aucune n'est spécifiée.

En chiffres 10-12 , nous pouvons voir un échantillon de certaines des données que nous pourrions trouver le modèle et les paramètres antérieurs. Nous pouvons voir dans la colonne discriminateur EMP_TYPE que il existe trois types différents d'entités concrètes. Nous voyons également des valeurs nulles dans les colonnes qui ne s'appliquent pas à une instance d'entité.

Épisode 485

EMPLOYÉ

ID	NOM	NOM	VACANCES	À TERME	D_RATE	SALAIRE	H_RATE	PENSION	EMP_TYPE
1	John	020101	500	12					ContratEmployé
2	Paul	020408	600	24					ContratEmployé
3	Sarah	030610	700	18					ContratEmployé
4	Patrick	040701			15		55 000	100 000	FTEmp
5	Joan	030909			15		59 000	200 000	FTEmp
6	Sam	000312			20		60000	450000	FTEmp
7	marque	041101			15	17.00			PTEmp
8	Ryan	051205			15	16.00			PTEmp
9	Jackie	080103			15	15.00			PTEmp

Figure 10-12. *Exemple de données d'héritage de table unique*

Stratégie rejoint

Du point de vue d'un développeur Java, un modèle de données qui mappe chaque entité à la sienne
table a beaucoup de sens. Chaque entité, qu'elle soit abstraite ou concrète, aura
son état mappé sur une table différente. Conformément à notre description précédente, mappé
les superclasses ne sont pas mappées à leurs propres tables mais sont mappées dans le cadre de leur entité
sous-classes.

Le mappage d'une table par entité fournit la réutilisation des données qu'un schéma de données normalisé [2]
offre et constitue le moyen le plus efficace de stocker des données partagées par plusieurs sous-classes dans
une hiérarchie. Le problème est que, quand vient le temps de réassembler une instance de l'un des
les sous-classes, les tables des sous-classes doivent être jointes avec la superclasse
les tables. Cela rend assez évident pourquoi cette stratégie est appelée stratégie conjointe. C'est aussi
un peu plus cher d'insérer une instance d'entité, car une ligne doit être insérée dans
chacune de ses tables de superclasse en cours de route.

Rappelez-vous de la stratégie de table unique que l'identifiant doit être du même type pour
chaque classe de la hiérarchie. Dans une approche conjointe, nous aurons le même type de clé primaire
dans chacune des tables, et la clé primaire d'une table de sous-classe agit également comme une clé étrangère qui
joint à sa table de superclasse. Cela devrait sonner une cloche en raison de sa similitude avec le multiple-
cas de table plus tôt dans le chapitre où nous avons joint les tables ensemble à l'aide des clés primaires
des tables et utilisé l'annotation @PrimaryKeyJoinColumn pour l'indiquer. Nous utilisons ceci
même annotation dans le cas d'héritage joint puisque nous avons plusieurs tables qui chacune

[2] La normalisation des données est une pratique de base de données qui tente de supprimer les données stockées de manière redondante.
Pour l'article fondateur sur la normalisation des données, voir «Un modèle relationnel de données pour
Databanks »par E. F. Codd (Communications de l'ACM, 13 (6) juin 1970). Aussi, toute base de données
le livre ou le papier de conception devrait avoir un aperçu.

Épisode 486

contiennent le même type de clé primaire et chacun a potentiellement une ligne qui contribue à la
état final de l'entité combinée.

Si l'héritage joint est à la fois intuitif et efficace en termes de stockage de données, le
se joindre à ce que cela nécessite rend son utilisation quelque peu coûteuse lorsque les hiérarchies sont profondes
ou large. Plus la hiérarchie est profonde, plus il faudra de jointures pour assembler des instances de
l'entité concrète en bas. Plus la hiérarchie est large, plus il faudra de jointures pour
requête sur une superclasse d'entités.

En chiffres 10-13 , nous voyons notre exemple Employee mappé à une architecture de table jointe.
Les données d'une sous-classe d'entités sont réparties sur les tables de la même manière qu'elles
répartis dans la hiérarchie des classes. Lors de l'utilisation d'une architecture jointe, la décision de
si CompanyEmployee est une superclasse mappée ou une entité fait une différence, car
les superclasses mappées ne sont pas mappées aux tables. Une entité, même si c'est un abstrait
classe, fait toujours. Figure10-13 le montre comme une superclasse mappée, mais s'il s'agissait d'une entité,
alors une table COMPANY_EMP supplémentaire existerait avec les colonnes ID et VACATION,
et la colonne VACATION dans les tables FT_EMP et PT_EMP ne serait pas présente.

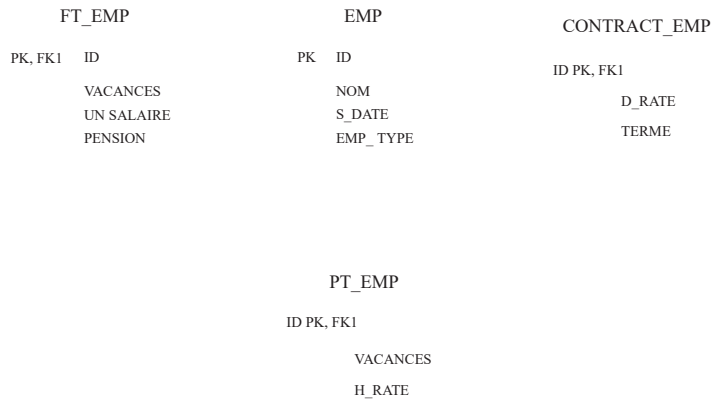

Figure 10-13. *Modèle de données d'héritage joint*

Pour mapper une hiérarchie d'entités à un modèle joint, l'annotation @Inheritance doit spécifiez uniquement JOINED comme stratégie. Comme l'exemple de table unique, les sous-classes adoptez la même stratégie que celle spécifiée dans la superclasse d'entité racine.

Épisode 487

Même s'il existe plusieurs tables pour modéliser la hiérarchie, le discriminateur La colonne n'est définie que sur la table racine, donc l'annotation @DiscriminatorColumn est placé sur la même classe que l'annotation @Inheritance.

Astuce Certains fournisseurs proposent des implémentations de l'héritage joint sans utiliser d'une colonne discriminante. les colonnes discriminantes doivent être utilisées si le fournisseur la portabilité est requise.

Notre exemple de hiérarchie d'employés peut être mappé à l'aide de l'approche jointe illustrée dans l'annonce 10-35. Dans cet exemple, nous avons utilisé des colonnes discriminantes d'entiers au lieu de type de chaîne par défaut.

Annonce 10-35. Hiérarchie d'entités mappée à l'aide de la stratégie jointe

```
@Entité
@Table (nom = "EMP")
@Inheritance (stratégie = InheritanceType.JOINED)
@DiscriminatorColumn (nom = "EMP_TYPE", discriminatorType = DiscriminatorType.
ENTIER)
Classe abstraite publique Employé {...}

@Entité
@Table (nom = "CONTRACT_EMP")
@DiscriminatorValue ("1")
Public class ContractEmployee étend Employee {...}

@MappedSuperclass
classe abstraite publique CompanyEmployee étend Employee {...}

@Entité
@Table (nom = "FT_EMP")
@DiscriminatorValue ("2")
public class FullTimeEmployee étend CompanyEmployee {...}

@Entité
@Table (nom = "PT_EMP")
@DiscriminatorValue ("3")
classe publique PartTimeEmployee étend CompanyEmployee {...}
```


Épisode 488

Stratégie table par classe de béton

Une troisième approche pour mapper une hiérarchie d'entités consiste à utiliser une stratégie où une table par la classe concrète est définie. Cette architecture de données va dans le sens inverse du non-normalisation des données d'entité et mappe chaque classe d'entité concrète et tous ses hérités état dans une table séparée. Cela a pour effet de redéfinir tous les états partagés dans les tableaux de toutes les entités concrètes qui en héritent. Il n'est pas nécessaire que cette stratégie soit pris en charge par les fournisseurs, mais est inclus car il est prévu qu'il sera nécessaire dans une future version de l'API. Nous le décrivons brièvement par souci d'exhaustivité.

Le côté négatif de l'utilisation de cette stratégie est qu'elle crée des requêtes polymorphes à travers une hiérarchie de classes plus chère que les autres stratégies. Le problème est qu'il doit soit émettez plusieurs requêtes distinctes dans chacune des tables de sous-classes, soit interrogez sur tous utilisant une opération UNION, qui est généralement considérée comme coûteuse lorsque beaucoup de données sont impliquées. S'il y a des classes concrètes non-feuille, alors chacune d'elles aura sa propre table. Les sous-classes des classes concrètes devront stocker les champs dans leurs propres tables, ainsi que leurs propres champs définis.

Le bon côté des hiérarchies table par classe concrète par rapport à joint les hiérarchies sont observées dans les cas d'interrogation sur des instances d'une seule entité concrète. Dans le cas joint, chaque requête nécessite une jointure, même lors d'une requête sur un seul classe d'entité concrète. Dans le cas de la table par classe concrète, cela s'apparente à la table simple hiérarchie car la requête est limitée à une seule table. Un autre avantage est que le la colonne du discriminateur disparaît. Chaque entité concrète a sa propre table séparée, et il n'y a pas de mélange ou de partage de schéma, donc aucun indicateur de classe n'est jamais nécessaire.

Mapper notre exemple à ce type de hiérarchie est une question de spécification de la stratégie comme TABLE_PER_CLASS et en s'assurant qu'il existe une table pour chacune des classes concrètes. Si une base de données héritée est utilisée, les colonnes héritées pourraient être nommées différemment dans chacune des tables concrètes et l'annotation @AttributeOverride entrerait pratique. Dans ce cas, la table CONTRACT_EMP ne contenait pas les colonnes NAME et S_DATE mais à la place, FULLNAME et SDATE ont été définis pour les champs name et startDate dans Employee.

Si l'attribut que nous voulions remplacer était une association au lieu d'un simple mappage d'état, alors nous pourrions toujours remplacer le mappage, mais nous aurions besoin de utilisez une annotation @AssociationOverride au lieu de @AttributeOverride. le L'annotation @AssociationOverride nous permet de remplacer les colonnes de jointure utilisées pour référencer l'entité cible d'une association plusieurs-à-un ou un-à-un définie dans une superclasse mappée. Pour montrer cela, nous devons ajouter un attribut de gestionnaire à la CompanyEmployee a mappé la superclasse. La colonne de jointure est mappée par défaut dans le

Épisode 489

CompanyEmployee dans la colonne MANAGER dans les deux sous-classes FT_EMP et PT_EMP tables, mais dans PT_EMP, le nom de la colonne de jointure est en fait MGR. Nous remplaçons la jointure colonne en ajoutant l'annotation @AssociationOverride à PartTimeEmployee classe d'entité et spécifiant le nom de l'attribut que nous surchargons et la jointure colonne que nous remplaçons pour être. Le Listing 10-36 montre un exemple complet du mappages d'entités, y compris les remplacements.

Annonce 10-36. Hiérarchie d'entités mappée à l'aide d'une table par classe concrète Stratégie

```
@Entité
@Inheritance (strategy = InheritanceType.TABLE_PER_CLASS)
Classe abstraite publique Employé {
    @Id id int privé;
    nom de chaîne privé;
    @Temporal (TemporalType.DATE)
    @Column (nom = "S_DATE")
```

```
        private Date startDate;
        // ...
    }

    @Entité
    @Table (nom = "CONTRACT_EMP")
    @AttributeOverrides ({
        @AttributeOverride (nom = "nom", colonne = @ Colonne (nom = "FULLNAME")),
        @AttributeOverride (name = "startDate", column = @ Column (name = "SDATE"))
    })
    Public class ContractEmployee étend Employee {
        @Column (nom = "D_RATE")
        private int dailyRate;
        terme int privé;
        // ...
    }

    @MappedSuperclass
    classe abstraite publique CompanyEmployee étend Employee {
        vacances int privé;
        @ManyToOne
```


Épisode 490


```
        gestionnaire d'employé privé;
        // ...
}

@Entity @Table (nom = "FT_EMP")
public class FullTimeEmployee étend CompanyEmployee {
        long salaire privé;
        @Column (nom = "PENSION")
        pension privée à long terme
        // ...
}

@Entité
@Table (nom = "PT_EMP")
@AssociationOverride (nom = "manager",
                              joinColumns = @ JoinColumn (nom = "MGR"))
public class PartTimeEmployee étend CompanyEmployee {
        @Column (nom = "H_RATE")
        flotteur privé hourlyRate;
        // ...
}
```

L'organisation des tables montre comment ces colonnes sont mappées aux tables concrètes. Voir la figure 10-14 pour une image claire de ce à quoi ressembleraient les tableaux et de différents types d'instances d'employés seraient stockés.

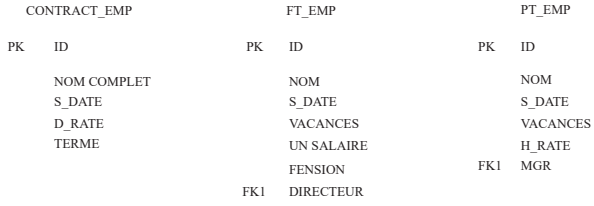

Figure 10-14. *Modèle de données table par classe concrète*

Épisode 491

Héritage mixte

Nous devrions commencer cette section en disant que la pratique de mélanger les types d'héritage au sein d'une même hiérarchie d'héritage est actuellement en dehors de la spécification. Nous sommes en l'incluant car il est à la fois utile et intéressant, mais nous vous avertissons qu'il peut ne pas être portable pour se fier à un tel comportement, même si votre fournisseur le prend en charge.

De plus, il est vraiment logique de ne mélanger que l'héritage à table unique et l'héritage joint les types. Nous montrons un exemple de mélange de ces deux, en gardant à l'esprit que leur soutien est spécifique au fournisseur. L'intention est que, dans les futures versions de la spécification, le plus utile les cas seront normalisés et devront être pris en charge par des implémentations conformes.

Le principe du mélange des types d'héritage est qu'il est bien dans le domaine de possibilités qu'un modèle de données comprenne une combinaison de table unique et de table jointe conceptions au sein d'une même hiérarchie d'entités. Cela peut être illustré en prenant notre joint exemple dans la figure 10-13 et stockage de FullTimeEmployee et PartTimeEmployee instances dans une seule table. Cela produirait un modèle comme celui montré dans la figure 10-15 .

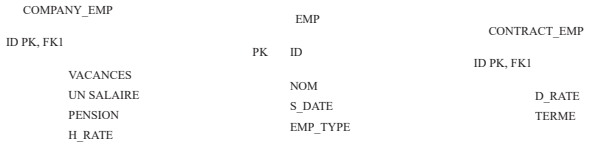

Figure 10-15. *Modèle de données d'héritage mixte*

Dans cet exemple, la stratégie jointe est utilisée pour l'employé et l'employé contractuel classes, tandis que les classes CompanyEmployee, FullTimeEmployee et PartTimeEmployee revenir à un modèle à table unique. Faire basculer cette stratégie d'héritage au niveau de la EntrepriseEmployé, nous devons apporter un simple changement à la hiérarchie. Nous devons tourner CompanyEmployee dans une entité abstraite au lieu d'une superclasse mappée afin de pouvoir porter les nouvelles métadonnées d'héritage. Notez qu'il s'agit simplement d'un changement d'annotation, pas apporter des modifications au modèle de domaine.

Les stratégies d'héritage peuvent être mappées comme indiqué dans la liste 10-37 . Notez que nous n'ont pas besoin d'avoir une colonne discriminante pour la sous-hiérarchie à table unique car nous en ont déjà un dans la table EMP de la superclasse.

Épisode 492

Annonce 10-37. Hiérarchie d'entités mappée à l'aide de stratégies mixtes

```
@Entité
@Table (nom = "EMP")
```

```
@Inheritance (stratégie = InheritanceType.JOINED)
@DiscriminatorColumn (nom = "EMP_TYPE")
Classe abstraite publique Employé {
        @Id id int privé;
        nom de chaîne privé;
        @Temporal (TemporalType.DATE)
        @Column (nom = "S_DATE")
        private Date startDate;
        // ...
}

@Entité
@Table (nom = "CONTRACT_EMP")
Public class ContractEmployee étend Employee {
        @Column (name = "D_RATE") private int dailyRate;
        terme int privé;
        // ...
}

@Entité
@Table (nom = "COMPANY_EMP")
@Inheritance (stratégie = InheritanceType.SINGLE_TABLE)
classe abstraite publique CompanyEmployee étend Employee {
        vacances int privé;
        // ...
}

@Entité
public class FullTimeEmployee étend CompanyEmployee {
        long salaire privé;
        @Column (nom = "PENSION")
        pension privée à long terme
        // ...
}
```

Épisode 493


```
@Entité
public class PartTimeEmployee étend CompanyEmployee {
        @Column (nom = "H_RATE")
        flotteur privé hourlyRate;
        // ...
}
```

Prise en charge de l'API de date et d'heure Java 8

Dans Java EE 8, l'API Date et heure a été ajoutée en tant que nouvelle fonctionnalité.

Pour cette raison, de nombreux programmeurs ont demandé un soutien officiel dans le JPA 2.2 pour être utilisé comme un simple AttributeConverter.

Avec JPA 2.2, nous n'avons plus besoin du convertisseur; il a ajouté un support pour la cartographie de les types java.time suivants:

java.time.LocalDate
java.time.LocalTime
java.time.LocalDateTime
java.time.OffsetTime
java.time.OffsetDateTime

À partir de l'exemple précédent, l'exemple d'API Date et heure est affiché dans

Référencement 10-38 .

Annonce 10-38. API de date et d'heure

```
@Entité
@Table (nom = "EMP")
@Inheritance (stratégie = InheritanceType.JOINED)
@DiscriminatorColumn (nom = "EMP_TYPE")
Classe abstraite publique Employé {
    @Id id int privé;
    nom de chaîne privé;

    @Colonne
    date LocalDate privée;
```


Épisode 494


```
    @Colonne
    private LocalDateTime dateTime;
    // ...
}
```

Remarque Vous pouvez trouver plus d'informations sur l'API de date et d'heure java ee 8 sur le lien suivant: http://www.oracle.com/technetwork/articles/java/jf14-date-heure-2125367.html

Résumé

Les exigences de mappage d'entités vont souvent bien au-delà des mappages simplistes qui mappent un champ ou une relation avec une colonne nommée. Dans ce chapitre, nous avons abordé certains des plus pratiques de cartographie variées et diverses prises en charge par l'API Java Persistence.

Nous avons discuté de la façon de délimiter les identifiants de base de données au cas par cas, ou pour tous les mappages dans une unité de persistance. Nous avons illustré comment la délimitation des identifiants permet inclusion de caractères spéciaux et respecte la casse lorsque la base de données cible l'exige.

Une méthode de conversion fine de l'état d'attribut de base s'est avérée une technique puissante pour adapter les données. En créant des convertisseurs, nous avons également pu persister types de données existants et nouvellement définis de manière hautement personnalisable. La conversion peut être contrôlées de manière déclarative sur une base flexible par attribut ou à tous les niveaux.

Nous avons montré comment les objets intégrables peuvent avoir un état, des collections d'éléments, Embeddables imbriqués, et même des relations. Nous avons donné des exemples de réutilisation d'un objet intégrable avec des relations en remplaçant les mappages de relations au sein de l'entité d'intégration.

Les identificateurs peuvent être composés de plusieurs colonnes. Nous avons révélé les deux approches pour définir et utiliser des clés primaires composées, et démontrer quand elles pourraient être utilisé. Nous avons établi comment d'autres entités peuvent avoir des références de clés étrangères à des entités avec identificateurs composés et expliqué comment plusieurs colonnes de jointure peuvent être utilisées dans contexte lorsqu'une seule colonne de jointure s'applique. Nous avons également montré quelques exemples de cartographie identifiants, appelés identifiants dérivés, qui incluaient une relation dans le cadre de leur identités.

Épisode 495

Nous avons expliqué certaines fonctionnalités de relation avancées, telles que les mappages en lecture seule et optionnelles, et a montré comment ils pouvaient être utiles à certains modèles. Nous sommes ensuite allés sur pour décrire certains des scénarios de mappage les plus avancés qui incluaient l'utilisation de la jointure tables ou en évitant parfois l'utilisation de tables de jointure. Le sujet de l'élimination des orphelins était également abordé et clarifié.

Nous avons ensuite montré comment répartir l'état de l'entité sur plusieurs tables et comment utiliser les tables secondaires avec des relations. Nous avons même vu comment un objet incorporé peut mapper à une table secondaire d'une entité.

Enfin, nous sommes allés dans le détail sur les trois différentes stratégies d'héritage qui peuvent être utilisé pour mapper les hiérarchies d'héritage aux tables. Nous avons expliqué les superclasses mappées et comment ils peuvent être utilisés pour définir un état et un comportement partagés. Nous avons passé en revue les données des modèles qui différencient les différentes approches et montrent comment cartographier une entité hiérarchie aux tables dans chaque cas. Nous avons terminé en illustrant comment mélanger l'héritage types au sein d'une même hiérarchie.

Dans le chapitre suivant, nous poursuivons notre discussion sur des sujets avancés mais tournons notre attention aux requêtes et à l'utilisation du SQL natif et des procédures stockées. Nous expliquons également comment pour créer des graphiques d'entité et les utiliser pour créer des plans d'extraction de requête.

Épisode 496

CHAPITRE 11

Requêtes avancées

`

À ce stade, la plupart d'entre vous en sauront suffisamment sur les requêtes à partir de ce que vous avez appris jusqu'à présent, vous n'aurez probablement besoin d'aucun des éléments de ce chapitre. En fait, à peu près la moitié de ce chapitre traite de l'utilisation de requêtes qui coupleront votre application à une cible base de données, ils doivent donc être utilisés avec un certain degré de planification et de prudence dans tous les cas.

La version 2.2 de JPA n'a introduit aucune nouvelle fonctionnalité liée aux requêtes avancées.

Nous commençons par passer en revue le support JPA pour les requêtes SQL natives et montrons comment le les résultats peuvent être mappés à des entités, des non-entités ou de simples résultats de projection de données. nous puis passez aux requêtes de procédure stockée et expliquez comment une procédure stockée peut être invoqué depuis une application JPA et comment les résultats peuvent être renvoyés.

La seconde moitié du chapitre traite des graphes d'entités et de la manière dont ils peuvent être utilisés pour remplacer le type de récupération d'un mappage pendant une requête. Lorsqu'il est passé en tant que graphiques de récupération ou graphiques de charge, ils offrent une grande flexibilité lors de l'exécution pour contrôler quel état doit obtenir chargé et quand.

Requêtes SQL

Avec tous les efforts consacrés à l'abstraction du modèle de données physique, à la fois termes de mappage objet-relationnel et JP QL, il peut être surprenant d'apprendre que SQL est bien vivant dans JPA. Bien que JP QL soit la méthode préférée d'interrogation sur les entités, SQL ne peut pas être ignoré comme un élément nécessaire dans de nombreuses applications d'entreprise. le la taille et l'étendue des fonctionnalités SQL, prises en charge par les principaux fournisseurs de bases de données, signifie qu'un langage portable tel que JP QL ne pourra jamais englober complètement tous leurs caractéristiques.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_11

Épisode 497

Remarque Les requêtes SQL sont également appelées requêtes natives. Méthodes EntityManager et les annotations de requête liées aux requêtes SQL utilisent également cette terminologie. Alors que ce permet à d'autres langages de requête d'être pris en charge à l'avenir, toute chaîne de requête dans un l'opération de requête native est supposée être SQL.

Avant de discuter de la mécanique des requêtes SQL, examinons d'abord certaines des raisons pourquoi un développeur utilisant JP QL pourrait vouloir intégrer des requêtes SQL dans son application.

Premièrement, JP QL, malgré les améliorations apportées à JPA 2.0, ne contient toujours qu'un sous-ensemble de les fonctionnalités prises en charge par de nombreux fournisseurs de bases de données. Vues en ligne (sous-requêtes dans le FROM clause), des requêtes hiérarchiques et des expressions de fonction supplémentaires pour manipuler la date et les valeurs de temps ne sont que quelques-unes des fonctionnalités non prises en charge dans JP QL.

Deuxièmement, bien que les fournisseurs puissent fournir des conseils pour vous aider à optimiser une expression JP QL, il existe des cas où le seul moyen d'atteindre les performances requises par une application consiste à remplacer la requête JP QL par une version SQL optimisée manuellement. Cela peut être un simple restructuration de la requête générée par le fournisseur de persistance, ou il peut s'agir d'un fournisseur version spécifique qui exploite les conseils de requête et les fonctionnalités spécifiques à une base de données particulière.

Bien sûr, ce n'est pas parce que vous pouvez utiliser SQL que vous devriez le faire. Persistance les fournisseurs sont devenus très compétents pour générer des requêtes haute performance, et la plupart des limitations de JP QL peuvent souvent être contournées dans le code d'application. Nous vous recommandons d'éviter SQL dans un premier temps si possible, puis de ne l'introduire que lorsque nécessaire. Cela permettra à vos requêtes d'être plus portables entre les bases de données et plus maintenable à mesure que vos mappages changent.

Les sections suivantes expliquent comment les requêtes SQL sont définies à l'aide de JPA et comment

leurs ensembles de résultats peuvent être mappés vers des entités. L'un des principaux avantages de la requête SQL support est qu'il utilise la même interface de requête que celle utilisée pour les requêtes JP QL. Avec quelques petits exceptions décrites plus loin, toutes les opérations d'interface de requête décrites dans les chapitres précédents s'appliquent également aux requêtes JP QL et SQL.

Requêtes natives et JDBC

Une question parfaitement valable pour quiconque étudie la prise en charge SQL dans JPA est de savoir si nécessaire du tout. JDBC est utilisé depuis des années, fournit un large éventail de fonctionnalités et fonctionne bien. C'est une chose d'introduire une API de persistance qui fonctionne sur les entités, mais une autre chose entièrement pour introduire une nouvelle API pour émettre des requêtes SQL.

Épisode 498

La principale raison d'envisager d'utiliser des requêtes SQL dans JPA, au lieu de simplement émettre JDBC requêtes, correspond au moment où le résultat de la requête sera reconverti en entités. En tant que Par exemple, considérons un cas d'utilisation typique de SQL dans une application qui utilise JPA. Donné l'identifiant d'employé d'un responsable, l'application doit déterminer tous les employés qui relèvent de ce gestionnaire directement ou indirectement. Par exemple, si la requête était pour un cadre supérieur, les résultats incluraient tous les gestionnaires qui relèvent cadre supérieur ainsi que les employés qui relèvent de ces gestionnaires. Ce type de la requête ne peut pas être implémentée en utilisant JP QL, mais une base de données telle qu'Oracle nativement prend en charge les requêtes hiérarchiques à cette fin. Le listing 11-1 montre le typique séquence d'appels JDBC pour exécuter cette requête et transformer les résultats en entités pour utilisation par l'application.

Annonce 11-1. Interrogation d'entités à l'aide de SQL et JDBC

```
@Apatride
La classe publique OrgStructureBean implémente OrgStructure {
    Chaîne finale statique privée ORG_QUERY =
        "SELECT emp_id, nom, salaire" +
        "FROM emp" +
        "START WITH manager_id =?" +
        "CONNECTER PAR PRIOR emp_id = manager_id";

    @Ressource
    DataSource hrDs;

    Liste publique findEmployeesReportingTo (int managerId) {
        Connexion conn = null;
        PreparedStatement sth = null;
        essayez {
            conn = hrDs.getConnection ();
            sth = conn.prepareStatement (ORG_QUERY);
            sth.setLong (1, managerId);
            ResultSet rs = sth.executeQuery ();

            ArrayList <Employee> result = new ArrayList <Employee> ();
            while (rs.next ()) {
                Employé emp = nouvel employé ();
                emp.setId (rs.getInt (1));
```

Épisode 499


```
                emp.setName (rs.getString (2));
                emp.setSalary (rs.getLong (3));
                result.add (emp);
            }
            résultat de retour;
        } catch (SQLException e) {
            lancer une nouvelle EJBException (e);
        }
    }
}
```

Considérez maintenant la syntaxe alternative prise en charge par JPA, comme indiqué dans la liste 11-2 . Par indiquant simplement que le résultat de la requête est l'entité Employee, le moteur de requête utilise le mappage objet-relationnel de l'entité pour déterminer quelle colonne de résultat mappe aux propriétés de l'entité et construit le jeu de résultats en conséquence.

Liste 11-2. Interrogation d'entités à l'aide de SQL et de l'interface de requête

```
@Apatride
La classe publique OrgStructureBean implémente OrgStructure {
    Chaîne finale statique privée ORG_QUERY =
        "SELECT emp_id, nom, salaire, manager_id, dept_id, address_id" +
        "FROM emp" +
        "START WITH manager_id =?" +
        "CONNECTER PAR PRIOR emp_id = manager_id";

    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    Liste publique findEmployeesReportingTo (int managerId) {
        renvoie em.createNativeQuery (ORG_QUERY, Employee.class)
                .setParameter (1, managerId)
                .getResultList ();
    }
}
```


Épisode 500

Non seulement le code est beaucoup plus facile à lire, mais il utilise également la même interface de requête qui peut être utilisé pour les requêtes JP QL. Cela permet de garder le code d'application cohérent car il doit se préoccuper uniquement des interfaces EntityManager et Query.

Un résultat malheureux de l'ajout de l'interface TypedQuery dans JPA 2.0 est que le La méthode createNativeQuery () était déjà définie dans JPA 1.0 pour accepter une chaîne SQL et une classe de résultat et renvoyer une interface de requête non typée. Maintenant, il n'y a pas de recul manière compatible de renvoyer une TypedQuery au lieu d'une Query. La conséquence regrettable est que lorsque la méthode createNativeQuery () est appelée avec un argument de classe de résultat on pourrait penser à tort qu'il produira un TypedQuery, comme createQuery () et createNamedQuery () fait quand une classe de résultat est transmise.

Définition et exécution de requêtes SQL

Les requêtes SQL peuvent être définies dynamiquement lors de l'exécution ou nommées dans l'unité de persistance métadonnées, similaires aux définitions de requête JP QL présentées au chapitre 7. La clé la différence entre la définition des requêtes JP QL et SQL réside dans la compréhension que la Le moteur de requête ne doit pas analyser et interpréter le SQL spécifique au fournisseur. Afin d'exécuter un Requête SQL et obtenez des instances d'entité en retour, des informations de mappage supplémentaires sur le le résultat de la requête est requis.

La première et la plus simple forme de définition dynamique d'une requête SQL qui renvoie un le résultat de l'entité consiste à utiliser la méthode createNativeQuery () de l'interface EntityManager, en passant la chaîne de requête et le type d'entité qui sera retourné. Référencement11-2 dans le la section précédente a démontré cette approche pour mapper les résultats d'une hiérarchie Oracle requête à l'entité Employee. Le moteur de requête utilise le mappage objet-relationnel de l'entité pour déterminer quels alias de colonne de résultat correspondent à quelles propriétés d'entité. Comme chaque ligne est traitée, le moteur de requête instancie une nouvelle instance d'entité et définit le données disponibles.

Si les alias de colonne de la requête ne correspondent pas exactement à l'objet-relationnel mappages pour l'entité ou si les résultats contiennent à la fois des résultats d'entité et de non-entité, Les métadonnées de mappage de l'ensemble de résultats SQL sont requises. Les mappages de jeux de résultats SQL sont définis comme les métadonnées de l'unité de persistance et sont référencées par leur nom. Lorsque le createNativeQuery () est appelée avec une chaîne de requête SQL et un nom de mappage d'ensemble de résultats, la requête Le moteur utilise ce mappage pour créer le jeu de résultats. Les mappages de jeux de résultats SQL sont discutés dans la section suivante.

Épisode 501

Les requêtes SQL nommées sont définies à l'aide de l'annotation @NamedNativeQuery. Cette l'annotation peut être placée sur n'importe quelle entité et définit le nom de la requête ainsi que le texte de la requête. Comme les requêtes nommées JP QL, le nom de la requête doit être unique dans le unité de persistance. Si le type de résultat est une entité, l'élément resultClass peut être utilisé pour indiquer la classe d'entité. Si le résultat nécessite un mappage SQL, le resultSetMapping L'élément peut être utilisé pour spécifier le nom du mappage. Le Listing 11-3 montre comment le la requête hiérarchique présentée précédemment serait définie comme une requête nommée.

Liste 11-3. Utilisation d'une annotation pour définir une requête native nommée

```
@NamedNativeQuery (
    name = "orgStructureReportingTo",
    query = "SELECT emp_id, nom, salaire, manager_id, dept_id, address_id" +
            "FROM emp" +
            "START WITH manager_id =?" +
            "CONNECTER PAR PRIOR emp_id = manager_id",
    resultClass = Employee.class
)
```

L'un des avantages de l'utilisation de requêtes SQL nommées est que l'application peut utiliser le méthode createNamedQuery () sur l'interface EntityManager pour créer et exécuter la requête. Le fait que la requête nommée ait été définie en utilisant SQL au lieu de JP QL est pas important pour l'appelant. Un autre avantage est que createNamedQuery () peut renvoyer un TypedQuery alors que la méthode createNativeQuery () renvoie une requête non typée.

Référencement 11-4 montre à nouveau le bean de structure de rapport, cette fois en utilisant un requête nommée. L'autre avantage d'utiliser des requêtes nommées au lieu de requêtes dynamiques est qu'ils peuvent être remplacés à l'aide de fichiers de mappage XML. Une requête spécifiée à l'origine en JP QL peut être remplacé par une version SQL, et vice versa. Cette technique est décrite dans Chapitre 13.

Annonce 11-4. Exécution d'une requête SQL nommée

```
@Apatride
La classe publique OrgStructureBean implémente OrgStructure {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;
```


Épisode 502


```
    Liste publique <Employee> findEmployeesReportingTo (int managerId) {
        return em.createNamedQuery ("orgStructureReportingTo",
                                    Employé.classe)
                .setParameter (1, managerId)
                .getResultList ();
    }
}
```

Une chose à laquelle il faut faire attention avec les requêtes SQL qui renvoient des entités est que le résultat les instances d'entité sont gérées par le contexte de persistance, tout comme les résultats d'un Requête JP QL. Si vous modifiez l'une des entités renvoyées, elle sera écrite dans la base de données lorsque le contexte de persistance est associé à une transaction. C'est normalement ce que vous voulez, mais cela nécessite que chaque fois que vous sélectionnez des données qui correspondent à instances d'entité, il est important de s'assurer que toutes les données nécessaires requises pour construire l'entité fait partie de la requête. Si vous omettez un champ de la requête, ou la valeur par défaut, puis modifiez l'entité résultante, il est possible que vous écraserez la version correcte déjà stockée dans la base de données. Ceci est dû au fait l'état manquant sera nul (ou une valeur par défaut selon le type) dans l'entité. Lorsque la transaction est validée, le contexte de persistance ne sait pas que l'état n'a pas été correctement lu à partir de la requête et pourrait simplement essayer d'écrire null ou le valeur par défaut.

Il y a deux avantages à récupérer des entités gérées à partir d'une requête SQL. le tout d'abord, une requête SQL peut remplacer une requête JP QL existante et ce code d'application devrait toujours fonctionner sans modifications. Le deuxième avantage est qu'il permet au développeur de utiliser les requêtes SQL comme méthode de construction de nouvelles instances d'entité à partir de tables qui peuvent pas de mappage objet-relationnel. Par exemple, dans de nombreuses architectures de bases de données, il y a une zone de transit pour contenir des données qui n'ont pas encore été vérifiées ou qui nécessitent une sorte de transformation avant de pouvoir être déplacé vers son emplacement final. Utilisation de JPA, un développeur pourrait démarrer une transaction, interroger les données intermédiaires pour construire des entités, effectuer modifications requises, puis validez. Les entités nouvellement créées seront écrites dans le tables mappées par l'entité, pas les tables intermédiaires utilisées dans la requête SQL. C'est plus attrayant que l'alternative d'avoir un deuxième ensemble de mappages qui mappe le même entités (ou pire encore, un deuxième ensemble parallèle d'entités) aux tables de transfert, puis écrire du code qui lit, copie et réécrit les entités.

Épisode 503

Les instructions de manipulation de données SQL (INSERT, UPDATE et DELETE) sont également prises en charge par commodité pour que les appels JDBC n'aient pas à être introduits dans une application sinon limité à JPA. Pour définir une telle requête, utilisez la méthode createNativeQuery (), mais sans aucune information cartographique. Référencement11-5 illustre ces types de requêtes sous la forme d'un bean session qui enregistre les messages dans une table. Notez que les méthodes bean s'exécuter dans un contexte de transaction REQUIRES_NEW pour vous assurer que le message est consigné même si un la transaction active est annulée.

Annonce 11-5. Utilisation des instructions SQL INSERT et DELETE

```
@Apatride
@TransactionAttribute (TransactionAttributeType.REQUIRES_NEW)
public class LoggerBean implémente Logger {
    Chaîne finale statique privée INSERT_SQL =
        "INSERT INTO message_log (id, message, log_dttm)" +
        "VALEURS (id_seq.nextval,?, SYSDATE)";
    Chaîne finale statique privée DELETE_SQL =
        "SUPPRIMER DU journal_message";

    @PersistenceContext (unitName = "Logger")
    EntityManager em;

    public void logMessage (String message) {
        em.createNativeQuery (INSERT_SQL)
          .setParameter (1, message)
          .executeUpdate ();
    }

    public void clearMessageLog () {
        em.createNativeQuery (DELETE_SQL)
          .executeUpdate ();
    }
}
```

L'exécution d'instructions SQL qui modifient les données des tables mappées par des entités est généralement découragés. Cela pourrait rendre les entités mises en cache incompatibles avec le base de données car le fournisseur ne peut pas suivre les modifications apportées à l'état de l'entité qui a été modifié par des instructions de manipulation de données.

Épisode 504

Mappage de l'ensemble de résultats SQL

Dans les exemples de requêtes SQL présentés jusqu'à présent, le mappage des résultats était simple. le alias de colonne dans la chaîne SQL correspondant directement à la colonne relationnelle objet mappage pour une seule entité. Ce n'est pas toujours le cas que les noms correspondent, ni toujours le cas où un seul type d'entité est renvoyé. JPA fournit un jeu de résultats SQL mappages pour gérer ces scénarios.

Un mappage d'ensemble de résultats SQL est défini à l'aide de l'annotation @SqlResultSetMapping. Il peut être placé sur une classe d'entité et se compose d'un nom (unique dans la persistance unit) et un ou plusieurs mappages d'entités et de colonnes. L'argument de classe de résultat d'entité sur la méthode createNativeQuery () est vraiment un raccourci pour spécifier un résultat SQL simple définir le mappage. Le mappage suivant équivaut à spécifier Employee.class dans un appel pour createNativeQuery ():

```
@SqlResultSetMapping (
    name = "EmployeeResult",
```

entités = @ EntityResult (entityClass = Employee.class)
)

Ici, nous avons défini un mappage de jeu de résultats SQL appelé EmployeeResult qui peut être référencé par toute requête renvoyant des instances d'entité Employee. La cartographie se compose d'un résultat d'entité unique, spécifié par l'annotation @EntityResult, qui référence la classe d'entité Employee. La requête doit fournir des valeurs pour toutes les colonnes mappées par l'entité, y compris les clés étrangères. Il est spécifique au fournisseur si l'entité est partiellement construit ou si une erreur se produit si un état d'entité requis est manquant.

Mappage de clés étrangères

Les clés étrangères n'ont pas besoin d'être explicitement mappées dans le cadre du mappage de l'ensemble de résultats SQL. Lorsque le moteur de requête tente de mapper les résultats de la requête à une entité, il considère colonnes de clé étrangère pour les associations à valeur unique également. Regardons le reporting structure à nouveau la requête SQL.

```
SELECT emp_id, nom, salaire, manager_id, dept_id, address_id
DE EMP
START WITH manager_id IS NULL
CONNECTER PAR PRIOR emp_id = manager_id
```


Épisode 505

Les colonnes MANAGER_ID, DEPT_ID et ADDRESS_ID sont toutes mappées aux colonnes de jointure de associations sur l'entité Employé. Une instance Employee renvoyée par cette requête peut utilisez les méthodes getManager (), getDepartment () et getAddress (), et les résultats sera comme prévu. Le fournisseur de persistance récupérera l'entité associée en fonction sur la valeur de clé étrangère lue à partir de la requête. Il n'y a aucun moyen de remplir la collection associations à partir d'une requête SQL. Les instances d'entité construites à partir de cet exemple sont effectivement les mêmes qu'ils auraient été s'ils avaient été renvoyés d'une requête JP QL.

Mappages de résultats multiples

Une requête peut renvoyer plus d'une entité à la fois. Ceci est le plus souvent utile s'il y a est une relation univoque entre deux entités; sinon, la requête aboutira à dupliquer les instances d'entité. Considérez la requête suivante:

```
SELECT emp_id, nom, salaire, manager_id, dept_id, address_id,
        identifiant, rue, ville, état, zip
FROM emp, adresse
WHERE id_adresse = id
```

Le mappage de l'ensemble de résultats SQL pour renvoyer les entités Employee et Address hors de cette requête est défini dans la liste 11-6 . Chaque entité est répertoriée dans un @EntityResult annotation, dont un tableau est affecté à l'élément entity. L'ordre dans lequel les entités sont répertoriées n'est pas important. Le moteur de requête utilise les noms de colonne du requête pour correspondre aux données de mappage d'entité, pas à la position de la colonne.

Annonce 11-6. Mappage d'une requête SQL qui renvoie deux types d'entité

```
@SqlResultSetMapping (
    name = "EmployeeWithAddress",
    entités = {@ EntityResult (entityClass = Employee.class),
               @EntityResult (entityClass = Address.class)}
)
```

Mappage d'alias de colonne

Si les alias de colonne de l'instruction SQL ne correspondent pas directement aux noms spécifiés dans les mappages de colonnes pour l'entité, les mappages de résultats de champ sont requis pour le moteur de recherche pour faire l'association correcte. Supposons, par exemple, que les deux

Épisode 506

Les tables EMP et ADDRESS répertoriées dans l'exemple précédent utilisaient l'ID de colonne pour leur clé primaire. La requête devrait être modifiée pour aliaser les colonnes ID afin qu'elles soient unique:

```
SELECT emp.id AS emp_id, nom, salaire, manager_id, dept_id, address_id,
            address.id, rue, ville, état, code postal
FROM emp, adresse
WHERE id_adresse = adresse.id
```

L'annotation @FieldResult est utilisée pour mapper les alias de colonne aux attributs d'entité dans les situations où le nom de la requête n'est pas le même que celui utilisé dans la colonne cartographie. Le Listing 11-7 montre le mappage requis pour convertir l'alias EMP_ID en attribut id de l'entité. Plusieurs @FieldResult peuvent être spécifiés, mais seul le les mappages qui sont différents doivent être spécifiés. Cela peut être une liste partielle d'entités les attributs.

Annonce 11-7. Mappage d'une requête SQL avec des alias de colonne inconnus

```
@SqlResultSetMapping (
      name = "EmployeeWithAddress",
      entités = {@ EntityResult (entityClass = Employee.class,
                                 champs = @ FieldResult (
                                       name = "id",
                                       colonne = "EMP_ID")),
                 @EntityResult (entityClass = Address.class)}
)
```

Mappage des colonnes de résultats scalaires

Les requêtes SQL ne se limitent pas à renvoyer uniquement les résultats d'entité, bien que l'on s'attende à ce que ce sera le cas d'utilisation principal. Considérez la requête suivante:

```
SELECT e.name AS nom_emp, m.name AS nom_gérant
DE EMP e,
      emp m
O e.manager_id = m.emp_id (+)
COMMENCER AVEC e.manager_id EST NULL
CONNECTER PAR PRIOR e.emp_id = e.manager_id
```


Épisode 507

Les types de résultats non-entité, appelés types de résultats scalaires, sont mappés à l'aide de @ColumnResult annotation. Un ou plusieurs mappages de colonnes peuvent être affectés à l'attribut colonnes de l'annotation de mappage. Le seul attribut disponible pour un mappage de colonne est le

nom de colonne. Référencement11-8 montre le mappage SQL pour l'employé et le gestionnaire
requête hiérarchique.

Annonce 11-8. Mappages de colonnes scalaires

```
@SqlResultSetMapping (
      name = "EmployeeAndManager",
      colonnes = {@ ColumnResult (name = "EMP_NAME"),
                  @ColumnResult (name = "MANAGER_NAME")}
)
```

Les résultats scalaires peuvent également être mélangés avec des entités. Dans ce cas, les résultats scalaires sont
fournissant généralement des informations supplémentaires sur l'entité.

Regardons un exemple plus complexe dans lequel ce serait le cas. Un rapport pour un
l'application a besoin de voir des informations sur chaque département, montrant le responsable, le
le nombre d'employés et le salaire moyen. La requête JP QL suivante produit le
rapport correct:

```
SELECT d, m, COUNT (e), AVG (e.salary)
DU Département d GAUCHE REJOINDRE d. Employés e
                        REJOINDRE GAUCHE d. Employés m
O m EST NULL OU m IN (SELECT de.manager
                              DE Employé de
                              WHERE de.department = d)
GROUPE PAR d, m
```

Cette requête est particulièrement difficile car il n'y a pas de relation directe entre
Département à l'Employé qui est le responsable du département. Par conséquent, la
relation de salariés doit être rejointe deux fois: une fois pour les salariés affectés au
département et une fois pour l'employé de ce groupe qui est également le directeur. C'est
possible car la sous-requête réduit la deuxième jointure de la relation d'employés à un
résultat unique. Nous devons également tenir compte du fait qu'il pourrait ne pas y avoir d'employés
actuellement affecté au département, et en outre qu'un département pourrait ne pas avoir
gestionnaire affecté. Cela signifie que chacune des jointures doit être une jointure externe et que nous
doivent en outre utiliser une condition OR pour autoriser le gestionnaire manquant dans la clause WHERE.

Épisode 508

Une fois en production, il est déterminé que la requête SQL générée par le fournisseur
ne fonctionne pas bien, le DBA propose donc une requête alternative qui tire parti de
les vues en ligne possibles avec la base de données Oracle. La requête pour obtenir ce résultat est
montré dans le listing 11-9.

Annonce 11-9. Requête récapitulative du service

```
SELECT d.id, d.name AS nom_dépt,
          e.emp_id, e.name, e.salary, e.manager_id, e.dept_id,
          e.address_id,
          s.tot_emp, s.avg_sal
DU département d,
       (SÉLECTIONNER *
         DE EMP e
         WHERE EXISTS (SELECT 1 FROM emp WHERE manager_id = e.emp_id)) e,
       (SELECT d.id, COUNT (*) AS tot_emp, AVG (e.salary) AS avg_sal
         DEpt d, emp e
         O d.id = e.dept_id (+)
         GROUP BY d.id) s
O d.id = e.dept_id (+) ET
```

```
        d.id = s.id
```

Heureusement, il est beaucoup plus facile de cartographier cette requête que de la lire. Les résultats de la requête se composent d'une entité départementale; une entité Employé; et deux résultats scalaires, le nombre de les employés et le salaire moyen. Le listing 11-10 montre le mappage de cette requête.

Annonce 11-10. Mappage pour la requête de service

```
@SqlResultSetMapping (
    name = "DepartmentSummary",
    entités = {
        @EntityResult (entityClass = Department.class,
                       fields = @ FieldResult (name = "name", column = "DEPT_NAME")),
        @EntityResult (entityClass = Employee.class)
    },
    colonnes = {@ ColumnResult (name = "TOT_EMP"),
                @ColumnResult (nom = "AVG_SAL")}
)
```

Épisode 509

Mappage des clés composées

Lorsqu'une clé primaire ou étrangère est composée de plusieurs colonnes ayant un alias aux noms non mappés, une notation spéciale doit être utilisée dans les annotations @FieldResult pour identifier chaque partie de la clé. Considérez la requête affichée dans la liste11-11 qui retourne à la fois l'employé et le directeur de l'employé. Le tableau de cet exemple est le même que nous avons démontré dans la figure 10-4 du chapitre 10 . Parce que chaque colonne est répété deux fois, les colonnes de l'état du gestionnaire ont été associées à de nouveaux noms.

Annonce 11-11. SQL Query Returning Employee and Manager

```
SELECT e.country, e.emp_id, e.name, e.salary,
       e.manager_country, e.manager_id, m.country AS mgr_country,
       m.emp_id AS mgr_id, m.name AS mgr_name, m.salary AS mgr_salary,
       m.manager_country AS mgr_mgr_country, m.manager_id AS mgr_mgr_id
DE EMP e,
       emp m
O e.manager_country = m.country ET
       e.manager_id = m.emp_id
```

Le mappage de l'ensemble de résultats pour cette requête dépend du type de classe de clé primaire utilisée par l'entité cible. Référencement11-12 montre le mappage dans le cas où une classe id a déjà utilisé. Pour la clé primaire, chaque attribut est répertorié en tant que résultat de champ distinct. Pour le clé étrangère, chaque attribut de clé primaire de l'entité cible (l'entité Employee à nouveau cet exemple) est suffixé au nom de l'attribut de relation.

Annonce 11-12. Mappage pour la requête des employés à l'aide de la classe d'ID

```
@SqlResultSetMapping (
    name = "EmployeeAndManager",
    entités = {
        @EntityResult (entityClass = Employee.class),
        @EntityResult (
            entityClass = Employee.class,
            champs = {
                @FieldResult (nom = "pays", colonne = "MGR_COUNTRY"),
                @FieldResult (nom = "id", colonne = "MGR_ID"),
```

```
                @FieldResult (nom = "nom", colonne = "MGR_NAME"),
```


Épisode 510


```
                @FieldResult (nom = "salaire", colonne = "MGR_SALARY"),
                @FieldResult (nom = "manager.country",
                        colonne = "MGR_MGR_COUNTRY"),
                @FieldResult (nom = "manager.id", colonne = "MGR_MGR_ID")
            }
        )
    }
)
```

Si Employee utilise une classe id intégrée au lieu d'une classe id, la notation est légèrement différent. Nous devons inclure le nom de l'attribut de clé primaire ainsi que l'individu attributs dans le type incorporé. Référencement11-13 montre le mappage de l'ensemble de résultats en utilisant cette notation.

Annonce 11-13. Mappage pour la requête des employés à l'aide de la classe d'identifiant intégrée

```
@SqlResultSetMapping (
    name = "EmployeeAndManager",
    entités = {
        @EntityResult (entityClass = Employee.class),
        @EntityResult (
            entityClass = Employee.class,
            champs = {
                @FieldResult (nom = "id.country", colonne = "MGR_COUNTRY"),
                @FieldResult (nom = "id.id", colonne = "MGR_ID"),
                @FieldResult (nom = "nom", colonne = "MGR_NAME"),
                @FieldResult (nom = "salaire", colonne = "MGR_SALARY"),
                @FieldResult (nom = "manager.id.country",
                        colonne = "MGR_MGR_COUNTRY"),
                @FieldResult (nom = "manager.id.id", colonne = "MGR_MGR_ID")
            }
        )
    }
)
```


Épisode 511

Mappage de l'héritage

À bien des égards, les requêtes polymorphes en SQL ne sont pas différentes des requêtes régulières renvoyer un seul type d'entité. Toutes les colonnes doivent être prises en compte, y compris les clés étrangères et la colonne de discrimination pour les stratégies d'héritage à table unique et jointe. le la chose clé à retenir est que si les résultats incluent plus d'un type d'entité, chacun des les colonnes de tous les types d'entités possibles doivent être représentées dans la requête. Le champ

les techniques de mappage des résultats présentées précédemment peuvent être utilisées pour personnaliser les colonnes qui utilisent des alias inconnus. Ces colonnes peuvent être à n'importe quel niveau de l'arborescence d'héritage. Le seul élément spécial de l'annotation @EntityResult à utiliser avec l'héritage est l'élément discriminatorColumn. Cet élément permet le nom du discriminateur colonne à spécifier dans le cas peu probable où elle est différente de la version mappée.

Supposons que l'entité Employé a été mappée au tableau illustré à la figure 10-11 du chapitre dix. Pour comprendre l'aliasing d'une colonne discriminante, considérez ce qui suit requête qui renvoie des données d'une autre table EMPLOYEE_STAGE structurée pour utiliser une seule table héritage:

```
SELECT id, nom, date_début, taux_jour, terme, vacances,
       taux_horaire, salaire, pension, type
FROM stage_employé
```

Pour convertir les données renvoyées par cette requête en entités Employee, les éléments suivants le mappage de l'ensemble de résultats serait utilisé:

```
@SqlResultSetMapping (
    name = "EmployeeStageMapping",
    entités =
        @EntityResult (
            entityClass = Employee.class,
            discriminatorColumn = "TYPE",
            champs = {
                @FieldResult (name = "startDate", column = "START_DATE"),
                @FieldResult (name = "dailyRate", column = "DAILY_RATE"),
                @FieldResult (name = "hourlyRate", column = "HOURLY_RATE")
            }
        )
)
```

Épisode 512

Mappage vers des types non-entité

Depuis JPA 2.1, la possibilité de construire des types non-entité à partir de requêtes natives via un constructeur des expressions ont été ajoutées. Expressions de constructeur dans les requêtes natives, tout comme expressions de constructeur de JP QL, aboutissent à l'instanciation des types spécifiés par l'utilisateur par en utilisant les données de ligne du jeu de résultats sous-jacent pour appeler le constructeur. Toutes les données requis pour construire l'objet doit être représenté comme des arguments du constructeur.

Considérez l'exemple d'expression de constructeur que nous avons présenté au chapitre 8 à créer des instances du type de données EmployeeDetails en sélectionnant des champs spécifiques dans le Entité salariée:

```
SELECT NEW example.EmployeeDetails (e.name, e.salary, e.department.name)
DE Employé e
```

Pour remplacer cette requête par son équivalent natif et obtenir le même résultat, nous devons définir à la fois la requête native de remplacement et un SqlResultSetMapping qui définit la manière dont les résultats de la requête correspondent au type spécifié par l'utilisateur. Commençons par définir le Requête native SQL pour remplacer le JP QL.

```
SELECT e.name, e.salary, d.name AS deptName
FROM emp e, dépt d
O e.dept_id = d.id
```

Le mappage de cette requête est plus détaillé que l'équivalent JP QL. Contrairement à JP QL où les colonnes sont implicitement mappées au constructeur en fonction de leur position,

les requêtes natives doivent définir complètement l'ensemble de données qui seront mappées au constructeur dans l'annotation de mappage. L'exemple suivant illustre le mappage de cette requête:

```
@SqlResultSetMapping (
    name = "EmployeeDetailMapping",
    classes = {
        @ConstructorResult (
            targetClass = exemple.EmployeeDetails.class,
            colonnes = {
                @ColumnResult (nom = "nom"),
                @ColumnResult (nom = "salaire", type = Long.classe),
                @ColumnResult (nom = "deptName")
            })})
```


Épisode 513

Comme avec d'autres exemples qui utilisent ColumnResult, le champ de nom fait référence à la colonne alias tel que défini dans l'instruction SQL. Les résultats de la colonne sont appliqués au constructeur du type spécifié par l'utilisateur dans l'ordre dans lequel les mappages de résultats de colonne sont définis. Dans les cas où plusieurs constructeurs peuvent être ambigus en fonction de la position uniquement, le type de résultat de la colonne peut également être spécifié afin de garantir la correspondance correcte.

Il est à noter que si un type d'entité est spécifié comme classe cible d'un Mappage ConstructorResult, toutes les instances d'entité résultantes seraient considérées non géré par le contexte de persistance actuel. Mappages d'entités sur les types spécifiés par l'utilisateur sont ignorés lorsqu'ils sont traités dans le cadre d'une requête native à l'aide d'expressions de constructeur.

Liaison de paramètres

Les requêtes SQL ont traditionnellement pris en charge uniquement la liaison de paramètres positionnels. Le JDBC la spécification elle-même ne prend en charge que les paramètres nommés sur les objets CallableStatement, pas PreparedStatement, et tous les fournisseurs de bases de données ne prennent même pas en charge cette syntaxe. Par conséquent, JPA garantit uniquement l'utilisation de la liaison de paramètres positionnels pour les requêtes SQL. Vérifier avec votre fournisseur pour voir si les méthodes de paramètres nommés de l'interface de requête sont pris en charge, mais comprenez que leur utilisation peut rendre votre application non portable entre les fournisseurs de persistance.

Une autre limitation de la prise en charge des paramètres pour les requêtes SQL est que les paramètres d'entité Ne peut pas être utilisé. La spécification ne définit pas comment ces types de paramètres doivent être traité. Soyez prudent lors de la conversion ou du remplacement d'une requête JP QL nommée avec un SQL natif demande que les valeurs des paramètres sont toujours interprétées correctement.

Procédures stockées

Depuis JPA 2.1, la possibilité de mapper et d'appeler des procédures stockées à partir d'une base de données était ajoutée. JPA 2.1 a également introduit un support de première classe pour les requêtes de procédure stockée sur le EntityManager et définit un nouveau type de requête, StoredProcedureQuery, qui étend la requête et gère mieux la gamme d'options ouvertes aux développeurs qui exploitent les procédures stockées dans leurs applications. Aucune nouvelle fonctionnalité liée aux procédures stockées n'a été ajoutée dans JPA 2.2.

Les sections suivantes décrivent comment mapper et appeler des requêtes de procédure stockée en utilisant JPA. Notez que l'implémentation des procédures stockées est fortement basée sur la base de données dépendante et sort donc du cadre de ce livre. Contrairement à JP QL et à d'autres types des requêtes natives, le corps d'une procédure stockée n'est jamais défini dans JPA et est à la place jamais référencé par son nom dans l'application.

Épisode 514

Définition et exécution de requêtes de procédure stockée

Comme d'autres types de requêtes JPA, des requêtes de procédure stockée peuvent être créées par programme à partir de l'interface EntityManager ou ils peuvent être définis à l'aide de métadonnées d'annotation et référencées ultérieurement par leur nom. Afin de définir une procédure stockée mapping, le nom de la procédure stockée doit être fourni ainsi que le nom et type de tous les paramètres de cette procédure stockée.

Les définitions de procédure stockée JPA prennent en charge les principaux types de paramètres définis pour Procédures stockées JDBC: IN, OUT, INOUT et REF_CURSOR. Comme leur nom l'indique, le Les types de paramètres IN et OUT transmettent des données à la procédure stockée ou les renvoient à l'appelant, respectivement. Les types de paramètres INOUT combinent les comportements IN et OUT en un seul type qui peut à la fois accepter et renvoyer une valeur à l'appelant. Les types de paramètres REF_CURSOR sont utilisés pour renvoyer les jeux de résultats à l'appelant. Chacun de ces types a une valeur enum correspondante défini sur le type ParameterMode.

Les procédures stockées sont supposées renvoyer toutes les valeurs via des paramètres sauf dans le cas où la base de données prend en charge le renvoi d'un jeu de résultats (ou de plusieurs jeux de résultats) à partir de une procédure stockée. Bases de données qui prennent en charge cette méthode de renvoi d'ensembles de résultats généralement faites-le comme alternative à l'utilisation des types de paramètres REF_CURSOR. Malheureusement, stocké le comportement de la procédure est très spécifique au fournisseur, et ceci est un exemple où l'écriture dans un une caractéristique particulière d'une base de données est inévitable dans le code d'application.

Types de paramètres scalaires

Pour commencer, considérons une procédure stockée simple nommée "bonjour" qui accepte une seule chaîne argument nommé "nom" et renvoie un message d'accueil amical à l'appelant via le même paramètre. L'exemple suivant montre comment définir et exécuter une telle procédure stockée:

```
StoredProcedureQuery q = em.createStoredProcedureQuery ("bonjour");
q.registerStoredProcedureParameter ("nom", String.class, ParameterMode.INOUT);
q.setParameter ("nom", "massimo");
q.execute ();
Valeur de chaîne = (chaîne) q.getOutputParameterValue ("nom");
```

Une fois la requête exécutée via la méthode execute (), tout paramètre IN ou INOUT les valeurs renvoyées à l'appelant sont accessibles en utilisant l'un des getOutputParameterValue () méthodes, en spécifiant le nom du paramètre ou sa position. Comme pour les requêtes natives, les positions des paramètres sont numérotées en commençant par un. Par conséquent, si un paramètre OUT est

Épisode 515

inscrit en deuxième position pour une requête, alors le numéro deux serait utilisé pour accéder cette valeur, même si le premier paramètre est de type IN et ne renvoie aucune valeur.

Notez que l'appel des méthodes getSingleResult () ou getResultList () du l'interface de requête héritée entraînera une exception si la procédure stockée ne renvoie que valeurs scalaires via des paramètres.

Types de paramètres de l'ensemble de résultats

Passons maintenant à un exemple plus sophistiqué dans lequel nous avons un procédure nommée fetch_emp qui récupère les données des employés et les renvoie au appelant via un paramètre REF_CURSOR. Dans ce cas, nous utilisons la forme surchargée de

createStoredProcedureQuery () pour indiquer que le jeu de résultats se compose de Employee
entités. L'exemple suivant illustre cette approche:

```
StoredProcedureQuery q = em.createStoredProcedureQuery ("fetch_emp");
q.registerStoredProcedureParameter ("empList", void.class, ParameterMode.
REF_CURSOR);
if (q.execute ()) {
    List <Employee> emp = (List <Employee>) q.getOutputParameterValue ("empList");
    // ...
}
```

Le premier point à noter à propos de cet exemple est que nous ne spécifions pas de paramètre
type de classe pour le paramètre empList. Chaque fois que REF_CURSOR est spécifié comme
type de paramètre, l'argument de type de classe à registerStoredProcedureParameter () est
ignoré. Le deuxième point à noter est que nous vérifions la valeur de retour de execute ()
pour voir les jeux de résultats renvoyés lors de l'exécution de la requête. Si aucun jeu de résultats
sont retournés (ou la requête ne renvoie que des valeurs scalaires), execute () retournera false. dans le
événement où aucun enregistrement n'a été renvoyé et que nous n'avons pas vérifié le résultat de execute (), alors
la valeur du paramètre serait nulle.

Auparavant, il a été noté que certaines bases de données permettent de renvoyer des ensembles de résultats
procédures stockées sans avoir à spécifier un paramètre REF_CURSOR. Dans ce cas, nous
peut simplifier l'exemple en utilisant la méthode getResultList () de l'interface de requête pour
accéder directement aux résultats:

```
StoredProcedureQuery q = em.createStoredProcedureQuery ("fetch_emp");
List <Employee> emp = (List <Employee>) q.getResultList ();
// ...
```

Épisode 516

En tant que raccourci pour exécuter des requêtes, getResultList () et getSingleResult ()
invoque implicitement execute (). Dans le cas où la procédure stockée renvoie plus
qu'un jeu de résultats, chaque appel à getResultList () renverra le jeu de résultats suivant dans le
séquence.

Mappage des procédures stockées

Les requêtes de procédure stockées peuvent être déclarées à l'aide de @NamedStoredProcedureQuery
annotation puis référencée par son nom à l'aide de createNamedStoredProcedureQuery ()
méthode sur EntityManager. Cela simplifie le code requis pour appeler la requête et
permet des mappages plus complexes que ce qui est possible avec StoredProcedureQuery
interface. Comme les autres types de requêtes JPA, les noms des requêtes de procédure stockée doivent être uniques
dans le cadre d'une unité de persistance.

Nous commençons par déclarer le simple exemple "bonjour" en utilisant des annotations, puis
passer progressivement à des exemples plus complexes. La procédure stockée "bonjour" serait
mappé comme suit:

```
@NamedStoredProcedureQuery (
    name = "bonjour",
    procedureName = "bonjour",
    paramètres = {
        @StoredProcedureParameter (nom = "nom", type = String.class,
                                   mode = ParameterMode.INOUT)
    })
```

Ici, nous pouvons voir que le nom de la procédure native doit être spécifié en plus
au nom de la requête de procédure stockée. Il n'est pas défini par défaut comme lors de l'utilisation
la méthode createStoredProcedureQuery (). Comme pour l'exemple programmatique,

tous les paramètres doivent être spécifiés en utilisant les mêmes arguments que ceux qui seraient utilisés avec registerStoredProcedureParameter ().

La requête "fetch_emp" est mappée de la même manière, mais dans cet exemple, nous devons également spécifiez un mappage pour le type d'ensemble de résultats:

```
@NamedStoredProcedureQuery (
    name = "fetch_emp",
    procedureName = "fetch_emp",
```


Épisode 517


```
    paramètres = {
        @StoredProcedureParameter (nom = "empList", type = void.class,
                                   mode = ParameterMode.REF_CURSOR)
    },
    resultClasses = Employee.class)
```

La liste des classes fournie dans le champ resultClasses est une liste de types d'entités. Dans la version de cet exemple où REF_CURSOR n'est pas utilisé et les résultats sont renvoyés directement à partir de la procédure stockée, le paramètre empList serait simplement omis. Le champ resultClasses serait suffisant. Il est important de noter que les entités référencées dans resultClasses doivent correspondre à l'ordre dans lequel les paramètres du jeu de résultats sont déclarées pour la procédure stockée. Par exemple, s'il y a deux REF_CURSOR paramètres — empList et deptList — alors le champ resultClasses doit contenir Employé et service dans cet ordre.

Pour les cas où les ensembles de résultats renvoyés par la requête ne sont pas mappés nativement à l'entité types, les mappages d'ensemble de résultats SQL peuvent également être inclus dans NamedStoredProcedureQuery annotation à l'aide du champ resultSetMappings. Par exemple, une procédure stockée qui a effectué la requête d'employé et de responsable définie dans le Listing 11-11 (et mappée dans Référencement 11-12) serait mappé comme suit:

```
@NameStoredProcedureQuery (
    name = "queryEmployeeAndManager",
    procedureName = "fetch_emp_and_mgr",
    resultSetMappings = "EmployeeAndManager"
)
```

Comme avec resultClasses, plusieurs mappages de jeux de résultats peuvent également être spécifiés si la procédure stockée renvoie plusieurs jeux de résultats. Il faut cependant noter que la combinaison de resultClasses et resultSetMappings n'est pas définie. Prise en charge de l'ensemble de résultats mappages dans l'annotation NamedStoredProcedureQuery garantit que même le plus les définitions de procédures stockées complexes peuvent probablement être mappées par JPA, ce qui simplifie l'accès et centraliser les métadonnées de manière gérable.

Épisode 518

Graphiques d'entité

Contrairement à ce que son nom implique, un graphe d'entités n'est pas vraiment un graphe d'entités, mais plutôt un graphe d'entités modèle pour spécifier les attributs d'entité et incorporables. Il sert de modèle qui peut être transmis à une méthode ou une requête de recherche pour spécifier l'entité et les attributs intégrables doit être récupéré dans le résultat de la requête. Plus concrètement, les graphes d'entités sont utilisés pour remplacer lors de l'exécution les paramètres de récupération des mappages d'attributs. Par exemple, si un attribut est mappé pour être récupéré avec impatience (défini sur FetchType.EAGER), il peut être défini pour être récupéré paresseusement pour une seule exécution d'une requête. Cette fonctionnalité est similaire à ce qui a été évoqué par certains comme la possibilité de définir un plan de récupération, un plan de chargement ou un groupe de récupération.

> Notez bien que les graphiques d'entités sont actuellement utilisés pour remplacer l'attribut extraire l'état dans les requêtes, ce ne sont que des structures qui définissent l'inclusion d'attributs. Là aucune sémantique inhérente n'y est stockée. ils pourraient tout aussi bien être utilisés comme entrée à toute opération qui pourrait bénéficier d'un modèle d'attribut d'entité, et dans à l'avenir, ils pourraient également être utilisés avec d'autres opérations. cependant, nous nous concentrons sur ce qui existe actuellement et discutez des graphiques d'entités dans le contexte de la définition chercher des plans.

La structure d'un graphe d'entité est assez simple en ce qu'il n'y a en réalité que trois types d'objets qui y sont contenus.

- *Nœuds de graphe d'entité* : il existe exactement un nœud de graphe d'entité pour chaque graphe d'entité. C'est la racine du graphe d'entité et représente la racine type d'entité. Il contient tous les nœuds d'attribut pour ce type, plus tous des nœuds de sous-graphe pour les types associés au type d'entité racine[1] .
- *Nœuds d'attribut* : chaque nœud d'attribut représente un attribut dans un entité ou type intégrable. Si c'est pour un attribut de base, alors là ne sera pas associé à un sous-graphe, mais s'il s'agit d'une relation attribut, attribut intégrable ou collection d'éléments de embeddables, alors il peut faire référence à un sous-graphe nommé.

[1] Dans une annotation @NamedEntityGraph, il contiendra les nœuds de sous-graphe pour l'entité entière graphique.

Épisode 519

- *Nœuds de* sous-graphe: un nœud de sous-graphe est l'équivalent d'un graphe d'entité node en ce qu'il contient des nœuds d'attribut et des nœuds de sous-graphe, mais représente un graphique d'attributs pour une entité ou un type intégrable qui n'est pas le type racine [2].

Pour aider à illustrer la structure, supposons que nous ayons le modèle de domaine décrit dans la Figure 8-1 (Chapitre 8) et veulent une représentation graphique d'entité qui spécifie un sous-ensemble de ces attributs et entités, comme illustré dans la figure 11-1 .

L'état de la figure 11-1 est clairement un sous-ensemble de l'état de la figure 8-1 , et l'entité la structure graphique pour stocker un plan de récupération représentatif pour cet état peut ressembler à représentation sur la figure 11-2 .

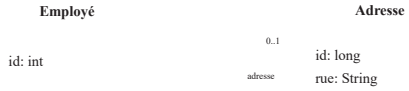

Figure 11-1. *Sous-ensemble de modèle de domaine*

Les graphiques d'entités peuvent être définis de manière statique sous forme d'annotation ou dynamiquement via une API. La composition diffère légèrement lors de l'utilisation du formulaire d'annotation de création les graphes d'entités par rapport à leur création avec l'API; cependant, figure 11-2 montre la base structure.

[2] Un sous-graphe peut être défini pour le type racine, mais dans ce cas, il s'agit d'un plan d'extraction supplémentaire utilisé lorsque le type d'entité racine est accédé via une relation.

Épisode 520

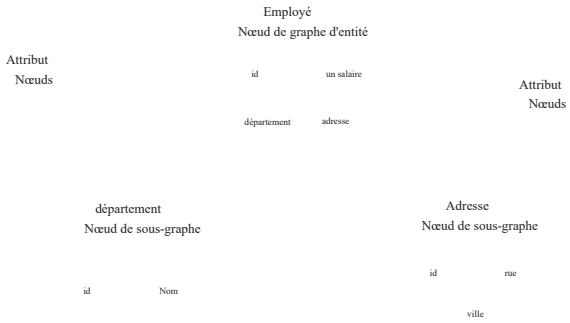

Figure 11-2. *État du graphe d'entité*

Un point clé à mentionner est que l'identifiant et les attributs de version (voir le chapitre 12 pour un description complète des attributs de version) sera toujours inclus dans l'entité nœud de graphe et chacun des nœuds de sous-graphe. Ils n'ont pas besoin d'être explicitement inclus lors de la création d'un graphe d'entité, bien que si l'un d'eux est inclus, ce n'est pas une erreur, juste une redondance.

Chaque entité et classe intégrable a un *graphique d'extraction par défaut* qui est composé du fermeture transitive de tous les attributs qui sont soit explicitement définis comme, soit par défaut être, recherché avec impatience. La partie fermeture transitive signifie que la règle est récursive appliquée, de sorte que le graphique d'extraction par défaut d'une entité n'inclut pas seulement tous les attributs désireux défini dessus, mais aussi tous les attributs désireux des entités qui lui sont liées, et ainsi jusqu'à ce que le graphe de relations soit épuisé. Le graphique d'extraction par défaut nous fera économiser un beaucoup d'efforts quand vient le temps de définir des graphiques d'entités plus vastes.

Annotations de graphique d'entité

La définition statique d'un graphique d'entité dans une annotation est utile lorsque vous savez à l'avance

de temps pendant lequel un modèle de récupération d'accès aux attributs doit être différent de ce qui a été configuré dans les mappages. Vous pouvez définir n'importe quel nombre de graphiques d'entités pour le même entité, chacune représentant un plan d'extraction d'attribut différent.

Épisode 521

L'annotation parente pour définir un graphique d'entité est @NamedEntityGraph. Ce doit être défini sur l'entité racine du graphe d'entité. Tous ses composants sont imbriqués dans le une annotation, donc cela signifie deux choses. Premièrement, si vous définissez un grand et impliqué graphique d'entité, alors votre annotation sera également volumineuse et impliquée (et peu probable Très belle). Deuxièmement, si vous définissez plusieurs graphiques d'entités, vous ne pouvez partager aucun des les parties de sous-graphe constituantes l'une avec l'autre [3]. Ils sont complètement encapsulés dans le graphe d'entité nommé englobant.

Vous avez probablement remarqué que l'annotation commence par le préfixe nommé. C'est un clair giveaway que les graphes d'entités sont nommés, et comme d'autres types d'objets nommés dans JPA, les noms doivent être uniques dans la portée de l'unité de persistance. Si non spécifié, le nom d'un graphe d'entité sera par défaut le nom d'entité (qui, comme vous vous en souviendrez Chapitre 2, est le nom non qualifié de la classe d'entité). Nous verrons plus tard comment un nom est utilisé pour faire référence à un graphe d'entité donné lors de son obtention auprès du gestionnaire d'entités.

Notez que les sous-graphes d'un graphe d'entité sont également nommés, mais ces noms sont valable uniquement dans le cadre du graphique d'entité et, comme vous le verrez dans la section suivante, sont utilisé pour connecter les sous-graphiques.

Graphiques d'attributs de base

Utilisons le modèle de domaine de la figure 8-1 comme point de départ. Dans ce modèle, il y a un Entité d'adresse avec quelques attributs de base. Nous pourrions créer un simple graphe d'entités nommées en annotant l'entité Address, comme indiqué dans l'extrait 11-14.

Annonce 11-14. Graphique d'entité nommée avec attributs de base

```
@Entité
@NamedEntityGraph (
    attributeNodes = {
        @NamedAttributeNode ("rue"),
        @NamedAttributeNode ("ville"),
        @NamedAttributeNode ("état"),
        @NamedAttributeNode ("zip")}
)
```

[3] Vous pouvez cependant partager des sous-graphiques dans le même graphique d'entité.

Épisode 522


```
Adresse de classe publique {
    @Id id long privé;
```

```
    rue privée String;
    ville privée de String;
    état de chaîne privé;
    zip de chaîne privé;
    // ...
}
```

Le graphique d'entité reçoit le nom par défaut de l'adresse et tous les attributs sont explicitement inclus dans l'élément attributeNodes, ce qui signifie qu'ils doivent être récupérés. Il existe en fait un raccourci pour ce cas au moyen d'un includeAllAttributes spécial élément dans l'annotation @NamedEntityGraph:

```
@Entité
@NamedEntityGraph (includeAllAttributes = true)
Adresse de la classe publique {...}
```

Étant donné que l'entité Address ne contient de toute façon que des attributs désireux (tous les attributs de base par défaut d'être impatient), nous pouvons le rendre encore plus court:

```
@Entité
@NamedEntityGraph
Adresse de la classe publique {...}
```

Annoter la classe sans lister d'attributs est un raccourci pour définir un graphe d'entité nommée composé du graphe d'extraction par défaut pour cette entité. Mettre l'annotation sur la classe entraîne la création du graphe d'entité nommé et référençable par nom dans une requête.

Utilisation des sous-graphes

L'entité Address était une entité assez simple, cependant, et n'avait même pas de des relations. L'entité Employé est plus compliquée et nécessite que nous ajoutions certains sous-graphiques, comme indiqué dans la liste 11-15 .

Épisode 523

Liste 11-15. Graphique d'entité nommée avec sous-graphiques

```
@Entité
@NamedEntityGraph (nom = "Employee.graph1",
    attributeNodes = {
        @NamedAttributeNode ("nom"),
        @NamedAttributeNode ("salaire"),
        @NamedAttributeNode (valeur = "adresse"),
        @NamedAttributeNode (valeur = "téléphones", subgraph = "téléphone"),
        @NamedAttributeNode (value = "department" subgraph = "dept")},
    sous-graphes = {
        @NamedSubgraph (nom = "téléphone",
            attributeNodes = {
                @NamedAttributeNode ("nombre"),
                @NamedAttributeNode ("type")}),
        @NamedSubgraph (nom = "dept",
            attributeNodes = {
                @NamedAttributeNode ("nom")})
```

```
    })
Employé de classe publique {
    @Id
    id int privé;
    nom de chaîne privé;
    long salaire privé;
    @Temporal (TemporalType.DATE)
    private Date startDate;

    @Un par un
    adresse d'adresse privée;

    @OneToMany (mappedBy = "employé")
    Collection privée <Phone> téléphones = new ArrayList <Phone> ();

    @ManyToOne
    département du département privé;
```


Épisode 524


```
    @ManyToOne
    gestionnaire d'employé privé;

    @OneToMany (mappedBy = "manager")
    Collection privée <Employee> dirige = new ArrayList <Employee> ();

    @ManyToMany (mappedBy = "employés")
    Private Collection <Project> projects = new ArrayList <Project> ();
    // ...
}
```

Pour chaque attribut Employee que nous voulons récupérer, il existe un @NamedAttributeNode listant le nom de l'attribut. L'attribut d'adresse est une relation, donc lister cela signifie que l'adresse sera récupérée, mais quel état sera récupéré dans le Instance d'adresse? C'est là que le groupe d'extraction par défaut entre en jeu. Lorsqu'une relation l'attribut est répertorié dans le graphique mais n'a pas de sous-graphique associé, alors le le groupe d'extraction par défaut de la classe associée sera utilisé. En fait, la règle générale est que pour tout type d'entité ou d'intégration qui a été spécifié pour être récupéré, mais pour lequel il n'est pas un sous-graphe, le graphe d'extraction par défaut pour cette classe sera utilisé. C'est comme tu pourrais attendez-vous puisque le graphique d'extraction par défaut spécifie le comportement que vous obtenez lorsque vous n'utilisez pas graphe d'entité du tout, de sorte que l'absence d'en spécifier un devrait être équivalente. Pour notre adresse Par exemple, vous avez vu que le graphe d'extraction par défaut de Address est tous ses attributs (de base).

Pour les autres attributs de relation, le @NamedAttributeNode inclut en plus un élément de sous-graphe, qui fait référence au nom d'un @NamedSubgraph. Le nommé le sous-graphe définit la liste d'attributs pour ce type d'entité associé, donc le sous-graphe que nous avons nommé dept définit les attributs de l'entité Department liés via l'attribut department de l'employé.

Tous les sous-graphes nommés sont définis dans l'élément sous-graphes de @NamedEntityGraph, quel que soit leur emplacement dans le graphique de type. Cela signifie que si le type de sous-graphe contenait un attribut de relation, puis son nœud d'attribut nommé contiendrait une référence à un autre sous-graphique, qui serait également répertorié dans le élément subgraphs de @NamedEntityGraph. On peut même définir un sous-graphe qui définit un plan de récupération alternatif pour notre classe de graphe d'entité racine lorsqu'elle est chargée via un entité liée. En fait, il y a des moments où nous devrons faire exactement cela.

Regardons un exemple dans l'extrait 11-16.

Épisode 525

Annonce 11-16. Graphique d'entités nommées avec plusieurs définitions de type

```
@Entité
@NamedEntityGraph (nom = "Employee.graph2",
    attributeNodes = {
        @NamedAttributeNode ("nom"),
        @NamedAttributeNode ("salaire"),
        @NamedAttributeNode (valeur = "adresse"),
        @NamedAttributeNode (valeur = "téléphones", subgraph = "téléphone"),
        @NamedAttributeNode (value = "manager", subgraph = "namedEmp"),
        @NamedAttributeNode (value = "department", subgraph = "dept")},
    sous-graphes = {
        @NamedSubgraph (nom = "téléphone",
            attributeNodes = {
                @NamedAttributeNode ("nombre"),
                @NamedAttributeNode ("type"),
                @NamedAttributeNode (value = "employee", subgraph = "namedEmp")}),
        @NamedSubgraph (nom = "namedEmp",
            attributeNodes = {
                @NamedAttributeNode ("nom")}),
        @NamedSubgraph (nom = "dept",
            attributeNodes = {
                @NamedAttributeNode ("nom")})
    })
Employé de classe publique {...}
```

Ce graphe d'entité nommée contient deux changements majeurs par rapport au précédent. La première le changement est que nous avons ajouté l'attribut manager à récupérer. Depuis le gérant est une entité Employee, vous pourriez être surpris que le sous-graphe namedEmp soit spécifié, penser que l'employé responsable serait simplement chargé conformément au plan de récupération décrit par le graphe d'entité nommée que nous définissons (c'est un graphe d'entité Employé, après tout). Ce n'est cependant pas ce que dit la règle. La règle est que sauf si un sous-graphe est spécifié pour un type d'attribut de relation, le graphique d'extraction par défaut pour ce type sera utilisé comme plan de récupération. Pour l'employé, cela signifierait que le gestionnaire aurait tous ses relations désireuses chargées, et toutes les relations désireuses de ses entités associées, et ainsi sur. Cela pourrait entraîner le chargement de beaucoup plus de données que prévu. La solution est de faites comme indiqué dans l'extrait 11-16 et spécifiez un sous-graphe minimal pour l'employé responsable.

Épisode 526

Le deuxième changement apporté à ce graphique d'entité est que le sous-graphique de téléphone inclut le attribut employé. Une fois de plus, nous référençons le sous-graphe namedEmp pour spécifier que l'employé n'est pas chargé selon le graphique d'extraction par défaut. La première chose à noter est que nous pouvons réutiliser le même sous-graphe nommé à plusieurs endroits dans le graphe d'entité. Après que vous devriez remarquer que l'attribut employé est en fait un pointeur arrière vers un employé

dans le jeu de résultats du graphe d'entité nommé. Nous voulons simplement nous assurer que le référencement du téléphone ne provoque pas la récupération de plus que ce qui est déjà défini par le graphe d'entité nommé employé, lui-même.

Graphiques d'entité avec héritage

Nous avons jusqu'ici évité l'aspect héritage de notre modèle d'entité, mais nous allons ne l'ignorez plus. Un graphique d'entité à échelle réduite qui inclut uniquement le nom de l'employé et les projets sont présentés dans le Listing 11-17.

Annonce 11-17. Graphique d'entité nommée avec héritage

```
@Entité
@NamedEntityGraph (nom = "Employee.graph3",
    attributeNodes = {
        @NamedAttributeNode ("nom"),
        @NamedAttributeNode (valeur = "projets", subgraph = "projet")},
    sous-graphes = {
        @NamedSubgraph (nom = "projet", type = Project.class,
            attributeNodes = {
                @NamedAttributeNode ("nom")}),
        @NamedSubgraph (nom = "projet", type = QualityProject.class,
            attributeNodes = {
                @NamedAttributeNode ("qaRating")})
    })
Employé de classe publique {...}
```

L'état d'extraction de l'attribut pour l'attribut projects est défini dans un sous-graphe appelé project, mais comme vous pouvez le voir, il existe deux sous-graphes nommés project. Chacun de les classes / sous-classes possibles qui pourraient être dans cette relation ont un sous-graphe nommé project et inclut son état défini qui doit être récupéré, plus un élément de type à identifier de quelle sous-classe il s'agit. Cependant, comme DesignProject n'introduit aucun nouveau state, nous n'avons pas besoin d'inclure un sous-graphe nommé project pour cette classe.

Épisode 527

Le cas d'héritage que cela ne couvre pas est celui où la classe d'entité racine est elle-même un superclasse. Pour ce cas particulier, il existe un élément subclassSubgraphs pour lister la racine sous-classes d'entités. Par exemple, s'il y avait une sous-classe ContractEmployee de Employee qui avait un attribut hourlyRate supplémentaire que nous voulions récupérer, nous pourrions alors utiliser le graphique d'entité présenté dans l'extrait 11-18.

Annonce 11-18. Graphique d'entité nommée avec héritage de racine

```
@Entité
@NamedEntityGraph (nom = "Employee.graph4",
    attributeNodes = {
        @NamedAttributeNode ("nom"),
        @NamedAttributeNode ("adresse"),
        @NamedAttributeNode (value = "department", subgraph = "dept")},
    sous-graphes = {
        @NamedSubgraph (nom = "dept",
            attributeNodes = {
                @NamedAttributeNode ("nom")})},
    subclassSubgraphs = {
        @NamedSubgraph (nom = "notUsed", type = ContractEmployee.class,
            attributeNodes = {
                @NamedAttributeNode ("hourlyRate")})
```

```
})
Employé de classe publique {...}
```

Astuce, un problème mineur dans la définition d'annotation est que la sous-classe l'élément est de type NamedSubgraph [], ce qui signifie qu'un nom doit être spécifié même si dans ce cas il n'est utilisé nulle part. Nous l'avons étiqueté non Utilisé pour montrer qu'il est étranger.

Carte des sous-graphiques clés

Le dernier cas particulier est réservé à notre vieil ami, la Carte. Quand une relation L'attribut est de type Map, il y a le problème de la partie clé supplémentaire de la Map. Si la key est un type d'entité ou incorporable, un sous-graphe supplémentaire peut être nécessaire

Épisode 528

spécifié (sinon la règle d'extraction de graphe par défaut s'appliquera). Pour gérer ces cas là est un élément keySubgraph dans NamedAttributeNode. Pour illustrer avoir une carte avec un sous-graphe de clé intégrable, nous utilisons une classe intégrable EmployeeName similaire à Référencement 5-13 (Chapitre 5) et modifiez légèrement l'entité de notre département pour qu'elle soit similaire à Référencement 5-14. Nous listons les définitions de type dans le Listing 11-19 et ajoutons un graphe d'entité nommé.

Annonce 11-19. Graphe d'entité nommée avec sous-graphe de clé de carte

```
@Embeddable
public class EmployeeName {
    private String firstName;
    private String lastName;
    // ...
}

@Entité
@NamedEntityGraph (nom = "Department.graph1",
    attributeNodes = {
        @NamedAttributeNode ("nom"),
        @NamedAttributeNode (valeur = "employés",
            subgraph = "emp",
            keySubgraph = "empName")},
    sous-graphes = {
        @NamedSubgraph (nom = "emp",
            attributeNodes = {
                @NamedAttributeNode (valeur = "nom",
                    subgraph = "empName"),
                @NamedAttributeNode ("salaire")}),
        @NamedSubgraph (nom = "empName",
            attributeNodes = {
                @NamedAttributeNode ("firstName"),
                @NamedAttributeNode ("lastName")})
    })
Département de classe publique {
    @Id id int privé;
    @Embedded
```

Épisode 529


```
    nom privé EmployeeName;
    @OneToMany (mappedBy = "département")
    @MapKey (nom = "nom")
    Carte privée <EmployeeName, Employee> employés;
    // ...
}
```

À ce stade, vous avez à peu près autant d'expertise sur les graphes d'entités nommées que presque n'importe quel développeur, donc après avoir vu tous les exemples d'annotations que vous pourriez avoir remarqué qu'ils étaient plus compliqués qu'il ne le fallait. Dans de nombreux cas, ils a répertorié les attributs qui auraient pu être facilement définis par défaut en utilisant le graphique d'extraction par défaut règle. C'était pour essayer de garder le modèle aussi simple que possible tout en restant correct et capable pour démontrer les concepts. Maintenant que vous avez les règles dans votre esprit, vous devraient revenir en arrière sur chacun des graphiques d'entités nommés et voir comment ils pourrait être raccourci à l'aide de la règle d'extraction de graphique par défaut.

API Entity Graph

L'API est utile pour créer, modifier et ajouter dynamiquement des graphiques d'entités dans le code. Les graphiques d'entités peuvent être utilisés pour générer des plans de récupération basés sur les paramètres du programme, entrée de l'utilisateur, ou dans certains cas, même des données statiques lorsque la création par programmation est préférée. Dans cette section, nous décrivons les classes et la plupart des méthodes de l'API. Nous les appliquons dans des exemples montrant comment créer des équivalents dynamiques des graphiques d'entités nommées dans la section d'annotation précédente.

Alors que les graphiques d'entités résultant des annotations sont les mêmes que ceux créés en utilisant l'API, il existe quelques différences mineures entre les modèles qu'ils utilisent chacun. Ceci est principalement dû aux différences inhérentes entre les annotations et un code API, mais c'est aussi un peu par choix de style.

La façon de commencer à créer un nouveau graphique d'entité est d'utiliser le createEntityGraph () méthode d'usine sur EntityManager. Il prend la classe d'entité racine comme paramètre et renvoie une nouvelle instance EntityGraph typée dans la classe d'entité:

```
EntityGraph <Address> graph = em.createEntityGraph (Address.class);
```

L'étape suivante consiste à ajouter des nœuds d'attribut au graphique d'entité. Les méthodes d'ajout sont conçu pour effectuer la plupart des travaux de création des structures de nœuds pour vous. On peut utiliser

Épisode 530

la méthode variable-arg addAttributeNodes () pour ajouter les attributs qui n'auront pas sous-graphes qui leur sont associés:

```
graph.addAttributeNodes ("rue", "ville", "état", "zip");
```

Cela créera un objet AttributeNode pour chacun des paramètres d'attribut nommés et ajoutez-le au graphe d'entité. Il n'existe malheureusement pas de méthode équivalente à la Élément includeAllAttributes dans l'annotation @NamedEntityGraph.

Il existe également des équivalents de méthode fortement typés à ceux qui prennent noms d'attributs. Les versions tapées utilisent le métamodèle, vous devez donc vous assurer

que le métamodèle a été généré pour votre modèle de domaine (voir le chapitre 9).
Un exemple d'invocation de la méthode addAttributeNodes () fortement typée serait:

```
graph.addAttributeNodes (Address_.street, Address_.city,
                         Address_.state, Address_.zip);
```

Lors de l'ajout d'un attribut pour lequel vous souhaitez également ajouter un sous-graphe, le
Les méthodes addAttributeNodes () ne doivent pas être utilisées. Au lieu de cela, il existe un certain nombre de
Les variantes de méthode addSubgraph () qui devraient être utilisées à la place. Chacun des addSubgraph ()
méthodes créera d'abord une instance d'AttributeNode pour l'attribut transmis, puis
créer une instance de Subgraph, puis lier le sous-graphe au nœud d'attribut, et enfin
renvoie l'instance Subgraph. La version basée sur des chaînes peut être utilisée pour répliquer notre nom
graphique d'entité de la liste 11-15 . Le graphique d'entité résultant est affiché dans la liste11-20 .

Liste 11-20. Graphique d'entité dynamique avec sous-graphiques

```
EntityGraph <Employee> graph = em.createEntityGraph (Employee.class);
graph.addAttributeNodes ("nom", "salaire", "adresse");
Subgraph <Phone> phone = graph.addSubgraph ("téléphones");
phone.addAttributeNodes ("nombre", "type");
Subgraph <Department> dept = graph.addSubgraph ("department");
dept.addAttributeNodes ("nom");
```

Le graphe d'entités basé sur l'API est clairement plus court et plus net que l'annotation-
basé un. C'est l'un de ces cas où les méthodes ne sont pas seulement plus expressives que
annotations mais aussi plus faciles à lire. Bien sûr, les méthodes d'argument variable ne font pas de mal
Soit.

Épisode 531

L'exemple du Listing 11-16 illustre un graphique d'entité contenant un second
définition de la classe d'entité racine, ainsi que le fait qu'un sous-graphe se réfère à un autre.
Référencement 11-21 montre que l'API souffre réellement dans ce cas car elle ne permet pas
partage d'un sous-graphe dans le même graphe d'entité. Puisqu'il n'y a pas d'API à transmettre
une instance Subgraph existante, nous devons construire deux employés nommés identiques
sous-graphiques.

Annonce 11-21. Graphique d'entité dynamique avec plusieurs définitions de type

```
EntityGraph <Employee> graph = em.createEntityGraph (Employee.class);
graph.addAttributeNodes ("nom", "salaire", "adresse");
Subgraph <Phone> phone = graph.addSubgraph ("téléphones");
phone.addAttributeNodes ("nombre", "type");

Sous-graphique <Employee> namedEmp = phone.addSubgraph ("employee");
namedEmp.addAttributeNodes ("nom");

Subgraph <Department> dept = graph.addSubgraph ("department");
dept.addAttributeNodes ("nom");

Sous-graphe <Employé> mgrNamedEmp = graph.addSubgraph ("manager");
mgrNamedEmp.addAttributeNodes ("nom");
```

L'exemple d'héritage dans la liste 11-17 peut être traduit en un
version. Lorsqu'une classe associée est en fait une hiérarchie de classes, chaque appel à addSubgraph () peut
prendre la classe comme paramètre pour distinguer les différentes sous-classes, comme indiqué dans
Référencement 11-22 .

Annonce 11-22. Graphique d'entité dynamique avec héritage

```
EntityGraph <Employee> graph = em.createEntityGraph (Employee.class);
graph.addAttributeNodes ("nom", "salaire", "adresse");
Subgraph <Project> project = graph.addSubgraph ("projets", Project.class);
project.addAttributeNodes ("nom");
Sous-graphe <QualityProject> qaProject = graph.addSubgraph ("projets",
QualityProject.class);
qaProject.addAttributeNodes ("qaRating");
```


Épisode 532

Lorsque l'héritage existe au niveau de la classe d'entité racine, la méthode addSubclassSubgraph () Devrait être utilisé. La classe est le seul paramètre requis. La version API du l'annotation dans l'extrait 11-18 est indiqué dans l'extrait 11-23 .

Annonce 11-23. Graphique d'entité dynamique avec héritage de racine

```
EntityGraph <Employee> graph = em.createEntityGraph (Employee.class);
graph.addAttributeNodes ("nom", "adresse");
graph.addSubgraph ("département"). addAttributeNodes ("nom");
graph.addSubclassSubgraph (ContractEmployee.class) .addAttributeNodes ("horaire
Taux");
```

Notez que dans la liste 11-23 nous utilisons le fait qu'aucun autre sous-graphe n'est ajouté aux sous-graphes connectés au nœud du graphe d'entité, donc aucun des sous-graphes créés sont enregistrés dans des variables de pile. Au contraire, la méthode addAttributeNodes () est appelée directement sur chacun des résultats Subgraph de addSubgraph () et addSubclassSubgraph () méthodes.

Notre dernier exemple à convertir est l'exemple de carte du Listing 11-19 . L'API l'équivalent est montré dans la liste 11-24. L'entité racine est l'entité Department et la carte est codé par le Embeddable EmployeeName.

Annonce 11-24. Graphique d'entité dynamique avec sous-graphique de clé de carte

```
EntityGraph <Department> graph = em.createEntityGraph (Department.class);
graph.addAttributeNodes ("nom");
graph.addSubgraph ("employés"). addAttributeNodes ("salaire");
graph.addKeySubgraph ("employés"). addAttributeNodes ("firstName", "lastName");
```

Dans cet exemple, la méthode addKeySubgraph () a été appelée sur le graphe d'entité racine node mais la même méthode existe également sur Subgraph, donc un sous-graphe clé peut être ajouté à tout niveau où une carte se produit.

Gestion des graphiques d'entité

Les sections précédentes vous ont appris à créer des graphiques d'entités nommées et dynamiques. la prochaine étape logique consiste à voir comment ils peuvent être gérés. Aux fins de cette section, gérer les graphiques d'entités signifie y accéder, les enregistrer, les modifier et en créer de nouveaux en utilisant un existant comme point de départ.

Épisode 533

Accès aux graphiques d'entités nommées

Il est facile d'accéder à un graphique d'entités dynamiques puisque vous l'avez stocké dans la même variable que vous avez utilisé lors du processus de création du graphe d'entité. Lorsque vous avez défini un graphe d'entité nommé, cependant, vous devez y accéder via le gestionnaire d'entités avant ça peut être utilisé. Ceci est réalisé en passant le nom du graphe d'entité au méthode getEntityGraph (). Il sera retourné en tant qu'objet EntityGraph. Nous pouvons accéder le graphe d'entité que nous avons défini dans l'extrait 11-16 avec la déclaration suivante:

```
EntityGraph <?> EmpGraph2 = em.getEntityGraph ("Employee.graph2");
```

Notez que le paramètre type est joker car le type du graphique d'entité n'est pas connu du gestionnaire d'entité. Plus tard, lorsque nous montrerons des façons d'utiliser un graphique d'entité, vous voyez qu'il n'est pas nécessaire de le taper fortement pour l'utiliser.

S'il y a plusieurs graphes d'entités définis sur une seule classe et que nous avons une raison de séquence à travers eux, nous pouvons le faire en utilisant la méthode d'accesseur basée sur les classes. le le code suivant examine les noms d'attribut des classes d'entité racine pour chacun des Graphiques d'entité des employés:

```
Liste <EntityGraph <? super employé >> egList =
                                                em.getEntityGraphs (Employee.class);
for (EntityGraph <? super Employee> graphe: egList) {
    System.out.println ("EntityGraph:" + graph.getName ());
    List <AttributeNode <? >> attribs = graph.getAttributeNodes ();
    pour (AttributeNode <?> attr: attribs) {
        System.out.println ("Attribut:" + attr.getAttributeName ());
    }
}
```

Dans ce cas, le type de paramètre EntityGraph a une limite inférieure pour être Employee, mais peut également être une superclasse d'employés. Si, par exemple, Person était une superclasse entity of Employee, nous obtiendrions également les graphiques d'entité Person inclus dans la sortie.

Ajout de graphiques d'entités nommées

L'API Entity Graph permet la création dynamique de graphiques d'entités, mais vous pouvez aller encore plus loin en prenant ces graphiques d'entité et en les enregistrant sous forme de graphes d'entités nommés. Une fois qu'ils sont nommés ils peuvent être utilisés comme un graphe d'entité nommé qui a été défini statiquement dans une annotation.

Épisode 534

Nous pouvons ajouter tous les graphiques d'entités que nous avons créés dans les listes 11-20 à11-24 en tant que graphe d'entité nommée à l'aide de la méthode addNamedEntityGraph () sur le gestionnaire d'entités usine:

```
em.getEntityManagerFactory (). addNamedEntityGraph ("Employee.graphX", graphique);
```

Notez que le nom que nous choisissons pour le graphe d'entités nous appartient, tout comme il l'était lorsque nous l'a défini sous forme d'annotation, sauf que dans ce cas, il n'y a pas de nom par défaut. Nous devons fournir un nom comme paramètre.

Si un graphe d'entité nommée portant le même nom existait déjà dans le l'espace de noms du graphe d'entité, il sera remplacé par celui que nous fournissons dans le addNamedEntityGraph (), car il ne peut y avoir qu'un seul graphique d'entité d'un nom donné dans une unité de persistance.

Création de nouveaux graphiques d'entités à partir de ceux nommés

Dans certains cas, vous constaterez peut-être que vous avez un graphique d'entité existant mais que vous souhaitez créer un nouveau qui est très similaire à celui existant, mais diffère par un petit facteur. Ceci peut être particulièrement vrai en raison du fait que les sous-graphes ne peuvent pas être partagés entre les graphes d'entités. La meilleure façon de faire est d'utiliser la méthode createEntityGraph (). En utilisant un existant graphique, vous pouvez modifier uniquement les parties que vous souhaitez modifier, puis réenregistrer la modification un sous un nom différent.

Dans la liste 11-16, nous avons défini un graphique d'entité pour les employés incluant leurs téléphones et leur département, mais pas leurs projets. Dans la liste11-25 , nous accédons à cette entité graphique et modifiez-le pour inclure également leurs projets. Ajout d'un nœud d'attribut qui est un relation, mais ne pas ajouter de sous-graphe pour cela, provoquera le graphe d'extraction par défaut pour cela type d'entité à appliquer. Nous pouvons alors choisir de sauvegarder le graphe d'entité sous le même nom, écrasant effectivement le précédent, ou enregistrez-le sous un nom différent afin que nous avons le choix entre les deux graphiques d'entités. Référencement11-25 fait ce dernier.

Annonce 11-25. Création d'un graphique d'entité à partir d'un graphique existant

```
EntityGraph <?> Graph = em.createEntityGraph ("Employee.graph2");
graph.addAttributeNodes ("projets");
em.getEntityManagerFactory (). addNamedEntityGraph ("Employee.newGraph",
graphique);
```


Épisode 535

La modification que nous avons apportée au graphique d'entité dans la liste 11-25 était en fait assez trivial à dessein car il s'avère que vous pouvez être quelque peu limité dans ce que vous peut faire quand il s'agit d'apporter des modifications à un graphique d'entité existant. La raison est que le Javadoc de l'API Entity Graph ne spécifie pas si vous pouvez muter le accesseurs de collection. Par exemple, EntityGraph a une méthode getAttributeNodes () mais la méthode ne spécifie pas si le List <AttributeNode <? >> renvoyé par elle est le Liste réelle référencée par l'instance EntityGraph. S'il s'agit d'une copie, alors s'il est modifié, il n'aura aucun effet sur l'instance EntityGraph à partir de laquelle il a été obtenu. Cela ferait il est impossible de supprimer un attribut du graphique car il n'y a pas d'API alternative pour modifier les collections autrement que pour y ajouter.

Utilisation des graphiques d'entité

La partie la plus difficile des graphiques d'entités est de les créer pour être corrects et produire le résultat que vous attendez. Une fois que vous avez créé les bons graphiques d'entités, les utiliser est assez Facile. Ils sont transmis comme valeurs de l'une des deux propriétés standard. Ces propriétés peut être soit passé dans une méthode find (), soit défini comme indicateur de requête sur n'importe quel nom ou requête dynamique. En fonction de la propriété utilisée, le graphe d'entité prendra le rôle d'un graphe d'extraction ou d'un graphe de charge. Les sections suivantes expliquent la sémantique des deux types de graphiques et montrez quelques exemples de comment et quand ils peuvent être utilisés.

Récupérer des graphiques

Lorsqu'un graphe d'entité est passé dans une méthode find () ou une requête en tant que valeur du javax.persistence.fetchgraph, le graphe d'entité sera traité comme une extraction graphique. La sémantique d'un graphe d'extraction est que tous les attributs inclus dans le graphe sont à traiter comme ayant un type d'extraction de EAGER, comme si le mappage avait été spécifié comme fetch = FetchType.EAGER, indépendamment de ce que le mappage statique spécifie réellement. Les attributs qui ne sont pas inclus dans le graphique doivent être traités comme LAZY. Comme décrit plus tôt, tous les attributs d'identifiant ou de version seront traités comme EAGER et chargés, indépendamment

si elles sont incluses dans le graphique. Aussi, comme nous l'avons expliqué, si une relation ou l'attribut incorporé est inclus dans le graphe mais aucun sous-graphe n'est spécifié pour lui, alors le Le graphique d'extraction par défaut pour ce type sera utilisé.

L'utilité d'un graphe de récupération est principalement de permettre une extraction paresseuse des attributs quand ils ont été configurés ou définis par défaut pour être récupérés avec impatience dans le mappage. La chose à retenir est que la même sémantique de LAZY s'applique ici que celles qui ont été décrites

Épisode 536

au chapitre 4 . Autrement dit, lorsqu'un attribut est marqué comme LAZY, il n'y a aucune garantie que le L'attribut restera déchargé jusqu'au premier accès. Si un attribut est LAZY, il signifie uniquement que le fournisseur peut optimiser en ne récupérant pas l'état de l'attribut avant on y accède. Le fournisseur a toujours le droit de charger les attributs LAZY avec empressement s'il le souhaite.

Regardons un exemple d'utilisation d'un graphique d'extraction. Dans la liste11-16 , nous avons défini une entité graphique pour l'entité Employé. Puisqu'il a été défini comme un graphe d'entité nommé, nous pouvons accédez-y en utilisant la méthode getEntityGraph () et utilisez-la comme valeur pour notre javax. Propriété persistence.fetchgraph.

```
Map <String, Object> props = new HashMap <String, Object> ();
props.put ("javax.persistence.fetchgraph",
        em.getEntityGraph ("Employee.graph2"));
Employé emp = em.find (Employee.class, empId, accessoires);
```

Nous pouvons tout aussi facilement passer dans la version dynamique que nous avons créée dans Listing 11-21 . Si le graphique dynamique était en cours de création dans la variable graphique, alors nous pourrions simplement le passer dans la méthode find (), ou comme indice de requête:

```
EntityGraph <Employee> graph = em.createEntityGraph (Employee.class);
// ... (composer le graphe comme dans l'extrait 11-21)
TypedQuery <Employee> query = em.createQuery (
        «SELECT e FROM Employee e WHERE e.salary> 50000», Employee.class);
query.setHint ("javax.persistence.fetchgraph", graphique);
List <Employee> results = query.getResultList ();
```

Graphiques de charge

Un graphe de charge est un graphe d'entité fourni en tant que valeur à javax.persistence. propriété loadgraph. La principale différence entre un graphique d'extraction et un graphique de charge est de savoir comment les attributs manquants sont traités. Dans un graphique d'extraction, un attribut exclu doit être traités comme LAZY, dans un graphique de charge, tous les attributs manquants doivent être traités tels qu'ils sont définis dans les mappages. L'exprimer en termes du graphique d'extraction par défaut dont nous avons discuté dans les sections précédentes, un graphique de charge vide sans attribut inclus est le même que le graphique d'extraction par défaut pour ce type. L'intérêt d'utiliser un graphique de charge réside dans la capacité à provoquer un ou plusieurs attributs à traiter comme EAGER même s'ils étaient définis statiquement être fainéant.

Épisode 537

Au chapitre 6 , nous avons montré quelques exemples de moyens de garantir que le service des employés a été chargé, même s'il a été spécifié comme LAZY. C'est un cas parfait pour utiliser une charge graphique. Dans la liste 11-26, nous montrons une version alternative du Listing 6-27 (Chapitre 6).

Annonce 11-26. Déclencher une relation paresseuse

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    privé EntityManager em;

    Liste publique findAll () {
        EntityGraph <Employee> graph = em.createEntityGraph (Employee.class);
        graph.addAttributeNodes ("département");
        TypedQuery <Employee> query = em.createQuery (
                «SELECT e FROM Employee e», Employee.class);
        query.setHint ("javax.persistence.loadgraph", graphique);
        return query.getResultList ();
    }
    // ...
}
```

Comme vous pouvez le voir, la création de graphe est assez simple puisque nous n'en ajoutons qu'un attribut et pas de sous-graphiques. L'attribut département est une relation, et nous n'avons pas inclure un sous-graphe pour cela afin que le graphe d'extraction par défaut pour Department soit utilisé.

LE CAS DE LA PROPRIÉTÉ MANQUANTE

la difficulté d'utiliser un graphe d'extraction est qu'il s'agit d'une spécification complète du type. à rendre un seul attribut paresseux, vous devez spécifier tous les attributs désireux. en d'autres termes, vous ne pouvez pas simplement remplacer le mode de récupération pour un seul attribut; lorsque vous créez un graphique de récupération pour une entité ou un type intégrable, vous écrasez effectivement tous les attributs du type, soit en les incluant, soit en les excluant du graphique de récupération. Ce dont la spécification a vraiment besoin, c'est une propriété supplémentaire, javax.persistence.lazygraph, qui spécifierait que tous les les attributs inclus dans le graphique doivent être paresseux et tous les attributs exclus reviennent à ce que ils sont définis comme étant dans leurs mappages. cela permettrait l'inclusion sélective d'attributs paresseux.

Meilleures pratiques pour l'extraction et le chargement de graphiques

Quand et comment utiliser un graphique de récupération ou un graphique de charge deviendra évident lorsque vous essayez de Utilise l'un ou l'autre; il ne faudra pas longtemps pour découvrir lequel correspond à votre chargement d'attribut Besoins. Cependant, nous vous proposons quelques conseils pour vous lancer et peut-être vous faire économiser un peu temps au départ.

Découvrez ce que votre fournisseur cible soutient en termes de paresse. Si votre fournisseur charge chaque attribut paresseux avec impatience et vous prévoyez de créer une série d'extraction graphiques pour que vos requêtes rendent les attributs paresseux, alors cela ne vaut peut-être pas la peine d'y aller à travers l'effort du tout. Vous pouvez tester le comportement de chargement de votre fournisseur en utilisant le PersistenceUnitUtil.isLoaded () sur un attribut paresseux avant d'y accéder. Essayer sur les différents types d'attributs que vous prévoyez de définir sur LAZY.

Si vos graphiques d'entités n'agissent pas comme vous le pensez, recherchez attributs qui n'ont pas de sous-graphes. N'oubliez pas que le graphique d'extraction par défaut sera utilisé lorsqu'aucun sous-graphe n'est spécifié. Ceci est pertinent pour toutes les relations bidirectionnelles (rappelez-vous pour redéfinir un sous-graphe pour la relation avec le type d'objet d'origine), mais surtout ceux qui reviennent au type d'entité racine. Votre intuition peut vous dire que l'entité racine

la spécification de graphe sera utilisée, mais c'est le graphe d'extraction par défaut qui sera utilisé à la place.

Si vous finissez par utiliser un graphique d'extraction ou de chargement pour modifier le type d'extraction d'un attribut plus souvent qu'autrement, vous voudrez peut-être envisager de changer la façon dont le type de récupération est défini dans la cartographie. Le mappage doit définir le mode de récupération le plus couramment utilisé, avec le extraire ou charger des graphiques en le remplaçant dans les cas exceptionnels.

L'utilisation de graphes d'entités nommées permet la réutilisation du graphe d'entités et constitue un façon de créer un graphique légèrement différent pour des modifications ponctuelles. Tout en les déclarant dans le code est un moyen plus soigné et préféré pour définir les graphes d'entités, vous devriez continuer à les enregistrer en tant que graphes d'entités nommées pour atteindre la réutilisabilité. Vous voudrez également les définir dans le code dont vous êtes sûr qu'elle sera exécutée avant que les requêtes qui les utilisent soient exécutées.

Résumé

Nous avons commencé le chapitre par un regard sur les requêtes SQL. Nous avons examiné le rôle de SQL dans applications qui utilisent également JP QL et les situations spécialisées où seul SQL peut être utilisé. Pour combler le fossé entre le SQL natif et les entités, nous avons décrit le jeu de résultats processus de cartographie en détail, montrant un large éventail de requêtes et comment elles se traduisent dans le modèle de domaine d'application.

Épisode 539

Nous avons ensuite montré comment les procédures stockées peuvent être appelées via une requête JPA et comment les résultats peuvent être obtenus via les paramètres de sortie, les curseurs de référence et les ensembles de résultats. Les procédures stockées seront toujours quelque peu spécifiques à la base de données, il faut donc faire attention prises lors de leur utilisation.

Nous avons terminé en parlant des graphiques d'entités, de ce qu'ils sont et de la façon dont ils sont construit. Nous avons montré comment les graphes d'entités nommées peuvent être définis sous forme d'annotations et a poursuivi en décrivant l'API pour créer des graphiques d'entités dynamiques dans le code. L'entité les méthodes de gestion pour obtenir des graphiques d'entités nommées ou modifiables ont été discutées, y compris un exemple qui a pris un graphe d'entité nommé existant, y a apporté une modification, puis ajouté le graphique modifié en tant que graphique d'entité nommé distinct. Enfin, nous avons montré comment des graphiques d'entités sont utilisés. Vous avez vu comment ils prennent la sémantique de l'extraction de graphes ou du chargement graphes lorsqu'ils sont passés en tant que valeurs de propriété aux méthodes ou requêtes find ().

Dans le chapitre suivant, nous aborderons des sujets plus avancés, principalement dans les domaines de rappels de cycle de vie, écouteurs d'entité, validation, accès concurrentiel, verrouillage et mise en cache.

Épisode 540

CHAPITRE 12

Autres sujets avancés

Lorsqu'un titre de chapitre comprend l'expression «sujets avancés», il y a toujours le risque que le contenu peut ne pas correspondre à ce que chaque lecteur considère comme avancé. Le terme «Avancé» est au mieux subjectif et dépend du contexte et de l'expérience du développeur ainsi que la complexité de l'application en cours de développement.

Ce que nous pouvons dire, c'est que, dans une large mesure, les sujets de ce chapitre sont ceux qui destiné (lors de l'élaboration de la spécification) à être de nature plus avancée ou à utiliser par des développeurs plus avancés. Il y a quelques exceptions à cette règle, bien que. Par exemple, nous avons inclus le verrouillage optimiste dans ce chapitre même si la plupart des applications doivent connaître et utiliser le verrouillage optimiste. Cependant, le les appels de verrouillage réels sont rarement utilisés, et il était juste logique de couvrir tous les modes de verrouillage ensemble. En général, nous pensons que la plupart des applications n'utiliseront pas plus de quelques les fonctionnalités décrites dans ce chapitre. Dans cet esprit, explorons certains des autres fonctionnalités de l'API Java Persistence.

Rappels de cycle de vie

Chaque entité a le potentiel de passer par un ou plusieurs cycles de vie définis événements. Selon les opérations invoquées sur une entité, ces événements peuvent ou peuvent ne se produisent pas pour cette entité, mais il y a au moins un potentiel pour qu'elles se produisent. Afin de répondre à l'un ou plusieurs des événements, une classe d'entité ou l'une de ses superclasses peut déclarer une ou plusieurs méthodes qui seront invoquées par le fournisseur lorsque l'événement sera mis à la porte. Ces méthodes sont appelées *méthodes de rappel.*

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_12

Épisode 541

Événements du cycle de vie

Les types d'événements qui composent le cycle de vie se répartissent en quatre catégories: persistance, mise à jour, suppression et chargement. Ce sont vraiment des événements au niveau des données qui correspondent à

opérations d'insertion, de mise à jour, de suppression et de lecture de bases de données; et sauf pour le chargement, chacun a un événement pré et un événement post. Dans la catégorie de chargement, il n'y a qu'un PostLoad événement car cela n'aurait aucun sens qu'il y ait PreLoad sur une entité qui n'était pas encore construit. Ainsi, la suite complète des événements du cycle de vie qui peuvent survenir est composée de PrePersist, PostPersist, PreUpdate, PostUpdate, PreRemove, PostRemove et PostLoad.

PrePersist et PostPersist

L'événement PrePersist notifie une entité lorsque EntityManager.persist () a été invoqué avec succès dessus. Les événements PrePersist peuvent également se produire lors d'un appel à merge () lorsqu'un la nouvelle entité a été fusionnée dans le contexte de persistance. Si l'option en cascade PERSIST est défini sur une relation d'un objet qui est persistant et l'objet cible est également un nouvel objet, l'événement PrePersist est déclenché sur l'objet cible. Si plusieurs entités sont en cascade pendant la même opération, l'ordre dans lequel les rappels PrePersist ne peut pas être invoqué.

Les événements PostPersist se produisent lorsqu'une entité est insérée, ce qui se produit normalement pendant la phase d'achèvement de la transaction. Le déclenchement d'un événement PostPersist n'indique pas que l'entité s'est engagée avec succès dans la base de données car la transaction dans laquelle elle a persisté peut être annulé après l'événement PostPersist mais avant la transaction s'engage avec succès.

PreRemove et PostRemove

Lorsqu'un appel EntityManager.remove () est appelé sur une entité, le rappel PreRemove est déclenché. Ce rappel implique qu'une entité est mise en file d'attente pour suppression, et tout entités liées à travers les relations qui ont été configurées avec la cascade REMOVE L'option recevra également une notification PreRemove. Lorsque le SQL pour la suppression d'une entité finalement est envoyé à la base de données, l'événement PostRemove sera déclenché. Comme avec le Événement de cycle de vie PostPersist, l'événement PostRemove ne garantit pas le succès. le la transaction englobante peut encore être annulée.

Épisode 542

PreUpdate et PostUpdate

Les mises à jour des entités gérées peuvent survenir à tout moment, que ce soit au cours d'une transaction ou (dans le cas d'un contexte de persistance étendu) en dehors d'une transaction. Parce qu'il y a aucune méthode explicite sur l'EntityManager, le rappel PreUpdate est garanti appelée uniquement à un moment donné avant la mise à jour de la base de données. Certaines implémentations peuvent suivre les modifications de manière dynamique et peut invoquer le rappel à chaque modification, tandis que d'autres peut attendre la fin de la transaction et appeler une seule fois le rappel.

Une autre différence entre les implémentations est de savoir si les événements PreUpdate déclenché sur des entités qui ont été conservées dans une transaction puis modifiées dans le même transaction avant d'être validée. Ce serait un choix plutôt malheureux car à moins que les écritures ne soient effectuées avec empressement à chaque appel d'entité, il n'y aurait pas de symétrie Appel PostUpdate car dans le cas habituel d'écriture différée, un seul persiste base de données se produirait à la fin de la transaction. Le rappel PostUpdate se produit juste après la mise à jour de la base de données. Le même potentiel de restauration existe après PostUpdate callbacks comme avec PostPersist et PostRemove.

PostLoad

Le rappel PostLoad se produit après la lecture des données d'une entité dans la base de données et l'instance d'entité est construite. Cela peut être déclenché par toute opération qui provoque un

entité à charger, normalement par une requête ou une traversée d'une relation paresseuse. Ça peut
se produisent également à la suite d'un appel refresh () sur le gestionnaire d'entités. Quand une relation
est défini sur cascade REFRESH, les entités qui sont mises en cascade seront également chargées. le
l'invocation d'entités en une seule opération, qu'il s'agisse d'une requête ou d'une actualisation, n'est pas garantie
être dans n'importe quel ordre, nous ne devons donc pas nous fier à un ordre observé dans aucune implémentation.

Conseil D'autres méthodes de cycle de vie peuvent être définies par des fournisseurs spécifiques, par exemple lorsque une entité est fusionnée ou copiée / clonée.

Méthodes de rappel

Les méthodes de rappel peuvent être définies de différentes manières, la plus élémentaire étant de
définissez simplement une méthode sur la classe d'entité. Désignation de la méthode comme méthode de rappel
comporte deux étapes: définir la méthode en fonction d'une signature donnée et annoter
la méthode avec l'annotation d'événement de cycle de vie appropriée.

Épisode 543

La définition de signature requise est très simple. La méthode de rappel peut avoir n'importe quel
name, mais doit avoir une signature qui ne prend aucun paramètre et a un type de retour void.
Une méthode telle que public void foo () {} est un exemple de méthode valide. Finale ou statique
Cependant, les méthodes ne sont pas des méthodes de rappel valides.

Les exceptions vérifiées ne peuvent pas être levées à partir des méthodes de rappel car la méthode
La définition d'une méthode de rappel n'est pas autorisée à inclure une clause throws. Durée
des exceptions peuvent être levées, cependant, et si elles sont levées pendant une transaction, elles
amènera le fournisseur non seulement à abandonner l'invocation d'un événement de cycle de vie ultérieur
méthodes dans cette transaction, mais marque également la transaction pour la restauration.

Une méthode est indiquée comme étant une méthode de rappel en étant annotée avec un cycle de vie
annotation d'événement. Les annotations pertinentes correspondent aux noms des événements répertoriés précédemment:
@PrePersist, @PostPersist, @PreUpdate, @PostUpdate, @PreRemove, @PostRemove et
@PostLoad. Une méthode peut être annotée avec plusieurs annotations d'événement de cycle de vie, mais
une seule annotation de cycle de vie d'un type donné peut être présente dans une classe d'entité.

Certains types d'opérations peuvent ne pas être effectués de manière portative dans les méthodes de rappel.
Par exemple, appeler des méthodes sur un gestionnaire d'entités ou exécuter des requêtes obtenues
à partir d'un gestionnaire d'entités ne sont pas pris en charge, ainsi que l'accès à des entités autres que
celui auquel l'événement du cycle de vie s'applique. Recherche de ressources dans JNDI ou à l'aide de JDBC
et les ressources JMS sont autorisées, donc rechercher et appeler des beans session EJB serait
permis.

Maintenant que vous connaissez tous les différents types d'événements de cycle de vie qui peuvent être gérés, allons
regardez un exemple qui les utilise. Une utilisation courante des événements du cycle de vie est de maintenir
état non persistant à l'intérieur d'une entité persistante. Si nous voulons que l'entité enregistre sa mise en cache
l'âge ou l'heure de sa dernière synchronisation avec la base de données, nous pourrions facilement le faire correctement
à l'intérieur de l'entité à l'aide de méthodes de rappel. Notez que l'entité est considérée comme synchronisée
avec la base de données à chaque fois qu'il est lu ou écrit dans la base de données. L'entité est
montré dans le Listing 12-1. Les utilisateurs de cette entité Employé peuvent vérifier l'âge de la mise en cache de cette
objet pour voir s'il répond à leurs exigences de fraîcheur.

Annonce 12-1. Utilisation des méthodes de rappel sur une entité

```
@Entité
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    @Transient private long syncTime;
```

Épisode 544


```
    // ...

    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime () {
        syncTime = System.currentTimeMillis ();
    }

    public long getCachedAge () {
        return System.currentTimeMillis () - syncTime;
    }

    // ...
}
```

Contextes d'entreprise

Lorsqu'une méthode de rappel est appelée, le fournisseur n'entreprendra aucune action particulière pour suspendre ou établir tout autre type de dénomination, de transaction ou de contexte de sécurité dans le Environnement Java EE. Les méthodes de rappel sont exécutées dans tous les contextes actifs à le moment où ils sont invoqués.

Il est important de se souvenir de ce fait car il s'agira souvent d'un haricot avec un récipient-transaction gérée qui appelle des appels sur le gestionnaire d'entités, et ce sera ce bean contextes qui seront en vigueur lorsque les pré-appels sont appelés. Selon l'endroit où le transaction lancée et validée, les appels Post seront probablement appelés à la fin de la transaction et pourrait en fait se trouver dans un ensemble de contextes entièrement différent de celui méthodes. Cela est particulièrement vrai dans le cas d'un contexte de persistance étendue où les entités sont gérées et conservées en dehors d'une transaction, mais la transaction suivante commit entraînera l'écriture des entités persistantes.

Écouteurs d'entités

Les méthodes de rappel dans l'entité conviennent si cela ne vous dérange pas si la logique de rappel d'événement est inclus dans l'entité, mais que faire si vous souhaitez retirer le comportement de gestion des événements de la classe d'entité dans une classe différente? Pour ce faire, vous pouvez utiliser un écouteur d'entité. Un l'auditeur d'entité n'est pas une entité; c'est une classe sur laquelle vous pouvez définir un ou plusieurs cycles de vie

Épisode 545

méthodes de rappel à appeler pour les événements du cycle de vie d'une entité. Semblable à la méthodes de rappel sur l'entité, cependant, une seule méthode dans chaque classe d'écouteur peut être annoté pour chaque type d'événement. Plusieurs écouteurs d'événements peuvent être appliqués à une entité, bien que.

Lorsque le rappel est appelé sur un écouteur, celui-ci doit généralement avoir accès à l'état de l'entité. Par exemple, si nous devions implémenter l'exemple précédent du âge mis en cache d'une instance d'entité, alors nous voudrions faire passer l'instance d'entité. Pour cette raison, la signature requise des méthodes de rappel sur les écouteurs d'entité est légèrement

différent de celui requis sur les entités. Sur un écouteur d'entité, une méthode de rappel doit avoir une signature similaire en tant qu'entité à l'exception du fait qu'elle doit également avoir un seul paramètre défini d'un type compatible avec le type d'entité, en tant que classe d'entité, un superclasse (y compris Object), ou une interface implémentée par l'entité. Une méthode avec la signature public void foo (Object o) {} est un exemple de méthode de rappel valide sur un auditeur d'entité. La méthode doit ensuite être annotée avec l'événement nécessaire annotation (s).

Les classes d'écouteur d'entité doivent être sans état[1] , ce qui signifie qu'ils ne doivent déclarer aucun des champs. Une seule instance peut être partagée entre plusieurs instances d'entité et peut même être invoqué simultanément pour plusieurs instances d'entité. Afin que le fournisseur puisse pouvoir créer des instances de l'écouteur d'entité, chaque classe d'écouteur d'entité doit avoir un constructeur public sans argument.

Entity Listeners en tant que Beans CDI

Si CDI est activé dans le conteneur, les écouteurs d'entité peuvent également être des beans CDI. Si un l'écouteur d'entité est un bean CDI, non seulement il peut être une cible d'injection, mais il a également un cycle de vie de composant et peut inclure le cycle de vie PostConstruct et PreDestroy méthodes discutées au chapitre 3 . Cependant, contrairement à la plupart des autres types de beans CDI, les écouteurs d'entités ne sont pas contextuels, donc les instances ne sont pas stockées dans un contexte spécifique pour injection ultérieure dans d'autres objets.

Remarque Les écouteurs d'entité sont les seuls objets Jpa qui peuvent être des beans Cdi.

[1] Sauf si l'auditeur est également un bean CDI, comme décrit dans la section suivante.

Épisode 546

Clairement, si un écouteur d'entité est un bean CDI, il ne sera pas apatride et ***peut*** ont des champs d'état. Sinon, il serait inutile d'en faire un bean CDI. Mais depuis il n'est pas légal d'utiliser le gestionnaire d'entités à partir d'un rappel de cycle de vie, cela ne servirait à rien en injectant un gestionnaire d'entités. Il serait plus utile d'injecter d'autres types de ressources utilisé par l'écouteur d'entité, tel qu'un bean contextuel CDI, une file d'attente de messages ou autre ressource de conteneur.

Conseil Ce n'est que depuis Jpa 2.1 qu'il est possible de faire en sorte que les Haricots Cdi.

Association d'écouteurs d'entités à des entités

Une entité désigne les écouteurs d'entité qui doivent être notifiés des événements de son cycle de vie grâce à l'utilisation de l'annotation @EntityListeners. Un ou plusieurs écouteurs d'entité peuvent figurer dans l'annotation. Lorsqu'un événement de cycle de vie se produit, le fournisseur effectue une itération via chacun des écouteurs d'entité dans l'ordre dans lequel ils ont été répertoriés et instanciés une instance de la classe d'écouteur d'entité qui a une méthode annotée avec l'annotation pour l'événement donné. Il invoquera la méthode de rappel sur l'écouteur, en passant le entité à laquelle l'événement s'applique. Après avoir fait cela pour tous les écouteurs d'entités répertoriés, il invoquera la méthode de rappel sur l'entité s'il y en a une. Si l'un des auditeurs jette une exception, il abandonnera le processus de rappel, provoquant les écouteurs restants et le méthode de rappel sur l'entité à ne pas appeler.

Regardons maintenant l'exemple d'âge de l'entité en cache et ajoutons des écouteurs d'entité dans

le mélange. Parce que nous avons maintenant la possibilité d'effectuer plusieurs tâches dans plusieurs auditeurs, nous peut ajouter un auditeur pour effectuer une validation de nom ainsi que des actions supplémentaires sur l'employé enregistrer les modifications. Référencement12-2 montre l'entité avec ses écouteurs ajoutés.

Liste 12-2. Utilisation de plusieurs écouteurs d'entités

```
@Entité
@EntityListeners ({EmployeeDebugListener.class, NameValidator.class})
Public class Employee implémente NamedEntity {
    @Id id int privé;
    nom de chaîne privé;
    @Transient private long syncTime;
```


Épisode 547


```
    public String getName () {nom de retour; }

    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime () {
        syncTime = System.currentTimeMillis ();
    }

    public long getCachedAge () {
        return System.currentTimeMillis () - syncTime;
    }

    // ...
}

interface publique NamedEntity {
    public String getName ();
}

Public class NameValidator {
    statique final int MAX_NAME_LEN = 40;

    @PrePersist
    public void validate (NamedEntity obj) {
        if (obj.getName (). length ())> MAX_NAME_LEN)
            throw new ValidationException ("Identifier out of range");
    }
}

public class EmployeeDebugListener {
    @PrePersist
    public void prePersist (Employé emp) {
        System.out.println ("Persister sur l'identifiant d'employé:" + emp.getId ());
    }

    @PreUpdate
    public void preUpdate (Employé emp) {...}
```

Épisode 548


```
    @PreRemove
    public void preRemove (Employé emp) {...}

    @PostLoad
    public void postLoad (Employé emp) {...}
}
```

Comme vous pouvez le voir, différentes méthodes de rappel d'écouteur prennent différents types de paramètres. Les méthodes de rappel de la classe EmployeeDebugListener prennent Employee comme paramètre car ils ne sont appliqués qu'aux entités Employee. Dans la classe NameValidator, le paramètre de méthode validate () est de type NamedEntity. L'entité Employé et tout nombre d'autres entités qui ont des noms peuvent implémenter cette interface. La validation la logique peut être nécessaire car un aspect particulier du système peut avoir un nom courant. limitation de longueur mais peut changer dans le futur. Il est préférable de centraliser cette logique dans une seule classe que de dupliquer la logique de validation dans chacune des méthodes de définition de classe si il existe une possibilité de hiérarchie d'héritage.

Même si les écouteurs d'entité sont pratiques, nous avons décidé de quitter l'âge du cache logique dans l'entité parce qu'elle modifie en fait l'état de l'entité et parce que le mettre dans une classe séparée nous aurait obligé à assouplir l'accès du privé resetSyncTime (). En général, lorsqu'une méthode de rappel accède à l'état au-delà ce qui devrait être accessible au public, il est le mieux adapté pour être dans l'entité et non dans un écouteur d'entité.

Écouteurs d'entités par défaut

Un auditeur peut être attaché à plus d'un type d'entité simplement en étant répertorié dans le @EntityListeners annotation de plus d'une entité. Cela peut être utile dans les cas où l'écouteur fournit une fonctionnalité plus générale ou une logique d'exécution étendue.

Pour une utilisation encore plus large d'un écouteur d'entité sur toutes les entités d'une persistance unit, un ou plusieurs écouteurs d'entité par défaut peuvent être déclarés. Il n'y a actuellement aucune norme cible d'annotation pour les métadonnées étendues à l'unité de persistance, ce type de métadonnées peut donc être déclaré uniquement dans un fichier de mappage XML. Voir le chapitre13 pour les détails sur la façon de déclarer écouteurs d'entités par défaut.

Lorsqu'une liste d'écouteurs d'entités par défaut est déclarée, elle sera parcourue dans l'ordre ont été listés dans la déclaration, et chacun qui a une méthode annotée ou déclarée pour le l'événement en cours sera invoqué. Les écouteurs d'entités par défaut seront toujours appelés avant l'un des écouteurs d'entité répertoriés dans l'annotation @EntityListeners pour une entité donnée.

Épisode 549

Toute entité peut choisir de ne pas avoir les écouteurs d'entité par défaut qui lui sont appliqués en utilisant l'annotation @ExcludeDefaultListeners. Lorsqu'une entité est annotée avec ceci annotation, aucun des écouteurs par défaut déclarés ne sera appelé pour les événements du cycle de vie pour les instances de ce type d'entité.

Événements d'héritage et de cycle de vie

La présence d'événements avec des hiérarchies de classes nécessite que nous explorions le sujet de les événements du cycle de vie un peu plus en profondeur. Que se passe-t-il lorsque nous avons plusieurs entités chacun définit des méthodes de rappel ou des écouteurs d'entité ou les deux? Est-ce qu'ils sont tous invoqués sur un entité de sous-classe ou uniquement celles qui sont définies sur ou dans l'entité de sous-classe?

Ces questions et bien d'autres se posent en raison de la complexité supplémentaire de

hiérarchies d'héritage. Il s'ensuit qu'il doit y avoir des règles pour définir les
comportement face à des hiérarchies potentiellement complexes où les méthodes d'événement de cycle de vie sont
dispersés dans toute la hiérarchie.

Hériter des méthodes de rappel

Les méthodes de rappel peuvent se produire sur n'importe quelle entité ou superclasse mappée, qu'elle soit abstraite ou
béton. La règle est assez simple. C'est que chaque méthode de rappel pour un événement donné
type sera invoqué dans l'ordre en fonction de sa place dans la hiérarchie, le plus général
classes d'abord. Ainsi, si dans la hiérarchie des employés qui était dans la figure 10-10 (dans le chapitredix)
la classe Employee contient une méthode de rappel PrePersist nommée checkName (),
et FullTimeEmployee contient également une méthode de rappel PrePersist nommée
verifyPension (), lorsque l'événement PrePersist se produit, la méthode checkName () obtiendra
invoqué, suivi de la méthode verifyPension ().

Nous pourrions également avoir une méthode sur la superclasse mappée CompanyEmployee que nous
veulent s'appliquer à toutes les entités qui l'ont sous-classé. Si nous ajoutons une méthode PrePersist nommée
checkVacation () qui vérifie que le report de vacances est inférieur à un certain montant, il
sera exécuté après checkName () et avant verifyPension ().

Cela devient plus intéressant si nous définissons une méthode checkVacation () sur le
PartTimeEmployee parce que les employés à temps partiel ne reçoivent pas autant
vacances. Annoter la méthode remplacée avec PrePersist entraînerait le
PartTimeEmployee.checkVacation () à invoquer à la place de celle de
Employé.

Épisode 550

Hériter des écouteurs d'entité

Comme les méthodes de rappel dans une entité, l'annotation @EntityListeners est également valide sur
entités ou superclasses mappées dans une hiérarchie, qu'elles soient concrètes ou abstraites.
Également similaires aux méthodes de rappel, les écouteurs répertoriés dans l'annotation de la superclasse d'entité
être invoqué avant les écouteurs dans les entités de la sous-classe. En d'autres termes, définir un
L'annotation @EntityListeners sur une entité est additive en ce qu'elle n'ajoute que des écouteurs; il
ne les redéfinit ni leur ordre d'invocation.

Pour redéfinir les écouteurs d'entités invoqués et leur ordre d'appel, un
L'entité ou la superclasse mappée doit être annotée avec @ExcludeSuperclassListeners.
Cela empêchera les écouteurs définis dans toutes les superclasses d'être invoqués pour aucun des
les événements du cycle de vie de la sous-classe d'entités annotées. Si nous voulons un sous-ensemble d'auditeurs
pour être invoqués, ils doivent être répertoriés dans l'annotation @EntityListeners sur le
l'entité prioritaire et dans l'ordre approprié.

Ordre d'appel des événements du cycle de vie

Les règles d'appel des événements de cycle de vie sont désormais un peu plus complexes, elles justifient
être présenté plus soigneusement. La meilleure façon de le décrire est peut-être de décrire le processus
que le fournisseur doit suivre pour appeler les méthodes d'événement. Si un événement de cycle de vie donné X
se produit pour l'entité A, le fournisseur effectuera les opérations suivantes:

1. Vérifiez s'il existe des écouteurs d'entité par défaut (voir Chapitre 13).
   S'ils le font, parcourez-les dans l'ordre dans lequel ils sont définis
   et recherchez les méthodes annotées avec l'événement du cycle de vie
   Annotation X. Appelez la méthode du cycle de vie sur l'écouteur si un
   méthode a été trouvée.
2. Vérifiez la superclasse ou l'entité mappée la plus élevée de la hiérarchie
   pour les classes qui ont une annotation @EntityListeners. Répéter

via les classes d'écouteur d'entités répertoriées dans l'annotation
et recherchez les méthodes annotées avec l'événement du cycle de vie
Annotation X. Appelez la méthode du cycle de vie sur l'écouteur si un
méthode a été trouvée.

3. Répétez l'étape 2 en descendant la hiérarchie sur les entités et mappées
superclasses jusqu'à ce que l'entité A soit atteinte, puis répétez-la pour
entité A.

Épisode 551

4. Vérifiez la superclasse ou l'entité mappée la plus élevée de la hiérarchie pour
méthodes annotées avec l'annotation X d'événement de cycle de vie.
Appelez la méthode de rappel sur l'entité si une méthode a été trouvée
et la méthode n'est pas également définie dans l'entité A avec le cycle de vie
annotation d'événement X dessus.

5. Répétez l'étape 2 en descendant la hiérarchie sur les entités et mappées
superclasses jusqu'à ce que l'entité A soit atteinte.

6. Appelez toutes les méthodes définies sur A et annotées avec le
annotation d'événement X du cycle de vie.

Ce processus peut être plus facile à suivre si vous pouvez voir le code qui inclut ces cas et nous passons par l'ordre dans lequel ils sont exécutés. Le listing 12-3 montre l'entité hiérarchie avec un certain nombre d'écouteurs et de méthodes de rappel.

Annonce 12-3. Utilisation des écouteurs d'entité et des méthodes de rappel dans une hiérarchie

```
@Entité
@Inheritance (stratégie = InheritanceType.JOINED)
@EntityListeners (NameValidator.class)
Public class Employee implémente NamedEntity {
    @Id id int privé;
    nom de chaîne privé;
    @Transient private long syncTime;

    public String getName () {nom de retour; }

    @PostPersist
    @PostUpdate
    @PostLoad
    private void resetSyncTime () {syncTime = System.currentTimeMillis (); }
    // ...
}

interface publique NamedEntity {
    public String getName ();
}
```


Épisode 552

```
@Entité
@ExcludeSuperclassListeners
@EntityListeners (LongNameValidator.class)
Public class ContractEmployee étend Employee {
    private int dailyRate;
    terme int privé;

    @PrePersist
    public void verifyTerm () {...}
    // ...
}

@MappedSuperclass
@EntityListeners (EmployeeAudit.class)
classe abstraite publique CompanyEmployee étend Employee {
    vacances int protégées;
    // ...

    @PrePersist
    @PreUpdate
    public void verifyVacation () {...}
}

@Entité
public class FullTimeEmployee étend CompanyEmployee {
    long salaire privé;
    pension longue privée;
    // ...
}

@Entité
@EntityListeners ({})
public class PartTimeEmployee étend CompanyEmployee {
    flotteur privé hourlyRate;
    // ...
```


Épisode 553


```
    @PrePersist
    @PreUpdate
    public void verifyVacation () {...}
}

public class EmployeeAudit {
    @PostPersist
    public void auditNewHire (CompanyEmployee emp) {...}
}

Public class NameValidator {
    @PrePersist
    public void validateName (NamedEntity obj) {...}
}

public class LongNameValidator {
    @PrePersist
```

```
        public void validateLongName (NamedEntity obj) {...}
}

public class EmployeeDebugListener {
        @PrePersist
        public void prePersist (Employé emp) {
                System.out.println ("Persistez appelé sur:" + emp);
        }

        @PreUpdate
        public void preUpdate (Employé emp) {...}

        @PreRemove
        public void preRemove (Employé emp) {...}

        @PostLoad
        public void postLoad (Employé emp) {...}
}
```

C'est un exemple assez complexe à étudier, et le moyen le plus simple de l'utiliser est de dire ce qui se passe quand un événement donné se produit pour une entité spécifique. Nous supposerons que le La classe EmployeeDebugListener a été définie dans le fichier de mappage XML comme entité par défaut écouteur pour toutes les entités.

Épisode 554

Voyons ce qui se passe lorsque nous créons une nouvelle instance de PartTimeEmployee et passez-le à em.persist (). Parce que la première étape est toujours d'appeler la valeur par défaut écouteurs, et notre auditeur par défaut a en effet une méthode PrePersist dessus, le La méthode EmployeeDebugListener.prePersist () sera appelée en premier.

L'étape suivante consiste à parcourir la hiérarchie à la recherche d'écouteurs d'entités. le la première classe que nous trouvons est la classe Employee, qui définit un écouteur d'entité NameValidator. La classe NameValidator définit une méthode PrePersist, donc la méthode suivante à obtenir exécuté est NameValidator.validateName (). La classe suivante que nous avons frappée en descendant le hiérarchie est la classe CompanyEmployee. Cette classe définit un écouteur EmployeeAudit qui n'a pas de méthode PrePersist dessus, alors nous la sautons.

Ensuite, nous arrivons à la classe PartTimeEmployee qui a un @EntityListeners annotation mais ne définit aucun écouteur. Il s'agit essentiellement d'une fausse alarme qui pas vraiment écraser quoi que ce soit, mais c'est simplement un no-op en termes d'ajout d'auditeurs (probablement un reste d'un auditeur qui était autrefois là mais qui a depuis été supprimé).

La phase suivante du processus consiste à commencer à rechercher des méthodes de rappel sur les entités et superclasses cartographiées. Une fois de plus, nous commençons en haut de la hiérarchie et regardons à la classe Employee pour voir si une méthode PrePersist existe, mais aucune ne l'est. Nous avons PostPersist et d'autres, mais pas de PrePersist. Nous continuons jusqu'à CompanyEmployee et voyez une méthode PrePersist appelée verifyVacation (), mais en regardant vers le bas sur l'entité PartTimeEmployee, nous constatons que la méthode a été remplacée par une méthode verifyVacation () qui contient également une annotation @PrePersist. Cette est un cas de substitution de la méthode de rappel et aboutira à PartTimeEmployee. La méthode verifyVacation () est appelée à la place de CompanyEmployee. verifyVacation (). Nous avons enfin terminé, et l'entité sera persistante.

L'événement suivant pourrait alors être un événement PostPersist sur la même entité lors de la validation temps. Cela contournera l'écouteur par défaut car il n'y a pas de méthode PostPersist dans EmployeeDebugListener et contournera également NameValidator car il il n'y a pas non plus de méthode d'événement PostPersist. Le prochain auditeur qu'il essaiera sera la classe d'écouteur EmployeeAudit, qui inclut une méthode PostPersist appelée auditNewHire (), qui sera ensuite appelée. Il n'y a plus d'auditeurs à examiner, donc nous passons aux méthodes de rappel et trouvons la méthode resetSyncTime () dans Employee. Celui-ci est appelé, et parce que nous ne trouvons plus de méthodes de rappel PostPersist dans le

hiérarchie, nous avons terminé.

La prochaine chose que nous pouvons essayer est la persistance d'un ContractEmployee. C'est un simple structure de persistance avec uniquement les entités Employee et ContractEmployee. Quand

Épisode 555

nous créons un ContractEmployee et l'événement PrePersist est déclenché, nous obtenons d'abord notre par défaut EmployeeDebugListener.prePersist (), puis passez au traitement les écouteurs d'entité. La courbe est que l'annotation @ExcludeSuperclassListeners est présente sur le ContractEmployee, donc la méthode NameValidator.validateName () qui aurait autrement été invoqué ne sera pas pris en compte. Nous allons plutôt droit au @EntityListeners annotation sur la classe ContractEmployee et constatez que nous avons besoin pour regarder LongNameValidator. Lorsque nous le faisons, nous constatons qu'il a un validateLongName () méthode que nous exécutons, puis passons à l'exécution des méthodes de rappel. Il y a méthodes de rappel dans les deux classes de la hiérarchie et Employee.resetSyncTime () La méthode est appelée en premier, suivie de la méthode ContractEmployee.verifyTerm ().

Validation

Référencement 12-2 a montré un exemple d'écouteur d'entité dans lequel nous avons validé qu'un nom n'était pas plus long que prévu avant que l'entité ne soit conservée dans la base de données. La contrainte peut avoir été imposée par une définition de schéma de base de données ou une entreprise règle dans l'application. Dans Java EE 6, la validation a été introduite comme un aspect distinct de L'application; un mécanisme a été développé et standardisé (dans JSR 303[2]) pour le Plate-forme. Il a également été conçu pour fonctionner dans un environnement Java SE autonome. En Java EE 7, quelques éditions mineures ont été ajoutées à la version 1.1 de la spécification Bean Validation dans le cadre de JSR 349. Nous donnerons un aperçu de la validation et de la manière dont elle peut être utilisée, mais pour plus de détails, nous vous renvoyons à la spécification JSR 349 [3] développée dans le Java Processus communautaire.

La validation a un modèle d'annotation et une API dynamique qui peuvent être appelées à partir de n'importe quelle couche, et sur presque n'importe quel haricot. Les annotations de contrainte sont placées sur le champ ou propriété de l'objet à valider, ou même sur la classe d'objet elle-même, et plus tard lorsque le validateur exécute les contraintes seront vérifiées. La spécification de validation fournit un quelques annotations de contraintes prédéfinies pouvant être utilisées par n'importe quel développeur de bean, mais plus il comprend surtout un modèle de création de contraintes définies par l'utilisateur et spécifiques à logique ou schémas d'application.

[2] Voirwww.jcp.org/en/jsr/summary?id=303
[3] Voirwww.jcp.org/en/jsr/summary?id=349

Épisode 556

Utilisation des contraintes

L'ajout de contraintes à un objet peut être aussi simple que d'annoter la classe, ses champs ou

ses propriétés de style JavaBean. Par exemple, en utilisant la contrainte @Size intégrée, nous pouvons valider l'entité Employé dans le Listing 12-2 et nous éviter d'avoir à coder une entité auditeur. Le Listing 12-4 montre une classe d'objets Employee contenant des contraintes.

Annonce 12-4. Utilisation de contraintes prédéfinies

```
Employé de classe publique {
    @NotNull
    ID entier;

    ID d'entier privé;
    @NotNull (message = "Le nom de l'employé doit être spécifié")
    @ Taille (max = 40)
    nom de chaîne privé;

    @Passé
    private Date startDate;
    // ...
}
```

La contrainte @Size garantit que le nom se trouve dans la plage de 40 caractères, tout comme notre le validateur d'écouteur d'entité l'a fait. Nous avons également ajouté une contrainte @NotNull pour valider qu'un nom a toujours été spécifié. Si cette contrainte n'était pas satisfaite dans notre ancienne entité Employé, notre auditeur aurait explosé avec une NullPointerException car il n'a pas vérifié pour null avant de vérifier la longueur. En utilisant la validation, ces préoccupations sont séparées et peut être imposée indépendamment. L'annotation @Past validera que la date de début est un date valide qui se produit dans le passé. Notez que null est une valeur valide dans ce cas. Si nous voulions pour nous assurer qu'une date était présente, nous l'annotions également avec @NotNull.

Dans la deuxième contrainte @NotNull, nous avons également inclus un message à inclure dans l'exception si la vérification des contraintes échoue. Chaque annotation de contrainte intégrée a un message élément qui peut être spécifié pour remplacer le message par défaut qui serait généré[4].

[4] Certains mécanismes de localisation sont intégrés à l'interpolateur de message de validation, mais l'interpolation de message peut également être connectée au validateur pour effectuer la localisation de manière personnalisée façons.

Épisode 557

L'ensemble complet des contraintes intégrées pouvant être utilisées avec la validation est affiché dans Tableau 12-1 . Ceux-ci sont définis dans le package javax.validation.constraints.

Tableau 12-1. *Contraintes de validation intégrées*

Contrainte	Les attributs[5]	La description
@Nul		L'élément doit être nul.
@NotNull		L'élément ne doit pas être nul.
@AssertVrai		l'élément doit être vrai.
@AssertFalse		l'élément doit être faux.
@Min	valeur longue ()	l'élément doit avoir une valeur supérieure ou égale à le minimum.
@Max	valeur longue ()	l'élément doit avoir une valeur inférieure ou égale à maximum.
@DecimalMin	Valeur de chaîne()	l'élément doit avoir une valeur supérieure ou égale à le minimum.
@DecimalMax	Valeur de chaîne()	l'élément doit avoir une valeur inférieure ou égale à maximum.

@Taille	int min ()	L'élément doit être d'une longueur dans les limites spécifiées.
	int max ()	
@Chiffres	entier entier ()	l'élément doit être un nombre compris dans la plage spécifiée.
	fraction int ()	
@Passé		l'élément doit être une date dans le passé.
@Futur		l'élément doit être une date dans le futur.
@Modèle	L'élément string regexpr ()	doit correspondre à l'expression régulière spécifiée.
	Flag [] flags	(Les indicateurs offrent des paramètres d'expression régulière.)

[5] Seuls les attributs non standard sont répertoriés. Les attributs standard / requis sont traités dans la section sur la création de nouvelles contraintes.

Épisode 558

Invoquer la validation

La classe API principale pour appeler la validation sur un objet est javax.validation.
Classe de validateur. Une fois qu'une instance de Validator est obtenue, la commande validate ()[6] méthode peut être invoqué dessus, en passant l'objet à valider.

La validation est conçue de manière similaire à JPA à bien des égards. Il est divisé en un ensemble de API et une implémentation de validation, ou fournisseur de validation, et la manière dont les fournisseurs se faire connaître est en utilisant le même modèle de fournisseur de services. Les fournisseurs contiennent Fichiers META-INF / services qui indiquent leurs classes SPI à appeler.

Comme JPA, la validation est utilisée légèrement différemment selon qu'elle est utilisée en mode conteneur ou en mode Java SE. Dans un conteneur, une instance de Validator peut être injecté dans tout composant Java EE prenant en charge l'injection. L'exemple de définition d'un Le bean session sans état du Listing 12-5 montre que l'injection Java EE @Resource standard l'annotation peut être utilisée.

Annonce 12-5. Injection d'un validateur

```
@Ressource
Validateur validateur;

public void newEmployee (Employee emp) {

    Définissez <ConstraintViolation <Employee>> violations = validateur.
    valider (emp);
    // ...
}
```

Dans un environnement non-conteneur, une instance de Validator est obtenue à partir d'un javax.validation.ValidatorFactory, qui peut à son tour être acquis à partir d'un bootstrap javax.validation.Validation, comme suit:

```
ValidatorFactory usine =
Validation.buildDefaultValidatorFactory ();

Validator validator = factory.getValidator ();
```

[6] méthodes validateProperty () et validateValue () sont également disponibles, mais nous ne discutons que

représentant la méthode validate ().

Épisode 559

Une fois le validateur obtenu, on peut invoquer la méthode validate () dessus comme dans Référencement 12-5.

Lorsque la méthode de validation échoue et qu'une contrainte n'est pas satisfaite, un ValidationException est levée avec une chaîne de message d'accompagnement qui est dictée par la définition de la contrainte non satisfaite, notamment par la valeur de son élément d'annotation de message décrit dans la section précédente.

Groupes de validation

Il se peut que le même objet doive être validé à des moments différents pour plusieurs différents ensembles de contraintes. Pour ce faire, nous créerions des groupes de validation séparés et spécifiez le ou les groupes auxquels appartient la contrainte. Lorsqu'un groupe est passé comme un argument lors de la validation, toutes les contraintes qui font partie de ce groupe sont vérifiées validité. Quand aucun groupe n'est spécifié sur une contrainte ou comme argument de la validation () méthode, le groupe Par défaut est utilisé.

Les groupes sont définis et référencés en tant que classes, donc quelques exemples de définition les groupes peuvent être les suivants:

```
interface publique FullTime étend la valeur par défaut {}
interface publique PartTime étend la valeur par défaut {}
```

Il n'y a rien de valeur au sein de la classe de groupe, à part la classe elle-même, alors ils sont définis comme de simples interfaces. Bien que ces deux extensions étendent javax.validation. groupes Interface de groupe par défaut (une interface définie par la spécification) certainement pas une exigence, mais comme un groupe de sous-classes inclut des groupes, il hérite, il est pratique de les faire étendre le groupe par défaut.

Référencement 12-6 montre une classe d'objets qui utilise ces groupes pour s'assurer que le bon Le champ salaire est défini selon que l'employé est embauché pour travailler à temps plein ou à temps partiel.

Annonce 12-6. Utilisation de groupes de contraintes Integer, Long, Byte, Double, Float, Short

```
Employé de classe publique {
    @NotNull

    ID d'entier privé;
```


Épisode 560


```
    @NotNull
    @ Taille (max = 40)
    nom de chaîne privé;

    @NotNull (groupes = FullTime.class)
    @Null (groupes = PartTime.class)
    Salaire long privé;
```

```
    @NotNull (groupes = PartTime.class)
    @Null (groupes = FullTime.class)
    Salaire horaire double privé;

    // ...
}
```

Parce que les contraintes sur les champs id et name n'ont pas de groupe affecté à eux, ils sont supposés appartenir au groupe par défaut et seront vérifiés à chaque fois soit le groupe par défaut, soit aucun groupe, est passé à la méthode validate (). cependant, car nos deux nouveaux groupes ont étendu Default, les deux champs seront également validés lorsque les groupes FullTime ou PartTime sont transmis. Si nous n'avions pas étendu Par défaut dans nos deux définitions de groupe, nous aurions dû inclure les deux groupes dans les contraintes sur les champs id et name si nous voulions qu'ils soient vérifiés lorsque l'un des les deux groupes ont été spécifiés dans la méthode validate (), comme indiqué dans la liste 12-7 .

Annonce 12-7. Spécification de plusieurs groupes

```
Employé de classe publique {
    @NotNull (groupes = {FullTime.class, PartTime.class})
    ID d'entier privé;

    @NotNull (groupes = {FullTime.class, PartTime.class})
    @Size (groupes = {FullTime.class, PartTime.class}, max = 40)
    nom de chaîne privé;
    // ...
}
```


Épisode 561

Créer de nouvelles contraintes

L'un des aspects les plus précieux de la validation est la possibilité d'ajouter de nouvelles contraintes pour une application donnée, ou même pour partager entre les applications. Les contraintes peuvent être mis en œuvre pour être spécifiques à l'application et liés à une logique métier particulière, ou peuvent être généralisés et regroupés dans des bibliothèques de contraintes pour une réutilisation. Nous n'entrerons pas dans beaucoup de détails dans ce domaine, mais j'espère que vous aurez l'idée de quelques simples exemples, et si vous voulez faire une programmation de validation plus approfondie, vous pouvez regarder dans la spécification plus loin. Malheureusement, même après la deuxième version, il y a toujours pas de références disponibles au moment de la rédaction de cet article, enregistrez les spécifications réelles précédemment référencé.

Chaque nouvelle contrainte est composée de deux parties: la définition d'annotation et le classes d'implémentation ou de validation. Nous avons montré des exemples de annotations dans nos exemples jusqu'à présent, afin que vous sachiez comment les utiliser. Cependant, vous n'avez pas vu les classes d'implémentation associées pour ces annotations intégrées car ils sont supposés être implémentés par le fournisseur de validation. Lorsque vous écrivez votre propre contrainte, vous devez fournir une classe d'implémentation pour chaque type différent de l'objet qui peut être annoté par votre nouvelle annotation de contrainte. Puis, pendant la processus de validation, le validateur appellera la classe d'implémentation correspondante pour le type d'objet en cours de validation.

Annotations de contrainte

Ce n'est pas une tâche difficile de définir une nouvelle annotation de contrainte, mais il en faut quelques-unes
ingrédients à prendre en compte lors de cette opération. Ils sont inclus dans la contrainte simple
définition d'annotation dans la liste 12-8 qui marque un nombre pair. Notez que c'est
bonne pratique pour documenter dans la définition de contrainte les types qu'elle peut annoter.

Annonce 12-8. Définition d'une annotation de contrainte

```
/ **
  * Indiquez qu'un nombre doit être pair.
  * Peut être appliqué sur des champs ou des propriétés de type Integer.
  * /
@Constraint (validatedBy = {EvenNumberValidator.class})
@Target ({METHOD, FIELD})
@Retention (RUNTIME)
```

Épisode 562


```
public @interface Even {
        String message () default "Le nombre doit être pair";
        Classe <?> [] Groups () default {};
        Classe <? étend Payload> [] payload () default {};
        booléen includeZero () default true;
}
```

Comme le montre l'exemple, la stratégie @Retention doit être définie sur RUNTIME et le
@Target doit inclure au moins un ou plusieurs types, FIELD, METHOD ou ANNOTATION_TYPE.
D'autres types de cibles peuvent également être inclus, mais seuls ceux-ci doivent être découverts par
fournisseurs de validation. La définition doit également être annotée avec la méta @Constraint
annotation, qui indique la classe d'implémentation pour accompagner cette annotation
et contient le code de validation. Nous expliquons comment créer des classes d'implémentation dans
la section suivante.

Il y a trois éléments obligatoires dans chaque annotation de contrainte. nous
ont déjà discuté du premier, l'élément message, et comment il peut être utilisé pour définir
un message d'exception par défaut lorsque les contraintes ne sont pas respectées. Nous avons discuté de groupes dans le
section précédente, vous savez donc également que l'élément groups est utilisé lors de la validation
d'une contrainte doit se produire dans le cadre d'un ou plusieurs ensembles d'autres vérifications de contraintes associées.
Le troisième élément est l'élément de charge utile, qui est juste un endroit pour transmettre
métadonnées à la classe de validation. Les classes passées à cet élément sont définies par
le créateur de la contrainte et doit étendre le javax.validation défini par la spécification.
Type de PayLoad. N'importe quel nombre d'éléments supplémentaires spécifiques aux contraintes peuvent également être ajoutés.
Dans cet exemple, nous avons ajouté une option pour inclure ou exclure zéro comme nombre pair.

Bien que chacun des éléments requis ait ses objectifs, il est plutôt regrettable
qu'ils sont tous requis. Les exiger simplement parce qu'ils pourraient être utiles à certains
applications rappelle quelque peu l'époque des premiers EJB, lorsque les applications étaient
forcés d'insérer du code supplémentaire (qu'ils n'avaient aucune envie d'inclure ou d'utiliser) simplement parce que
le cahier des charges dit qu'ils doivent le faire. Nous espérons que cela sera corrigé dans une future version du
spécification de validation.

Classes d'implémentation de contraintes

Pour chaque annotation de contrainte, il doit y avoir une ou plusieurs implémentations de contrainte
Des classes. Chaque classe doit implémenter javax.validation.ConstraintValidator
interface et fournissez la logique pour valider la valeur vérifiée. Le listing 12-9 est le
classe d'implémentation pour accompagner notre annotation de contrainte @Even.

Épisode 563

Annonce 12-9. Définition d'une classe d'implémentation de contrainte

```
classe publique EvenNumberValidator
        implémente ConstraintValidator <Even, Integer> {

    boolean includesZero;

    public void initialize (même contrainte) {
        includesZero = constraint.includeZero ();
    }

    public boolean isValid (valeur entière,
                            ConstraintValidatorContext ctx) {
        if (valeur == null)
            retourne vrai;
        si (valeur == 0)
            return includesZero;
        valeur de retour% 2 == 0;
    }
}
```

La classe de validation implémente le javax.validation.ConstraintValidator interface avec deux paramètres de type. Le premier type est le type d'annotation de contrainte, et le second est le type de valeur que la classe d'implémentation s'attend à valider. Dans notre cas, nous validons des types entiers, ce qui signifie que la contrainte @Even l'annotation peut être appliquée aux champs ou aux getters de type Integer, ou à tout sous-type (dont il n'y en a pas dans ce cas). Le type int primitif correspondant au type de wrapper est également un type de candidat.

Les deux méthodes qui doivent être implémentées sont initialize () et isValid (). le La méthode initialize () est appelée en premier et transmet l'instance d'annotation qui a causé la classe de validation à invoquer en premier lieu. Nous prenons n'importe quel état de l'instance et initialiser la classe de validation avec elle, donc quand la méthode isValid () est appelée nous peut valider la valeur qui nous est passée en fonction des paramètres de la contrainte qui est l'annoter. Le paramètre supplémentaire ConstraintValidatorContext peut être utilisé pour génération d'erreur plus avancée, mais elle est quelque peu obtuse et dépasse le cadre de notre aperçu de la validation.

Épisode 564

Validation dans JPA

Maintenant que vous avez quelques connaissances de base en validation, nous sommes prêts à mettre les choses dans un Contexte JPA. Lors de la validation des entités JPA, une intégration spécifique est requise avec le Fournisseur JPA. Il y a plusieurs raisons à cette intégration.

D'abord et avant tout, une entité peut avoir des attributs chargés paresseusement, et parce qu'un le validateur n'a pas de dépendance ou de connaissance de JPA, il ne saurait un attribut n'a pas été chargé. Le processus de validation pourrait provoquer involontairement graphe d'objets entier à charger en mémoire! Un autre cas est si la validation se produit sur une entité JPA côté client et les attributs déchargés ne sont même pas chargeables. Dans

dans ce cas, la validation produirait une exception, pas aussi grave que le chargement de l'ensemble graphique d'objet, mais toujours clairement indésirable.

La raison la plus pratique d'une intégration JPA est que le plus souvent nous voulons une validation à appeler automatiquement à des phases spécifiques du cycle de vie. Rappelez-vous que dans notre exemple en Référencement 12-2 nous avons validé lors de la phase PrePersist pour nous assurer de ne pas entité dans un état non valide. Il s'avère que les événements de cycle de vie les plus pratiques à déclencher validation à sont PrePersist, PreUpdate et PreRemove, donc si la validation est activée, ces événements amèneront le validateur à faire son travail. Le listing 12-10 montre une entité et un type intégrable avec des contraintes de validation sur eux.

Annonce 12-10. Valider une entité

```
@Entité
Employé de classe publique {
    @Id @NotNull
    id int privé;

    @NotNull
    @ Taille (max = 40)
    nom de chaîne privé;

    @Passé
    @Temporal (TemporalType.DATE)
    private Date startDate;

    @Embedded
    @Valide
    informations privées EmployeeInfo;
```

Épisode 565


```
    @ManyToOne
    adresse d'adresse privée;
    // ...
}

@Embeddable
public class EmployeeInfo {
    @Passé
    @Temporal (TemporalType.DATE)
    date privée dob;

    @Embedded
    conjoint privé PersonInfo;
}
```

Lorsqu'une entité est validée, chacun des champs ou propriétés, voire le type lui-même, est validé selon les règles de validation habituelles. Cependant, la spécification de validation indique que lorsqu'une annotation @Valid est présente sur un champ ou une propriété, la validation Le processus passe à l'objet stocké dans ce champ ou cette propriété. Les éléments incorporables peuvent éventuellement être annoté avec @Valid afin d'être parcouru lors de la validation, mais les relations peuvent ne pas. En d'autres termes, l'objet EmployeeInfo dans le champ info sera validé lorsque le L'employé est validé, mais le conjoint ne le sera pas, et les entités liées, telles que l'adresse, ne seront pas non plus validés à moins qu'ils n'aient eux-mêmes été conservés, mis à jour ou supprimés.

Activation de la validation

Lorsqu'aucun paramètre de remplacement n'est présent au niveau de la configuration JPA, la validation est activée par défaut lorsqu'un fournisseur de validation est sur le chemin de classe. Pour contrôler explicitement si

la validation doit être activée ou désactivée, il existe deux paramètres possibles.

- L'élément mode validation dans le fichier persistence.xml. Cette
  L'élément peut être défini sur l'une des trois valeurs possibles.
  - AUTO: activer la validation lorsqu'un fournisseur de validation est présent sur le classpath (par défaut).
  - CALLBACK: active la validation et renvoie une erreur si aucune validation fournisseur est disponible.
  - NONE: désactiver la validation.

Épisode 566

- La persistance javax.persistence.validation.mode
  propriété. Cette propriété peut être spécifiée dans la carte transmise à la méthode createEntityManagerFactory () et remplace la réglage du mode de validation si présent. Les valeurs possibles sont la chaîne équivalents des valeurs du mode de validation (AUTO, CALLBACK et AUCUN) et ont exactement les mêmes significations que leur mode de validation homologues.

Définition des groupes de validation du cycle de vie

Par défaut, chacun des événements du cycle de vie PrePersist et PreUpdate déclenchera la validation sur l'entité affectée, immédiatement après le rappel de l'événement, en utilisant la valeur par défaut groupe de validation. Aucun groupe ne sera validé, par défaut, lors de la phase de pré-suppression. À modifier les groupes en cours de validation pour les trois types d'événements de cycle de vie différents, l'un des les propriétés suivantes peuvent être spécifiées, soit en tant que propriétés dans la section des propriétés de le fichier persistence.xml, ou dans le Map passé dans createEntityManagerFactory ():

- javax.persistence.validation.group.pre-persist: définir les groupes à valider au moment du PrePersist.
- javax.persistence.validation.group.pre-update: définir les groupes pour valider au moment de la pré-mise à jour.
- javax.persistence.validation.group.pre-remove: définir les groupes pour valider au moment de la pré-suppression.

En définissant ces propriétés sur un ou plusieurs groupes particuliers, vous pouvez isoler types de validation qui sont effectués sur les entités au cours de différents événements du cycle de vie. Pour Par exemple, vous pouvez créer des groupes appelés Créer, Mettre à jour et Supprimer, puis lorsque vous souhaitez qu'une sorte de validation se produise sur un ou plusieurs de ces événements, il vous suffit définir les groupes sur les contraintes pertinentes à vérifier. En fait, il est plus courant pour la validation doit avoir lieu lors de la création et de la mise à jour, et pour que la même validation se produise aux deux étapes, de sorte que le groupe Par défaut sera le plus souvent suffisant pour les deux. Cependant, vous peut vouloir spécifier un groupe distinct pour PreRemove, comme indiqué dans la liste 12-11 .

Épisode 567

Annonce 12-11. Validation variable selon les événements du cycle de vie

```
@Entité
Employé de classe publique {
        @Id @NotNull
        id int privé;

        @NotNull
        @ Taille (max = 40)
        nom de chaîne privé;

        @Passé
        @Temporal (TemporalType.DATE)
        private Date startDate;

        @Min (groupes = Remove.class, valeur = 0)
        @Max (groupes = Remove.class, valeur = 0)
        longues vacances privées;

        // ...
}
```

La validation en Listing 12-11 garantit qu'aucun employé n'est retiré du système vacances dues ou dues. Le reste des contraintes est validé pendant les événements PrePersist et PreUpdate. Cela suppose que le groupe Remove a été défini et que la propriété suivante est présente dans le fichier persistence.xml:

```
<property name = "javax.persistence.validation.group.pre-remove"
                value = "Supprimer" />
```

Astuce, il n'est actuellement pas portable dans Jpa pour définir les groupes sur une base par entité, bien que certains fournisseurs puissent fournir de telles capacités. fournisseurs qui prennent en charge cela permettrait généralement au nom de l'entité d'être un suffixe supplémentaire sur la propriété nom (par exemple, javax.persistence.validation.group.pre-remove. Employé).

Épisode 568

Concurrence

La concurrence d'accès aux entités et aux opérations d'entités n'est pas fortement spécifiée, mais sont quelques règles qui dictent ce à quoi nous pouvons et ne pouvons pas nous attendre. Nous passons en revue ces derniers et laissons le Il appartient aux vendeurs d'expliquer dans la documentation leurs implémentations respectives.

Opérations d'entité

Une entité gérée appartient à un seul contexte de persistance et ne doit pas être gérée par plus d'un contexte de persistance à un moment donné. Ceci est une responsabilité d'application, cependant, et ne peut pas nécessairement être appliquée par le fournisseur de persistance. Fusionner le la même entité dans deux contextes de persistance ouverts différents pourrait produire des résultats non définis.

Les gestionnaires d'entités et les contextes de persistance qu'ils gèrent ne sont pas destinés à accessible par plusieurs threads exécutés simultanément. L'application ne peut pas

s'attend à ce qu'il soit synchronisé et est responsable de s'assurer qu'il reste dans le
fil qui l'a obtenu.

Accès aux entités

Les applications ne peuvent pas accéder à une entité directement à partir de plusieurs threads pendant
géré par un contexte de persistance. Une application peut toutefois choisir d'autoriser
entités accessibles simultanément lorsqu'elles sont détachées. S'il choisit de le faire,
la synchronisation doit être contrôlée par les méthodes codées sur l'entité.
L'accès simultané à l'état d'entité n'est cependant pas recommandé, car le modèle d'entité
ne se prête pas bien aux modèles concurrents. Il serait préférable de simplement copier le
entité et transmettez l'entité copiée à d'autres threads pour y accéder, puis fusionnez les modifications
retour dans un contexte de persistance quand ils ont besoin d'être persistés.

Actualisation de l'état de l'entité

La méthode refresh () de l'interface EntityManager peut être utile dans les situations où
nous savons ou soupçonnons qu'il y a des changements dans la base de données que nous n'avons pas dans notre
entité gérée. L'opération d'actualisation s'applique uniquement lorsqu'une entité est gérée car
lorsque nous sommes détachés, il suffit généralement d'émettre une requête pour obtenir une version mise à jour
de l'entité à partir de la base de données.

Épisode 569

L'actualisation a plus de sens, plus la durée du contexte de persistance est longue
le contient. L'actualisation est particulièrement pertinente lors de l'utilisation d'une application étendue ou
contexte de persistance géré car il prolonge l'intervalle de temps pendant lequel une entité est
effectivement mis en cache dans le contexte de persistance indépendamment de la base de données.

Pour actualiser une entité gérée, nous appelons simplement refresh () sur le gestionnaire d'entités. Si la
l'entité que nous essayons d'actualiser n'est pas gérée, une exception IllegalArgumentException
sera jeté. Pour clarifier certains des problèmes liés à l'opération d'actualisation, nous utilisons le
exemple de bean montré dans l'extrait 12-12.

Annonce 12-12. Actualisation périodique d'une entité gérée

```
@Stateful
@TransactionAttribute (TransactionAttributeType.NOT_SUPPORTED)
public class EmployeeService {
    public statique final long REFRESH_THRESHOLD = 300000;

    @PersistenceContext (unitName = "EmployeeService",
                         type = PersistenceContextType.EXTENDED)
    EntityManager em;
    Employé emp;
    long loadTime;

    public void loadEmployee (int id) {
        emp = em.find (Employee.class, id);
        si (emp == null)
            lancer une nouvelle IllegalArgumentException (
                "Identifiant d'employé inconnu:" + id);
        loadTime = System.currentTimeMillis ();
    }

    public void deductEmployeeVacation (int days) {
        refreshEmployeeIfNeeded ();
```

```
        emp.setVacationDays (emp.getVacationDays () - jours);
    }

    public void AdjustEmployeeSalary (long salaire) {
        refreshEmployeeIfNeeded ();
        emp.setSalary (salaire);
    }
```


Épisode 570


```
    @Retirer
    @TransactionAttribute (TransactionAttributeType.REQUIRED)
    public void done () {}

    private void refreshEmployeeIfNeeded () {
        if ((System.currentTimeMillis () - loadTime)> REFRESH_THRESHOLD) {
            em.refresh (emp);
            loadTime = System.currentTimeMillis ();
        }
    }

    // ...
}
```

Le bean du Listing 12-12 utilise un contexte de persistance étendu afin de garder un Instance d'employé gérée tandis que diverses opérations lui sont appliquées via l'entreprise méthodes du bean session. Cela pourrait permettre un certain nombre d'opérations de modification dessus avant de valider les modifications, mais nous devons inclure seulement quelques opérations pour cet exemple.

Regardons ce haricot en détail. La première chose à noter est que la transaction par défaut L'attribut est passé de REQUIRED à NOT_SUPPORTED. Cela signifie qu'en tant que L'instance de l'employé est modifiée par les différentes méthodes commerciales du bean, celles les modifications ne seront pas écrites dans la base de données. Cela ne se produira que lorsque le paramètre terminé () est invoquée, qui a un attribut de transaction REQUIRED. C'est le seul méthode sur le bean qui associera le contexte de persistance étendue à un transaction et provoquer sa synchronisation avec la base de données.

La deuxième chose intéressante à propos de ce bean est qu'il stocke le temps que l'employé instance a été accédée pour la dernière fois depuis la base de données. Parce que l'instance de bean peut exister pour un depuis longtemps, les méthodes métier utilisent la méthode refreshEmployeeIfNeeded () pour voir si elle depuis trop longtemps depuis la dernière actualisation de l'instance Employee. Si le seuil de rafraîchissement a été atteint, la méthode refresh () est utilisée pour mettre à jour l'état de l'employé à partir du base de données.

Malheureusement, l'opération d'actualisation ne se comporte pas comme l'auteur de la session haricot attendu. Lorsque l'actualisation est appelée, elle écrasera l'entité gérée par dans la base de données, entraînant la perte des modifications apportées à l'entité. Par exemple, si le salaire est ajusté et cinq minutes plus tard, les vacances sont ajustées, le L'instance d'employé sera actualisée, entraînant la perte de la modification précédente du salaire.

Épisode 571

Donc, même si l'exemple de la liste 12-12 effectue en effet un rafraîchissement périodique du gérée, il en résulte non seulement une utilisation inappropriée de refresh () mais aussi une

résultat préjudiciable à l'application.

Alors, quand le rafraîchissement est-il valide pour les objets que nous modifions? La réponse est, pas comme souvent comme vous le pensez. L'un des principaux cas d'utilisation consiste à «annuler» ou à ignorer les modifications apportées dans la transaction en cours, les ramenant à leur valeur d'origine. Il peut également être utilisé dans des contextes de persistance de longue durée où des entités gérées en lecture seule sont mises en cache. Dans ces scénarios, l'opération refresh () peut restaurer en toute sécurité une entité à son état enregistré dans la base de données. Cela aurait pour effet de récupérer les modifications apportées dans la base de données depuis le dernier chargement de l'entité dans le contexte de persistance. le la stipulation est que l'entité doit être en lecture seule ou ne pas contenir de modifications.

Rappelez-vous notre session d'édition dans la liste 6-34 . En utilisant refresh (), nous pouvons ajouter la capacité pour rétablir une entité lorsque l'utilisateur décide d'annuler ses modifications apportées à une modification d'employé session. Référencement12-13 montre le bean avec sa méthode revertEmployee () supplémentaire.

Annonce 12-13. Session d'édition des employés avec retour

```
@Stateful
@TransactionAttribute (TransactionAttributeType.NOT_SUPPORTED)
Public class EmployeeEdit {
    @PersistenceContext (unitName = "EmployeeService",
                                type = PersistenceContextType.EXTENDED)
    EntityManager em;
    Employé emp;

    public void begin (int id) {
        emp = em.find (Employee.class, id);
        if (emp == null) {
            throw new IllegalArgumentException ("Identifiant d'employé inconnu:" + id);
        }
    }

    public Employee getEmployee () {return emp; }

    public Employee revertEmployee () {
        em.refresh (emp);
        return emp;
    }
```

Épisode 572


```
    @Retirer
    @TransactionAttribute (TransactionAttributeType.REQUIRES_NEW)
    public void save () {}

    @Retirer
    public void cancel () {}
}
```

Les opérations d'actualisation peuvent également être mises en cascade entre les relations. Ceci est fait sur le annotation de relation en définissant l'élément en cascade pour inclure la valeur REFRESH. Si la valeur REFRESH n'est pas présente dans l'élément en cascade, l'actualisation s'arrêtera au entité source. Le listing 12-14 montre comment définir l'opération en cascade REFRESH pour un relation plusieurs à un.

Annonce 12-14. Mise en cascade d'une opération REFRESH

```
@Entité
Employé de classe publique {
    @Id id int privé;
```

```
        nom de chaîne privé;
        @ManyToOne (cascade = {CascadeType.REFRESH})
        gestionnaire d'employé privé;
        // ...
    }
```

Verrouillage

Le verrouillage des surfaces à de nombreux niveaux différents est intrinsèque à JPA. Il est utilisé et supposé à divers points de l'API et de la spécification. Si votre application est simple ou complexe, il y a de fortes chances que vous utilisiez le verrouillage quelque part en cours de route.

Alors que nous discutons de tous les verrouillages définis et utilisés dans JPA, nous nous concentrons principalement sur verrouillage optimiste car c'est non seulement le moyen le plus répandu mais aussi le plus utile pour mettre à l'échelle une application.

Épisode 573

Verrouillage optimiste

Lorsque nous parlons de verrouillage, nous parlons souvent de verrouillage *optimiste* . L'optimiste le modèle de verrouillage repose sur l'hypothèse qu'il y a de fortes chances que la transaction en les modifications apportées à une entité seront les seules à modifier réellement l'entité pendant cet intervalle. Cela se traduit par la décision de ne pas acquérir de verrou sur l'entité jusqu'à ce que la modification soit effectivement apportée à la base de données, généralement à la fin de la transaction.

Lorsque les données sont effectivement envoyées à la base de données pour être mises à jour au moment du vidage ou à la fin de la transaction, le verrou d'entité est acquis et un contrôle est effectué sur les données dans la base de données. La transaction de vidage doit voir si une autre transaction a a commis un changement dans l'entité dans l'intervalle depuis que cette transaction l'a lu dans et l'a changé. En cas de changement, cela signifie que la transaction de vidage contient des données n'inclut pas ces modifications et ne doit pas écrire ses propres modifications dans la base de données, de peur qu'il n'écrase les modifications de la transaction intermédiaire. A ce stade, il faut rouler sauvegarder la transaction et lancer une exception spéciale appelée OptimisticLockException. L'exemple du Listing 12-15 montre comment cela peut arriver.

Annonce 12-15. Méthode qui ajuste le solde des vacances

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    public void deductEmployeeVacation (int id, int days) {
        Employé emp = em.find (Employee.class, id);
        int currentDays = emp.getVacationDays ();
        // Faites d'autres choses comme notifier le système RH, etc.
        // ...
        emp.setVacationDays (currentDays - jours);
    }
}
```

Bien que cette méthode puisse sembler assez inoffensive, ce n'est vraiment qu'un accident qui attend se passer. Le problème est le suivant. Imaginez que deux opérateurs de saisie de données RH, Frank et Betty, ont été chargés d'inscrire un arriéré d'ajustements de vacances dans le système

et ils ont tous deux saisi un ajustement pour l'employé avec l'ID 42 à la

Épisode 574

en même temps. Frank est censé déduire 1 jour de l'employé 42, tandis que Betty déduit 12 jours. La console de Frank appelle d'abord deductEmployeeVacation (), qui lit immédiatement l'employé 42 dans la base de données, trouve que l'employé 42 a 20 jours, puis continue dans l'étape de notification RH. Pendant ce temps, Betty commence à saisir ses données sur sa console, qui appelle également deductEmployeeVacation (). Il lit également l'employé 42 dans le base de données et constate que l'employé a 20 jours de vacances, mais Betty a un connexion beaucoup plus rapide au système RH. En conséquence, Betty dépasse la notification RH avant que Frank ne le fasse et continue de régler le nombre de jours de vacances à 8 avant de s'engager sa transaction et passer à l'élément suivant. Frank dépasse enfin le système RH notification et déduit 1 jour du 20, puis valide sa transaction. Si Frank s'engage, il a écrasé la déduction de Betty et l'employé 42 reçoit 12 jours supplémentaires de vacances.

Au lieu de valider la transaction de Frank, cependant, une stratégie de verrouillage optimiste découvrirait quand il était temps de s'engager que quelqu'un d'autre avait changé les vacances compter. Lorsque Frank a tenté de valider sa transaction, une exception OptimisticLockException aurait été levée, et sa transaction aurait été annulée à la place. le le résultat est que Frank devrait rentrer son changement et réessayer, ce qui est bien supérieur pour obtenir un résultat incorrect pour l'employé 42.

Gestion des versions

La question que vous vous posez peut-être est de savoir comment le fournisseur peut savoir si quelqu'un a fait des changements dans l'intervalle de temps depuis la transaction de validation lisez l'entité. La réponse est que le fournisseur gère un système de gestion des versions pour le entité. Pour ce faire, l'entité doit avoir un champ persistant dédié ou propriété déclarée pour stocker le numéro de version de l'entité obtenue dans le transaction. Le numéro de version doit également être stocké dans la base de données. Au retour à la base de données pour mettre à jour l'entité, le fournisseur peut vérifier la version de l'entité dans la base de données pour voir si elle correspond à la version obtenue précédemment. Si la version dans la base de données est la même, le changement peut être appliqué et tout se passe sans des problèmes. Si la version était supérieure, quelqu'un d'autre a changé l'entité puisqu'elle était obtenu dans la transaction et une exception doit être levée. Le champ de version être mis à jour à la fois dans l'entité et dans la base de données chaque fois qu'une mise à jour de l'entité est envoyé à la base de données.

Les champs de version ne sont pas obligatoires, mais nous recommandons que les champs de version figurent dans chaque entité qui a une chance d'être modifiée simultanément par plus d'un processus.

Épisode 575

Une colonne de version est une nécessité absolue chaque fois qu'une entité est modifiée en tant que entité détachée et fusionnée à nouveau dans un contexte de persistance par la suite. Le plus long une entité reste en mémoire, plus il y a de chances qu'elle soit modifiée dans la base de données par un autre processus, rendant la copie en mémoire invalide. Les champs de version sont au cœur de verrouillage optimiste et offrent la protection la meilleure et la plus performante pour les

modification d'entité simultanée.

Les champs de version sont définis simplement en annotant le champ ou la propriété sur l'entité avec une annotation @Version. Référencement12-16 montre une entité Employee annotée pour avoir un champ de version.

Annonce 12-16. Utilisation d'un champ de version

```
@Entité
Employé de classe publique {
    @Id id int privé;
    @Version version int privée;
    // ...
}
```

Les champs de verrouillage de version définis sur l'entité peuvent être de type int, short, long, the les types de wrapper correspondants et java.sql.Timestamp. La pratique la plus courante est juste pour utiliser int ou l'un des types numériques, mais certaines bases de données héritées utilisent des horodatages.

Comme pour l'identifiant, l'application ne doit pas définir ou modifier le champ de version une seule fois l'entité a été créée. Il pourrait y accéder, cependant, pour ses propres besoins s'il le souhaite utilisez le numéro de version pour une raison dépendante de l'application.

> Astuce certains fournisseurs n'exigent pas que le champ de version soit défini et stocké dans l'entité. les variantes de stockage dans l'entité sont le stockage dans un fournisseur spécifique cache, ou ne stockant rien du tout mais à la place en utilisant la comparaison de champs. Pour Par exemple, une option courante consiste à comparer une combinaison de l'état de l'entité dans la base de données avec l'état de l'entité en cours d'écriture, puis utilisez le résultats comme critères pour décider si l'état a été changé.

Quelques mots d'avertissement concernant les champs de version s'imposent. Le premier est qu'ils ne sont pas garantis d'être mis à jour, ni dans les entités gérées ni dans la base de données, car partie d'une opération de mise à jour en bloc. Certains fournisseurs offrent une assistance pour la mise à jour automatique

Épisode 576

du champ de version pendant les mises à jour en bloc, mais cela ne peut pas être utilisé de manière portative. Pour les fournisseurs qui ne prennent pas en charge les mises à jour automatiques de version, la version d'entité peut être mis à jour manuellement dans le cadre de l'instruction UPDATE, comme illustré par la requête suivante:

```
MISE À JOUR Employé e
SET e.salary = e.salary + 1000, e.version = e.version + 1
O EXISTE (SELECT p
                DE e.projects p
                WHERE p.name = 'Release2')
```

Le deuxième point à retenir est que les champs de version seront automatiquement mis à jour uniquement lorsque les champs de non-relation ou la clé étrangère propriétaire champs de relation (par exemple, relations de clé étrangère source plusieurs-à-un et un-à-un) sont modifiés. Si vous souhaitez qu'une relation non détenue et valorisée par la collection entraîne une mise à jour vers la version de l'entité, vous devrez peut-être utiliser l'une des stratégies de verrouillage décrit dans les sections de verrouillage suivantes.

Modes de verrouillage optimiste avancés

Par défaut, JPA suppose ce qui est défini dans la spécification SQL ANSI / ISO et connu dans jargon d'isolation de transaction comme isolation Read Committed. Ce niveau d'isolement standard garantit simplement que les modifications apportées à l'intérieur d'une transaction ne seront pas visibles par d'autres transactions jusqu'à ce que la transaction modifiée ait été validée. Exécution normale

l'utilisation du verrouillage de version fonctionne avec l'isolement Read Committed pour fournir des données supplémentaires contrôles de cohérence face aux écritures entrelacées. Satisfaire des contraintes de verrouillage plus strictes que ce qu'offre ce verrouillage nécessite qu'une stratégie de verrouillage supplémentaire soit utilisée. Être portable, ces stratégies ne peuvent être utilisées que sur des entités avec des champs de version.

Les options de verrouillage peuvent être spécifiées au moyen d'un certain nombre d'appels différents:

- EntityManager.lock (): méthode explicite de verrouillage d'objets déjà dans un contexte de persistance.
- EntityManager.refresh (): permet de passer un mode de verrouillage et s'applique à l'objet dans le contexte de persistance en cours d'actualisation.
- EntityManager.find (): permet de passer un mode de verrouillage et s'applique à l'objet renvoyé.
- Query.setLockMode (): définit le mode de verrouillage à appliquer pendant exécution de la requête.

Épisode 577

Chacune des méthodes EntityManager doit être appelée dans une transaction. Bien que la méthode Query.setLockMode () peut être appelée à tout moment, une requête qui a son verrou Le mode set doit être exécuté dans le contexte d'une transaction.

Les méthodes lock () et refresh () sont appelées sur les objets déjà dans le contexte de persistance, donc selon l'implémentation particulière, il pourrait ou aucune action autre que le simple signalement des objets comme étant verrouillés.

Verrouillage de lecture optimiste

Le niveau suivant d'isolement des transactions est appelé lecture répétable et empêche le anomalie de lecture dite non répétable. Cette anomalie peut être décrite de différentes manières moyens, mais peut-être le plus simple est de dire que lorsqu'une transaction demande le même données deux fois dans la même transaction, la deuxième requête renvoie une version différente de données que celles renvoyées la première fois car une autre transaction les a modifiées dans le temps intermédiaire. En d'autres termes, le niveau d'isolement Lecture répétable signifie qu'une fois une transaction a accédé à des données et une autre transaction modifie ces données, au moins un des transactions doivent être empêchées de s'engager. Un verrou de lecture optimiste dans JPA fournit ce niveau d'isolement.

Pour verrouiller en lecture une entité de manière optimiste, un mode de verrouillage de LockModeType.OPTIMISTIC peut être passé à l'une des méthodes de verrouillage. Le verrou résultant garantira que les deux transaction qui obtient le verrou de lecture de l'entité et toute autre qui tente de modifier cette entité l'instance ne réussira pas les deux. Au moins un échouera, mais comme les niveaux d'isolement de la base de données, celui qui échoue dépend de la mise en œuvre.

> Astuce, la valeur LockModeType.OPTIMISTIC a été introduite dans Jpa 2.0 et est vraiment juste un changement de nom de l'option LockModeType.READ qui existait dans Jpa 1.0. bien que READ soit toujours une option valide, OPTIMISTIC doit être utilisé dans tous les nouveaux applications à venir.

La façon dont le verrouillage de lecture est implémenté dépend entièrement du fournisseur. Même si c'est ce qu'on appelle un verrou de lecture optimiste, un fournisseur peut choisir d'être sévère et obtenir un verrou d'écriture impatient sur l'entité, auquel cas toute autre transaction qui tente de modifier l'entité échouera ou se bloquera jusqu'à ce que la transaction de verrouillage soit terminée. Le fournisseur Cependant, le plus souvent, il verrouille l'objet de manière optimiste, ce qui signifie que le fournisseur n'ira pas réellement dans la base de données pour un verrou lorsque la méthode de verrouillage est appelée. Ce sera

Épisode 578

à la place, attendez la fin de la transaction, et au moment de la validation, il relira l'entité pour voir si l'entité a été modifiée depuis sa dernière lecture dans la transaction. Si ce n'est pas le cas changé, le verrou de lecture a été honoré, mais si l'entité a changé, le pari a été perdu et la transaction sera annulée.

Un corollaire de cette forme optimiste d'implémentation du verrouillage en lecture est qu'elle ne importe à quel point la méthode de verrouillage est effectivement invoquée pendant la transaction. Il peut être invoqué juste avant la validation, et les mêmes résultats seront être produit. La méthode ne fait que marquer l'entité pour qu'elle soit relue au moment de la validation. Peu importe quand, lors de la transaction, l'entité est ajoutée à cette liste car l'opération de lecture réelle ne se produira pas avant la fin de la transaction. Vous pouvez Pensez aux appels de lock () ou de verrouillage refresh () comme étant rétroactifs au point auquel l'entité a été lue dans la transaction au départ car c'est à ce moment-là la version est lue et enregistrée dans l'entité gérée.

Le cas par excellence de l'utilisation de ce type de verrou est lorsqu'une entité a un dépendance à une ou plusieurs autres entités pour la cohérence. Il y a souvent une relation entre les entités, mais pas toujours. Pour le démontrer, pensez à un département qui a des employés; nous voulons générer un rapport de salaire pour un ensemble donné de départements et avoir le rapport indique les dépenses salariales de chaque département. Nous avons une méthode appelée generateDepartmentsSalaryReport () qui itérera dans l'ensemble des départements et utilisez une méthode interne pour trouver le salaire total de chacun. La méthode par défaut d'avoir un attribut de transaction REQUIRED, de sorte qu'il sera exécuté entièrement dans le contexte d'une transaction. Le code est dans l'extrait 12-17.

Annonce 12-17. Rapport sur les salaires du département

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    // ...

    public SalaryReport generateDepartmentsSalaryReport (
                                                    List <Integer> deptIds) {
        Rapport SalaryReport = nouveau SalaryReport ();
        total long = 0;
```


Épisode 579


```
        for (Integer deptId: deptIds) {
            long deptTotal = totalSalaryInDepartment (deptId);
            report.addDeptSalaryLine (deptId, deptTotal);
            total + = deptTotal;
        }
        report.addSummarySalaryLine (total);
        rapport de retour;
    }

    protected long totalSalaryInDepartment (int deptId) {
```

```
        total long = 0;
        Département dept = em.find (Department.class, deptId);
        pour (Employé emp: dept.getEmployees ())
            total + = emp.getSalary ();
        retour total;
    }

    public void changeEmployeeDepartment (int deptId, int empId) {
        Employé emp = em.find (Employee.class, empId);
        emp.getDepartment (). removeEmployee (emp);
        Département dept = em.find (Department.class, deptId);
        dept.addEmployee (emp);
        emp.setDepartment (dept);
    }
    // ...
}
```

Le rapport sera généré facilement, mais est-il correct? Que se passe-t-il si un employé est déplacé d'un département à un autre pendant le temps où nous calculons le total un salaire? Par exemple, disons que nous demandons un rapport sur les ministères 10, 11 et 12. La requête commence à générer le rapport pour le département 10. Il termine le département 10 et passe au service 11. Comme il parcourt tous les employés de département 11, l'employé avec l'ID 50 dans le département 10 est remplacé par department 12. Quelque part, un responsable appelle le changeEmployeeDepartment () méthode, la transaction est validée et l'employé 50 est changé pour être dans le département 12. Pendant ce temps, le générateur de rapports a terminé le département 11 et passe maintenant à générer un total de salaire pour le département 12. Lorsqu'il parcourt les employés, il

Épisode 580

trouver l'employé 50 même s'il a déjà compté cet employé dans le département 10, donc l'employé 50 sera compté deux fois. Nous avons tout fait dans les transactions mais nous avons quand même un vue incohérente des données des employés. Pourquoi?

Le problème résidait dans le fait que nous n'avons verrouillé aucun des objets employés de en cours de modification lors de notre opération. Nous avons émis plusieurs requêtes et étions vulnérables pour afficher le même objet avec un état différent, qui est la lecture non répétable phénomène. Nous pourrions le résoudre de plusieurs manières, dont l'une serait de définir le isolation de la base de données en lecture répétable. Parce que nous expliquons la méthode lock (), nous allons l'utiliser pour verrouiller chacun des employés afin qu'ils ne puissent pas non plus changer pendant notre transaction était active, ou si c'était le cas, notre transaction échouerait. Référencement12-18 spectacles la méthode mise à jour qui effectue le verrouillage.

Annonce 12-18. Utilisation d'un verrou de lecture optimiste

```
protected long totalSalaryInDepartment (int deptId) {
    total long = 0;
    Département dept = em.find (Department.class, deptId);
    for (Employé emp: dept.getEmployees ()) {
        em.lock (emp, LockModeType.OPTIMISTIC);
        total + = emp.getSalary ();
    }
    retour total;
}
```

Nous avons mentionné que la mise en œuvre est autorisée à se verrouiller avec empressement ou à différer acquisition des verrous jusqu'à la fin de la transaction. La plupart des implémentations majeures

reporter le verrouillage jusqu'au moment de la validation et, ce faisant, offrir des performances bien supérieures et l'évolutivité sans sacrifier aucune des sémantiques.

Verrouillage d'écriture optimiste

Un deuxième niveau de verrouillage optimiste avancé est appelé verrou d'écriture optimiste, qui par la vertu de son nom indique correctement que nous sommes en train de verrouiller l'objet pour l'écriture. le le verrou d'écriture garantit tout ce que fait le verrou de lecture optimiste, mais s'engage également à incrémenter le champ de version dans la transaction, qu'un utilisateur ait ou non mis à jour l'entité.
Cela promet un échec optimiste du verrouillage si une autre transaction tente également de modifier la même entité avant que celle-ci ne s'engage. Cela équivaut à faire un

Épisode 581

mise à jour de l'entité afin de déclencher l'augmentation du numéro de version, et c'est pourquoi l'option s'appelle OPTIMISTIC_FORCE_INCREMENT. La conclusion évidente est que si le l'entité est mise à jour ou supprimée par l'application, elle n'a jamais besoin d'être explicitement verrouillé en écriture, et que le verrouillage en écriture serait de toute façon redondant au mieux et au pire pourrait conduire à une mise à jour supplémentaire, en fonction de la mise en œuvre.

> Conseil que la valeur LockModeType.OPTIMISTIC_FORCE_INCREMENT était introduit dans Jpa 2.0 et n'est en réalité qu'un changement de nom de LockModeType.WRITE option qui existait dans Jpa 1.0. bien que WRITE soit toujours une option valide, OPTIMISTIC_ FORCE_INCREMENT doit être utilisé dans toutes les nouvelles applications à venir.

Rappelez-vous que les mises à jour de la colonne version ne se produisent normalement pas lorsque des modifications sont fait à une relation non possédée. Pour cette raison, le cas courant d'utilisation d'OPTIMISTIC_ FORCE_INCREMENT est de garantir la cohérence à travers les changements de relation d'entité (souvent ce sont des relations un-à-plusieurs avec des clés étrangères cibles) dans le modèle objet les pointeurs de relation d'entité changent, mais dans le modèle de données aucune colonne dans l'entité changement de table.

Par exemple, disons qu'un employé a un ensemble d'uniformes qui lui ont été attribués à lui, et son entreprise dispose d'un service de nettoyage bon marché qui le facture automatiquement par retenues salariales. L'employé a donc une relation un-à-plusieurs avec Uniform, et L'employé a un champ cleaningCost qui contient le montant qui sera déduit de son chèque de paie à la fin du mois. S'il existe deux beans session avec état différents ont des contextes de persistance étendus, un pour la gestion des employés (EmployeeManagement) et un autre qui gère les frais de nettoyage (CleaningFeeManagement) pour l'entreprise, alors si l'Employé existe dans les deux contextes de persistance, il y a une possibilité de incohérence.

Les deux copies de l'entité Employee commencent de la même manière, mais disons qu'un opérateur enregistre que l'employé a reçu un tout nouvel uniforme supplémentaire. Cela implique création d'une nouvelle entité Uniform et l'ajout à la collection un-à-plusieurs du Employé. La transaction est validée et tout va bien, sauf que maintenant le Le contexte de persistance EmployeeManagement a une version différente de Employee que le contexte de persistance CleaningFeeManagement a. L'opérateur a fait le premier tâche de maintenance et passe maintenant au calcul des frais de nettoyage pour les clients. le La session CleaningFeeManagement calcule les frais de nettoyage en fonction du un-à-plusieurs

Épisode 582

relation qu'il connaît (sans l'uniforme supplémentaire) et écrit un nouveau
version de l'employé avec les frais de nettoyage de l'employé basés sur un uniforme en moins.
La transaction est validée même si la première transaction avait déjà
commis et bien que les modifications de la relation uniforme aient déjà été engagées
à la base de données. Maintenant, nous avons une incohérence entre le nombre d'uniformes et
le coût de leur nettoyage et le contexte de persistance de CleaningFeeManagement
avec sa copie périmée de l'employé sans même connaître le nouvel uniforme et
ne jamais avoir de conflit de verrouillage.

La raison pour laquelle le changement n'a pas été vu et aucune exception de verrouillage ne s'est produite pour le
la deuxième opération était parce que dans la première opération aucune écriture sur l'employé en fait
s'est produite et la colonne de version n'a donc pas été mise à jour. Les seuls changements apportés au
L'employé était à sa relation, et parce qu'il appartenait au côté uniforme là-bas
n'avait aucune raison de faire des mises à jour à l'employé. Malheureusement pour l'entreprise
(mais pas pour l'employé), cela signifie qu'ils paieront des frais de nettoyage pour l'uniforme.

La solution consiste à utiliser l'option OPTIMISTIC_FORCE_INCREMENT, comme indiqué dans
Référencement 12-19 , et forcer une mise à jour de l'employé lorsque la relation a changé dans le
première opération. Cela entraînera l'échec de toutes les mises à jour dans tout autre contexte de persistance si elles
apporter des modifications sans connaître la mise à jour de la relation.

Annonce 12-19. Utilisation d'un verrou d'écriture optimiste

```
@Stateful
public class EmployeeManagement {
    @PersistenceContext (unitName = "EmployeeService",
                         type = PersistenceContextType.EXTENDED)
    EntityManager em;

    public void addUniform (int id, Uniform uniform) {
        Employé emp = em.find (Employee.class, id);
        em.lock (emp, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
        emp.addUniform (uniforme);
        uniform.setEmployee (emp);
    }

    // ...
}
```

Épisode 583


```
@Stateful
public class CleaningFeeManagement {
    Float final statique UNIFORM_COST = 4.7f;

    @PersistenceContext (unitName = "EmployeeService",
                         type = PersistenceContextType.EXTENDED)
    EntityManager em;

    public void CalculateCleaningCost (int id) {
        Employé emp = em.find (Employee.class, id);
        Coût flottant = emp.getUniforms (). Size () * UNIFORM_COST;
        emp.setCost (emp.getCost () + coût);
    }
```

```
    // ...
}
```

Récupération des échecs optimistes

Un échec optimiste signifie qu'une ou plusieurs des entités modifiées ont été
pas assez frais pour pouvoir enregistrer leurs modifications. La version de l'entité qui
a été modifiée était obsolète, et l'entité avait depuis été modifiée dans la base de données, d'où
une OptimisticLockException a été lancée. Il n'y a pas toujours de solution simple pour
récupération, et selon l'architecture de l'application, il peut ou non être
possible, mais si et quand cela est approprié, une solution pourrait être d'obtenir une nouvelle copie du
entité, puis réappliquez les modifications. Dans d'autres cas, il ne sera possible que de donner
le client (tel qu'un navigateur Web) une indication que les modifications étaient en conflit avec
une autre transaction et doit être réintroduite. La dure réalité est que dans la majorité des
cas, il n'est ni pratique ni faisable de traiter les problèmes de verrouillage optimiste autrement que
réessayez simplement l'opération à un point de démarcation transactionnel pratique.

Le premier problème que vous pourriez rencontrer lorsqu'une exception OptimisticLockException est
jeté pourrait être celui que vous ne voyez jamais. En fonction de vos paramètres, par exemple
si le bean appelant est géré par conteneur ou géré par bean, et si le
L'interface est distante ou locale, vous pouvez uniquement obtenir une exception EJBException lancée par le conteneur. Cette
exception n'entraînera même pas nécessairement l'OptimisticLockException car tout cela
est formellement requis du conteneur est de le consigner avant de lever l'exception.

Référencement 12-20 montre comment cela peut se produire lors de l'appel d'une méthode sur une session
bean qui lance une nouvelle transaction.

Épisode 584

Annonce 12-20. Client BMT Session Bean

```
@Apatride
@TransactionManagement (TransactionManagementType.BEAN)
classe publique EmpServiceClient {
    @EJB EmployeeService empService;

    public void AdjustVacation (int id, int days) {
        essayez {
            empService.deductEmployeeVacation (id, jours);
        } catch (EJBException ejbEx) {
            System.out.println (
                "Une erreur s'est produite, mais je n'ai aucune idée de quoi!");
        } catch (OptimisticLockException olEx) {
            System.out.println (
                "Cette exception serait bien, mais je le ferai" +
                                                "probablement jamais compris!");
        }
    }
}
```

Le problème est que lorsqu'une exception optimiste se produit dans les entrailles du
persistance, il sera renvoyé au bean de session EmployeeService et obtiendra
géré selon les règles de gestion des exceptions d'exécution par le conteneur. Parce que
EmpServiceClient utilise des transactions gérées par bean et ne démarre pas de transaction,
et EmployeeService utilise par défaut des transactions gérées par conteneur avec un REQUIS
attribut, une transaction sera initiée lorsque l'appel à deductVacationBalance ()
se produit.

Une fois la méthode terminée et les modifications apportées, le
container tentera de valider la transaction. Dans ce processus, le

le fournisseur de persistance recevra une notification de synchronisation de transaction du gestionnaire de transactions pour vider son contexte de persistance dans la base de données. En tant que fournisseur tente ses écritures, il trouve lors de son numéro de version vérifier que l'un des objets a été modifié par un autre processus depuis sa lecture par celui-ci, il lance donc un OptimisticLockException. Le problème est que le conteneur traite cette exception de la même manière que toute autre exception d'exécution. L'exception est simplement enregistrée et le container lève une EJBException.

Épisode 585

La solution à ce problème consiste à effectuer une opération flush () depuis l'intérieur du transaction gérée par conteneur pour le moment juste avant que nous soyons prêts à terminer la méthode. Cela force une écriture dans la base de données et verrouille les ressources uniquement à la fin de la méthode afin de minimiser les effets sur la concurrence. Cela nous permet également de gérer un échec optimiste pendant que nous sommes en contrôle, sans que le conteneur n'interfère et avaler l'exception. Si nous obtenons une exception de l'appel flush (), nous pouvons lancer une exception d'application que l'appelant peut reconnaître. Ceci est montré dans la liste12-21.

Annonce 12-21. Capture et conversion d'OptimisticLockException

```
@Apatride
public class EmployeeService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    public void deductEmployeeVacation (int id, int days) {
        Employé emp = em.find (Employee.class, id);
        emp.setVacationDays (emp.getVacationDays () - jours);
        // ...
        flushChanges ();
    }

    public void AdjustEmployeeSalary (int id, long salaire) {
        Employé emp = em.find (Employee.class, id);
        emp.setSalary (salaire);
        // ...
        flushChanges ();
    }

    protected void flushChanges () {
        essayez {
            em.flush ();
        } catch (OptimisticLockException optLockEx) {
            throw new ChangeCollisionException ();
        }
    }
    // ...
}
```


Épisode 586

```
@ApplicationException
Public class ChangeCollisionException étend RuntimeException {
    public ChangeCollisionException () {super (); }
}
```

L'OptimisticLockException peut contenir l'objet qui a provoqué l'exception, mais ce n'est pas garanti. Dans l'extrait 12-21 , il n'y a qu'un seul objet dans la transaction (l'employé), nous savons donc que c'est celui qui a causé l'échec. S'il y avait plusieurs objets dans la transaction, nous aurions pu appeler getEntity () sur le exception pour voir si l'objet incriminé a été inclus.

Nous excluons le vidage du reste du code de traitement car chaque La méthode doit vider et intercepter l'exception, puis relancer un domaine spécifique exception d'application. La classe ChangeCollisionException est annotée avec @ApplicationException, qui est une annotation de conteneur EJB dans le javax.ejb package, pour indiquer au conteneur que l'exception n'est pas vraiment un niveau système exception mais doit être renvoyée au client tel quel. Normalement, définir un une exception d'application empêchera le conteneur d'annuler la transaction, mais c'est une notion de conteneur EJB. Le fournisseur de persistance qui a lancé le OptimisticLockException ne connaît pas la sémantique spéciale des exceptions d'application et, voyant une exception d'exécution, ira de l'avant et marquera le transaction pour la restauration.

Le code client que vous avez vu précédemment peut désormais recevoir et gérer l'application exception et éventuellement faire quelque chose à ce sujet. À tout le moins, il est conscient du fait que l'échec était le résultat d'une collision de données au lieu d'une autre erreur plus fatale. le Le bean client est affiché dans le Listing 12-22.

Annonce 12-22. Gestion de l'exception OptimisticLockException

```
@Apatride
@TransactionManagement (TransactionManagementType.BEAN)
classe publique EmpServiceClient {
    @EJB EmployeeService empService;

    public void AdjustVacation (int id, int days) {
        essayez {
            empService.deductEmployeeVacation (id, jours);
        } catch (ChangeCollisionException ccEx) {
```



```
                System.out.println (
                        "Collision avec un autre changement - Réessayer ...");
                empService.deductEmployeeVacation (id, jours);
            }
        }
}
```

Lorsqu'une exception OptimisticLockException se produit dans ce contexte, la réponse simple est de recommencez. C'était vraiment un cas assez trivial, donc la décision de réessayer n'était pas difficile à prendre. Si nous sommes dans un contexte de persistance prolongée, cependant, nous pourrions avoir un travail beaucoup plus difficile de parce que toutes les entités du contexte de persistance étendue se détachent lorsqu'une la transaction est annulée. Essentiellement, nous aurions besoin de réenregistrer tous nos objets après avoir les relire puis rejouer toutes les modifications que nous avions appliquées dans le précédent échouent transaction. Ce n'est pas une chose très facile à faire dans la plupart des cas.

En général, il est assez difficile de coder pour le cas d'exception optimiste. Quand s'exécutant dans un environnement de serveur, il est probable que toute exception OptimisticLockException être encapsulé par une exception de niveau composant ou une exception de serveur. La meilleure approche est de

il suffit de traiter toutes les échecs de transaction de la même manière et de réessayer la transaction depuis le début
ou pour indiquer au navigateur client qu'il doit redémarrer et réessayer.

Verrouillage pessimiste

Le verrouillage pessimiste implique l'obtention d'un verrou sur un ou plusieurs objets immédiatement,
au lieu d'attendre avec optimisme la phase de validation et d'espérer que les données ont
n'a pas été modifié dans la base de données depuis sa dernière lecture. Un verrou pessimiste est synchrone
en ce qu'au moment où l'appel de verrouillage revient, il est garanti que l'objet verrouillé
ne pas être modifié par une autre transaction avant la fin de la transaction en cours
et libère son verrou. Cela ne laisse pas la fenêtre ouverte pour l'échec de la transaction en raison de
changements simultanés, une possibilité très réelle lorsque le verrouillage optimiste simple est utilisé.

Il était probablement évident dans la section précédente que la gestion du verrouillage optimiste
les exceptions ne sont pas toujours une question simple. C'est probablement l'une des raisons pour lesquelles de nombreux
les développeurs ont tendance à utiliser le verrouillage pessimiste car il est toujours plus facile d'écrire votre
logique d'application lorsque vous savez à l'avance si votre mise à jour réussira ou non.

En réalité, cependant, ils limitent souvent l'évolutivité de leurs applications
car le verrouillage inutile sérialise de nombreuses opérations qui pourraient facilement se produire en parallèle.
La réalité est que très peu d'applications nécessitent un verrouillage pessimiste, et celles

Épisode 588

qui n'en ont besoin que pour un sous-ensemble limité de requêtes. La règle est que si vous pensez avoir besoin
verrouillage pessimiste, détrompez-vous. Si vous êtes dans une situation où vous avez un
degré de concurrence en écriture sur le (s) même (s) objet (s) et l'occurrence d'optimiste
les échecs sont élevés, vous pourriez avoir besoin d'un verrouillage pessimiste car le coût des tentatives peut
devenir si coûteux que vous feriez mieux de verrouiller de manière pessimiste. Si vous
ne peut absolument pas réessayer vos transactions et êtes prêt à sacrifier une certaine quantité de
évolutivité, cela pourrait également vous conduire à utiliser un verrouillage pessimiste.

Modes de verrouillage pessimistes

En supposant que votre application relève du petit pourcentage d'applications
qui devraient acquérir des verrous pessimistes, vous pouvez verrouiller les entités de manière pessimiste en utilisant le
mêmes méthodes API décrites dans la section «Modes de verrouillage optimiste avancés». Comme
les modes optimistes, les modes de verrouillage pessimistes garantissent également la lecture répétable
isolement, ils le font simplement avec pessimisme. De même, une transaction doit être active pour
d'acquérir un verrou pessimiste.

Il existe trois modes de verrouillage pessimiste pris en charge, mais le plus courant est de loin
verrouillage d'écriture pessimiste, nous allons donc en discuter en premier.

Verrouillage d'écriture pessimiste

Lorsqu'un développeur décide qu'il souhaite utiliser un verrouillage pessimiste, il pense généralement
sur le type de verrouillage offert par le mode PESSIMISTIC_WRITE. Ce mode
sera traduit par la plupart des fournisseurs en une instruction SQL SELECT FOR UPDATE dans le
base de données, obtenant un verrou en écriture sur l'entité afin qu'aucune autre application ne puisse la modifier.
Référencement 12-23 montre un exemple d'utilisation de la méthode lock () avec PESSIMISTIC_WRITE
mode. Il montre un processus qui s'exécute tous les jours et accumule le montant des vacances pour chaque
employé.

Annonce 12-23. Verrouillage d'écriture pessimiste

```
@Apatride
public class VacationAccrualService {
    @PersistenceContext (unitName = "Employé")
```

EntityManager em;

Épisode 589


```
    public void accumulEmployeeVacation (int id) {
        Employé emp = em.find (Employee.class, id);
        // Trouver amt selon les règles syndicales et le statut emp
        Statut EmployeeStatus = emp.getStatus ();
        double accumedDays = CalculateAccrual (statut);
        si (jours courus)> 0 {
            em.lock (emp, LockModeType.PESSIMISTIC_WRITE);
            emp.setVacationDays (emp.getVacationDays () + accumuledDays);
        }
    }
}
```

Le bean session utilise des transactions gérées par le conteneur, donc une nouvelle transaction est démarré lorsque la méthode augmenteEmployeeVacation () est appelée. Le verrou pessimiste est dans la transaction de conteneur et ne se produit que s'il y a quelque chose à ajouter, nous semblent ne pas acquérir le verrou inutilement. En surface, tout semble être intelligent et correct. Cependant, ce n'est pas le cas et présente des défauts auxquels il faut remédier avant tout processus qui appelle ce code doit être démarré.

Ignorer la possibilité que l'employé n'existe pas (et la méthode find () pourrait renvoyer null), le problème le plus sérieux est que le code suppose le caractère pessimiste lock est rétroactif au moment où l'employé a été lu. Verrouillage à la dernière minute dans l'ordre minimiser le temps de maintien du verrou exclusif est la bonne idée, mais les données des employés qui est verrouillé dans la base de données peut ne pas être en fait le même que l'état dans lequel nous regardons. Le problème est enraciné dans le fait que nous lisons l'employé au début de la méthode, mais l'a verrouillée beaucoup plus tard, laissant la fenêtre ouverte pour une autre processus pour changer l'employé. Pendant ce temps, nous utilisons l'état de l'employé qui nous l'avons d'abord lu et modifié. Si nous n'avons pas de champ de version sur l'entité Employé, le changement effectué par un autre processus serait remplacé par notre copie périmée, même bien que nous ayons utilisé un verrou pessimiste.

Si nous avons un champ de version, le contrôle de verrouillage optimiste qui se produit toujours même lorsque le verrouillage pessimiste est utilisé, il attraperait la version périmée, et nous obtiendrions un OptimisticLockException. Cette exception intercepterait le code du Listing 12-23 par surprise car aucune manipulation n'est en place. De plus, nous aurions pu utiliser un verrouillage pessimiste car nous ne voulions pas être surpris au moment de l'engagement et avoir à gérer des problèmes si tard dans la transaction.

Épisode 590

La solution est soit d'acquérir le verrou sur l'employé à l'avant dans le find () méthode (et risquez les implications d'évolutivité) ou effectuez une actualisation de verrouillage (). Référencement12-24 montre la version actualisée améliorée de la méthode accumulEmployeeVacation ().

Annonce 12-24. Verrouillage pessimiste avec rafraîchissement

```
public void accumulEmployeeVacation (int id) {
    Employé emp = em.find (Employee.class, id);
    // Trouver amt selon les règles syndicales et le statut emp
    Statut EmployeeStatus = emp.getStatus ();
    double accumedDays = CalculateAccrual (statut);
    if (accumuledDays> 0) {
        em.refresh (emp, LockModeType.PESSIMISTIC_WRITE);
        if (statut! = emp.getStatus ())
            accumuledDays = CalculateAccrual (emp.getStatus ());
        si (jours courus> 0)
            emp.setVacationDays (emp.getVacationDays () + accumuledDays);
    }
}
```

Lorsque nous faisons un rafraîchissement, il se peut que l'employé déclare sur lequel notre les calculs dépendaient initialement a changé depuis. Pour nous assurer que nous ne nous retrouvons pas avec un employé incohérent, nous faisons une dernière vérification du statut de l'employé. S'il a changé, nous recalculons en utilisant le nouveau statut et faisons enfin la mise à jour.

Verrouillage de lecture pessimiste

Certaines bases de données prennent en charge des mécanismes de verrouillage pour obtenir une isolation de lecture répétable sans l'acquisition d'un verrou en écriture. Un mode PESSIMISTIC_READ peut être utilisé pour atteindre de manière pessimiste sémantique de lecture répétable lorsqu'aucune écriture dans l'entité n'est attendue. Le fait que ce type de situation ne sera pas rencontré très souvent, combiné avec l'allocation qui les fournisseurs ont de l'implémenter en utilisant un verrou d'écriture pessimiste, nous amène à dire que ce le mode n'est pas facile à saisir et couramment utilisé.

Lorsqu'une entité verrouillée avec un verrou de lecture pessimiste finit par être modifiée, le verrou sera mis à niveau vers un verrou d'écriture pessimiste. Cependant, la mise à niveau peut ne pas se produit jusqu'à ce que l'entité soit purgée, il est donc peu efficace car une acquisition de verrou a échoué l'exception ne sera pas lancée avant l'heure de validation de la transaction, ce qui rend le verrou équivalent à un optimiste.

Épisode 591

Verrouillage incrémentiel forcé pessimiste

Un autre mode qui cible le cas de l'acquisition de verrous pessimistes, même si l'entité est seulement en cours de lecture, est le mode PESSIMISTIC_FORCE_INCREMENT. Comme l'OPTIMISTIC_ FORCE_INCREMENT, ce mode incrémentera également le champ de version de l'entité verrouillée peu importe si des modifications y ont été apportées. C'est un cas quelque peu chevauchant avec Verrouillage de lecture pessimiste et verrouillage d'écriture optimiste, par exemple, lorsque non possédé les relations à valeur de collection sont présentes dans l'entité et ont été modifiées. Forcer le champ de version à incrémenter peut conserver un certain degré de cohérence de version à travers les relations.

Portée pessimiste

La section "Gestion des versions" mentionnait que les modifications apportées à toute relation détenue provoque la mise à jour du champ de version de l'entité propriétaire. Si un unidirectionnel un-à- de nombreuses relations devaient changer, par exemple, la version serait mise à jour même bien qu'aucune modification de la table d'entité n'aurait autrement été affirmée.

En ce qui concerne le verrouillage pessimiste, l'acquisition de verrous exclusifs sur des entités d'autres tables d'entités peuvent augmenter la probabilité de blocage. Pour éviter cela, le comportement par défaut des requêtes à verrouillage pessimiste est de ne pas acquérir de verrous sur les tables qui ne sont pas mappés à l'entité. Une propriété supplémentaire existe pour activer ce comportement dans

cas où quelqu'un a besoin d'acquérir les verrous dans le cadre d'une requête pessimiste. Le javax.
La propriété persistence.lock.scope peut être définie sur la requête en tant que propriété, avec sa valeur
défini sur PessimisticLockScope.EXTENDED. Lorsqu'elle est définie, cible les tables de
relations, tables de collection d'éléments et jointure de relation plusieurs-à-plusieurs appartenant
les tables verront toutes leurs lignes correspondantes verrouillées de manière pessimiste.

Cette propriété doit normalement être évitée, sauf lorsqu'il est absolument nécessaire de
verrouiller ces tables comme des tables de jointure qui ne peuvent pas être verrouillées d'une autre manière. Strict
l'ordre et une solide compréhension des mappages et de l'ordre des opérations devraient être un
condition préalable à l'activation de cette propriété pour garantir qu'il n'y a pas de blocage.

Timeouts pessimistes

Jusqu'à présent, nous n'avons fait aucune mention des délais d'attente ou de la façon de spécifier combien de temps attendre
serrures. Bien que JPA ne décrive pas de manière normative comment les fournisseurs doivent prendre en charge le délai d'expiration
modes d'acquisition de verrouillage pessimiste, JPA définit un indice que les fournisseurs peuvent utiliser.
Bien que ce ne soit pas obligatoire, l'indicateur javax.persistence.lock.timeout est probablement pris en charge

Épisode 592

par les principaux fournisseurs JPA; cependant, assurez-vous que votre fournisseur le prend en charge avant
codant à cet indice. Sa valeur peut être soit 0, ce qui signifie ne pas bloquer l'attente du verrou, soit
un entier décrivant le nombre de millisecondes à attendre le verrou. Ça peut être passé
dans l'une des méthodes d'API EntityManager qui acceptent à la fois un mode de verrouillage et une carte de
propriétés ou conseils:

```
Map <String, Object> props = new HashMap <String, Object> ();
props.put ("javax.persistence.lock.timeout", 5000);
em.find (Employee.class, 42, LockModeType.PESSIMISTIC_WRITE, accessoires);
```

Il peut également être défini sur une requête comme indice:

```
TypedQuery <Employee> q = em.createQuery (
            "SELECT e FROM EMPLOYEE e WHERE e.id = 42",
            Classe.employé);
q.setLockMode (LockModeType.PESSIMISTIC_WRITE);
q.setHint ("javax.persistence.lock.timeout", 5000);
```

Malheureusement, il n'y a pas de comportement par défaut lorsque l'indicateur de délai d'attente n'est pas spécifié.
Entre le fournisseur et la base de données, cela peut bloquer, cela peut être «pas d'attente» ou cela peut
ont un délai d'expiration par défaut.

Se remettre d'échecs pessimistes

Le dernier sujet à discuter autour du verrouillage pessimiste est ce qui se passe lorsque le verrou
ne peut pas être acquis. Nous n'avons placé aucun code de gestion des exceptions autour de notre
exemples, mais les appels de verrouillage pessimistes peuvent évidemment échouer pour de nombreuses raisons.

Lorsqu'un échec se produit suite à l'impossibilité d'acquérir un verrou pendant une requête,
ou pour toute raison considérée par la base de données comme non fatale à la transaction,
une LockTimeoutException sera lancée, et l'appelant peut l'attraper et réessayer simplement
appelez s'il le souhaite. Cependant, si l'échec est suffisamment grave pour provoquer une transaction
échec, une exception PessimisticLockException sera lancée et la transaction sera marquée
pour la restauration. Lorsque cette exception se produit, certaines des idées de la section «Récupération
from Optimistic Failures »peut être utile car la transaction est vouée à l'échec
et nous semblerions être dans le même bateau ici. La principale différence, cependant, est qu'un
PessimisticLockException se produit à la suite d'un appel de méthode, et non comme un échec différé
pendant la phase de validation. Nous avons plus de contrôle et pourrions attraper l'exception et convertir
à un autre plus significatif avant de le renvoyer à l'initiateur de démarcation de transaction.

Épisode 593

Mise en cache

La mise en cache est un terme assez large qui implique généralement de sauvegarder quelque chose en mémoire pour accès plus rapide par la suite. Même dans un contexte JPA, la mise en cache peut signifier des choses assez différentes à différentes personnes, selon la perspective. Dans cette section, nous parlons de la mise en cache des entités ou l'état qui constitue une entité.

Trier les couches

S'il y a une chose que nous, les types de logiciels, aimons faire, c'est de décomposer les choses en couches. Nous faisons cela parce que diviser un système complexe en plusieurs éléments cohésifs aide nous pour comprendre et communiquer plus facilement les aspects du système. Parce que ça marche plutôt bien, et nous ne sommes pas du genre à ignorer une bonne chose, nous partitionnerons de la même manière le JPA l'architecture en couches pour illustrer différentes opportunités de cache. Figure 12-1 donne un vue illustrée des différentes couches de mise en cache qui pourraient exister.

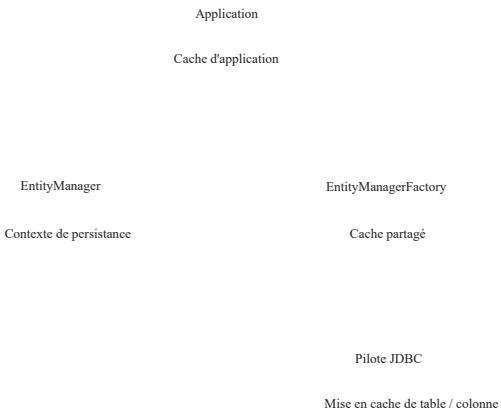

Figure 12-1. *Couches de mise en cache dans JPA*

La première couche que nous rencontrons est en fait au niveau de l'application. Toute application peut être écrit pour mettre en cache autant d'entités qu'il le souhaite simplement en conservant des références pour eux. Cela devrait être fait en prenant conscience que les entités vont probablement

Épisode 594

se détachent à un moment donné, et plus ils restent longtemps dans l'espace d'application, plus la probabilité qu'ils deviennent périmés. Les caches d'applications ont leur place, mais sont généralement déconseillé car les instances d'entités mises en cache ne seront jamais incluses dans aucun futurs résultats de requêtes JPA ou contextes de persistance.

Ensuite, le contexte de persistance référencé par un gestionnaire d'entités peut être considéré comme un cache car il conserve les références à toutes les entités gérées. Si, comme cela se fait dans le matériel architectures, nous catégorisons les différentes couches de mise en cache en niveaux, nous appellerions le contexte de persistance le premier niveau réel de la mise en cache JPA car c'est le premier endroit où

un fournisseur de persistance pourrait récupérer une entité en mémoire à partir de. Lors de l'exécution dans un gestionnaire d'entités à portée de transaction qui a un contexte de persistance délimité par le limites de transaction, le contexte de persistance peut être qualifié de cache transactionnel car il n'est disponible que pour la durée de la transaction. Lorsque le gestionnaire d'entités est une extension, son cache de contexte de persistance a une durée de vie plus longue et ne disparaîtra que lorsque le gestionnaire d'entités est effacé ou fermé.

L'exécution d'une recherche () ou d'une méthode de requête peut être considérée comme le chargement d'un ou plusieurs les entités dans le cache, tout en invoquant detach () sur une entité peut à certains égards considéré comme une éviction de cache de contexte de persistance de cette entité. La différence est que si il y a des changements d'état en attente dans cette entité, ils seront perdus à moins d'une opération de vidage se produit avant que l'entité ne soit détachée.

La mise en cache dans la fabrique de gestionnaires d'entités est appelée par certains le deuxième niveau cache, mais bien sûr, ce nom n'a de sens que s'il n'y a pas de couches de mise en cache entre et le contexte de persistance, ce qui n'est pas le cas pour tous les fournisseurs. Une chose qui est assez répandue parmi tous les fournisseurs est que les données d'entité dans ce cache sont partagées entre les gestionnaires d'entités de l'usine qui contient le cache, donc un meilleur nom est un cache partagé. Ce cache a une API spécifique qui lui est associée et est discuté dans son propre section.

Le dernier cache qui peut contribuer à l'état de l'entité à JPA est le cache du pilote JDBC. Plus les connexions et les instructions du cache des pilotes Certains caches gardent également une trace de la table ou état de colonne qui est essentiellement transparent pour le fournisseur JPA, mais qui peut néanmoins offrent des économies en évitant d'avoir à consulter la base de données pour obtenir des données à chaque appel. Cela n'est généralement possible dans le pilote que si l'on sait que les données sont en lecture seule ou le pilote contrôle exclusivement l'accès à la base de données.

Épisode 595

Notez certains fournisseurs, tels que l'implémentation de référence Jpa eclipseLink, fournissent des couches et des fonctionnalités de mise en cache beaucoup plus sophistiquées et exotiques, telles que mise en cache isolée pour la prise en charge de la base de données privée virtuelle (vpd), options affinées pour déterminer les politiques d'éviction automatique des entités et la coordination du cache distribué mécanismes. Beaucoup offrent également des intégrations avec de luxe et hautement spécialisés produits de mise en cache distribués.

Pour voir comment les différents niveaux de mise en cache du système sont accessibles pendant le cours d'une opération typique, traçons une requête find () pour l'employé avec l'ID 100:

```
Employé emp = em.find (Employee.class, 100);
```

La première chose qui se produit est que le client recherche dans son cache local l'employé et constate qu'il n'a pas cette instance Employee avec l'ID 100. Il émet alors le find () appelle le gestionnaire d'entités. Le gestionnaire d'entités aura probablement une persistance contexte qui lui est associé, il vérifie donc dans son contexte de persistance l'entité de type Employé avec l'ID 100. Si l'entité existe dans le contexte de persistance, le l'instance est renvoyée. S'il n'existe pas ou si aucun contexte de persistance n'a encore été associé avec le gestionnaire d'entités, le gestionnaire d'entités se rend en usine pour voir si le cache partagé a l'instance d'entité. Si tel est le cas, une nouvelle instance Employee avec l'ID 100 est créée à partir de celui partagé et inséré dans le contexte de persistance, et la nouvelle instance gérée est renvoyé à l'appelant. S'il ne se trouve pas dans le cache partagé, une requête SQL est générée pour sélectionnez l'entité dans la base de données. Le pilote JDBC peut avoir des données en cache, donc

il pourrait court-circuiter la clause select et renvoyer au moins une partie des données nécessaires. le
les données de requête résultantes sont ensuite composées en un objet et renvoyées. Cet objet est
inséré dans le cache d'usine du gestionnaire d'entités partagé, et une nouvelle copie d'instance de celui-ci est
créé et inséré dans le contexte de persistance à gérer. Cette instance d'entité est
finalement retourné à l'application cliente pour que le client le cache, s'il le souhaite.

Cache partagé

Dans les premiers jours de JPA 1.0, lorsque les gens demandaient la standardisation de la mise en cache partagée
au niveau de l'usine du gestionnaire d'entités, nous avons généralement pensé que cela n'en valait pas la peine car
chaque fournisseur semblait faire la mise en cache différemment. Certains fournisseurs mettent en cache les données JDBC brutes,
d'autres mettent en cache des objets entiers, d'autres préfèrent le juste milieu de la mise en cache d'objets partiels

Épisode 596

sans les relations établies, tandis que d'autres ne se cachent pas du tout. En fin de compte, fonctionnant à
le niveau de l'entité est le meilleur moyen de s'interfacer avec le cache, et est le plus naturel et
granularité pratique à utiliser pour l'API.

Le cache partagé est manipulé dans JPA via un slim javax.persistence.Cache
interface. Un objet qui implémente Cache peut être obtenu auprès du gestionnaire d'entités
factory en appelant EntityManagerFactory.getCache (). Même si un fournisseur ne
prend en charge la mise en cache, un objet Cache sera retourné, la différence étant que les opérations
n'aura aucun effet.

L'interface ne prend actuellement en charge qu'une méthode contains () et quelques méthodes
variantes d'expulsion. Bien qu'il ne s'agisse pas d'une API très complète, les applications ne devraient pas vraiment utiliser
une interface de mise en cache dans le code de l'application, donc peu d'API est nécessaire. En général,
les opérations de mise en cache sont principalement utiles pour les tests et le débogage, et les applications
ne devrait pas avoir besoin de modifier dynamiquement le cache lors de l'exécution. Le moyen le plus pratique
utiliser le cache, c'est simplement le vider entre les cas de test pour assurer un nettoyage correct et
isoler le comportement du test. Référencement12-25 montre un modèle de cas de test JUnit 4 simple qui garantit
que le cache partagé est effacé après chaque test est exécuté.

Annonce 12-25. Utilisation de l'interface de cache

```
classe publique SimpleTest {

    statique EntityManagerFactory emf;
    EntityManager em;

    @Avant les cours
    public static void classSetUp () {
        emf = Persistence.createEntityManagerFactory ("HR");
    }

    @Après les cours
    public static void classCleanUp () {
        emf.close ();
    }

    @Avant
    public void setUp () {
        em = emf.createEntityManager ();
    }
```

Épisode 597


```
    @Après
    public void cleanUp () {
        em.close ();
        emf.getCache (). evictAll ();
    }

    @Tester
    public void testMethod () {
        // Code de test ...
    }
}
```

Si nous avons accédé à une seule entité Employee avec la clé primaire 42 dans les tests et voulions être plus chirurgicaux sur ce que nous avons fait à la cache, nous ne pouvions expulser que cela entité en appelant expulser (Employee.class, 42). Ou, nous pourrions expulser toutes les instances du Classe d'employé en invoquant l'expulsion (Employee.class). Le problème de la suppression des entités ou des classes d'entités spécifiques est que si le cache est basé sur des objets, cela pourrait le laisser dans un état incohérent avec des références pendantes à des objets non mis en cache. Par exemple, si notre L'employé avec l'ID 42 avait une relation bidirectionnelle avec une entité Office et nous avons expulsé l'entité Employee en utilisant la méthode d'éviction basée sur la classe ou sur l'instance, nous laisserait l'entité Office en cache pointant vers un employé non mis en cache. Le suivant heure à laquelle l'employé avec l'ID 42 est interrogé et placé dans le cache, sa référence à son Office associé sera correctement configuré pour pointer vers l'entité Office mise en cache. Le problème est que le pointeur de retour du bureau vers l'employé ne sera pas corrigé et nous serait dans une position où l'entité Office ne pointe pas vers le même employé instance qui pointe vers elle. La morale est que ce n'est clairement pas une bonne idée d'aller autour de l'expulsion de classes d'objets qui ont des relations ou sont référencées par d'autres entités mises en cache. Nous préférons le gros marteau expulser tout ce qui balaie tout cache et garantit qu'il est entièrement propre et cohérent.

Du côté du débogage, nous pouvons vérifier si l'entité particulière est mise en cache par appel contient (Employee.class, 42). Un fournisseur qui ne fait aucune mise en cache partagée retournera simplement false à chaque invocation.

Épisode 598

Configuration statique du cache

La plus grande valeur offerte par l'abstraction du cache JPA est la possibilité de la configurer. La mise en cache peut être configurée au niveau de l'unité de persistance globale ou sur une par classe base. Il est réalisé grâce à une combinaison d'un paramètre d'unité de persistance et de paramètres de classe.

Le paramètre de cache d'unité de persistance est contrôlé par l'élément shared-cache-mode dans le fichier persistence.xml ou l'équivalent javax.persistence.sharedCache.mode propriété qui peut être transmise au moment de la création de l'usine du gestionnaire d'entités. Il a cinq options, dont l'un est le NOT_SPECIFIED par défaut. Cela signifie que lorsque le paramètre de cache partagé est non explicitement spécifié dans le fichier persistence.xml ou par la présence du javax.persistence.sharedCache.mode, il appartient au fournisseur de mettre en cache ou non cache, en fonction de ses propres valeurs par défaut et inclinaisons. Bien que cela puisse sembler un

peu étrange pour un développeur, c'est vraiment la valeur par défaut appropriée car différents fournisseurs ont des implémentations différentes qui reposent plus ou moins fortement sur la mise en cache.

Deux autres options, ALL et NONE, sont plus évidentes dans leur signification et leur sémantique, et faire en sorte que le cache partagé soit complètement activé ou désactivé, respectivement.

> Attention aux développeurs qui conservent des entités volatiles modifiées par plusieurs clients pensent souvent qu'ils devraient désactiver le cache partagé pour des raisons de cohérence. ce n'est pas souvent la bonne approche. le réglage du mode cache sur NONE ne peut pas ne cause qu'un ralentissement sévère de l'application mais peut aussi potentiellement faire échouer mécanismes de fournisseur qui optimisent la mise en cache. Utilisation du verrouillage, de la concurrence, et les mesures de rafraîchissement décrites plus haut dans ce chapitre sont les chemin recommandé.

Lorsqu'une classe d'entité est hautement volatile et hautement concurrente, elle est parfois avantageux de désactiver la mise en cache des instances de cette classe uniquement. Ceci est réalisé par définir le cache partagé sur DISABLE_SELECTIVE puis annoter l'entité spécifique classe qui doit rester non mise en cache avec @Cacheable (false). Le DISABLE_SELECTIVE L'option entraînera le comportement par défaut de mettre en cache toutes les entités de l'unité de persistance. Chaque fois qu'une classe d'entité est annotée avec @Cacheable (false), vous remplacez effectivement la valeur par défaut et la désactivation du cache pour les instances de ce type d'entité. Ceci peut être fait à autant de classes d'entités que vous le souhaitez. Lorsqu'elle est appliquée à une classe d'entité, la capacité de de ses sous-classes est également affectée par l'annotation @Cacheable sur l'entité. Ça peut être remplacé, cependant, au niveau de la sous-classe, si le besoin s'en fait sentir.

Épisode 599

Si vous arrivez au point où vous devez annoter plus de classes qu'autrement, vous pouvez emprunter la route opposée et définir le cache partagé sur ENABLE_SELECTIVE, ce qui signifie que le La valeur par défaut est de désactiver la mise en cache pour toutes les entités sauf celles qui ont été annotées avec @Cacheable (true) (ou simplement @Cacheable car la valeur par défaut de @Cacheable est true). En un mot, l'annotation @Cacheable est utile lorsque l'un des deux * _SELECTIVE les options sont en vigueur, et selon celle qui est active, les valeurs booléennes de tous les Les annotations @Cacheable dans l'unité de persistance doivent être toutes vraies ou toutes fausses.

Gestion dynamique du cache

Il est également possible au moment de l'exécution de remplacer si les entités sont lues à partir du cache pendant exécuter une requête ou être mis dans le cache lorsque des entités sont obtenues à partir de la base de données. Cependant, pour que ces remplacements soient effectifs, la mise en cache doit déjà être activée pour la ou les classes d'entités pertinentes. Cela peut être vrai car les paramètres statiques décrits dans la section précédente a été utilisée parce que le fournisseur utilise par défaut la mise en cache, ou car une option de mise en cache spécifique au fournisseur a activé le cache.

Nous avons mentionné les deux possibilités de lecture ou d'écriture dans le cache comme des options distinctes car, bien que liées, elles sont distinctes les unes des autres et peuvent être choisis indépendamment. Chacun a son propre nom de propriété et peut être passé en tant que propriété à un gestionnaire d'entités pour définir un comportement de mise en cache par défaut pour ce gestionnaire d'entités. Ça peut aussi être passé à une méthode find () ou comme un indice à une requête. Les noms de propriétés sont javax. persistence.cache.retrieveMode et javax.persistence.cache.storeMode, avec les valeurs étant membres de CacheRetrieveMode et CacheStoreMode énumérés types, respectivement.

Le mode de récupération a deux options simples: CacheRetrieveMode.USE pour utiliser le cache lors de la lecture d'entités de la base de données et CacheRetrieveMode.BYPASS pour contourner le cache. L'option USE est la valeur par défaut car la mise en cache doit quand même être activée pour la propriété à utiliser. Lorsque BYPASS est actif, les entités ne doivent pas être recherchées dans le

cache partagé. Notez que la seule raison pour laquelle USE existe même est de permettre la réinitialisation de la récupération retour à l'utilisation du cache lorsqu'un gestionnaire d'entités est défini sur BYPASS.

Remarque Dans des circonstances normales, le mode de récupération n'aura aucun effet pratique si les entités sont déjà présentes dans le contexte de persistance. mode de récupération uniquement détermine si une recherche dans le cache partagé est effectuée. si des entités interrogées existent dans le contexte de persistance active, ces instances seront toujours renvoyées.

Épisode 600

Le mode de stockage prend en charge une option CacheStoreMode.USE par défaut, qui place les objets dans le cache lorsqu'ils sont obtenus ou validés dans la base de données. Le CacheStoreMode. L'option BYPASS peut être utilisée pour empêcher l'insertion d'instances dans le cache partagé. UNE la troisième option de mode de stockage, CacheStoreMode.REFRESH, est utile lorsque les objets peuvent changer en dehors du domaine du cache partagé. Par exemple, si une entité peut être modifiée par un application différente qui utilise la même base de données, ou même par un gestionnaire d'entités différent factory (peut-être dans une JVM différente d'un cluster), l'instance dans le cache partagé peut devenir périmé. La définition du mode de stockage sur REFRESH entraînera l'instance d'entité dans le cache à actualiser lors de sa prochaine lecture à partir de la base de données.

Attention, l'option REFRESH doit toujours être activée s'il y a une chance ces données d'application peuvent être modifiées depuis l'extérieur de l'application Jpa.

Prenons l'exemple du Listing 12-26 qui renvoie toutes les actions ayant un prix supérieur à un certain montant. Les types de mode de cache de récupération et de stockage sont utilisés pour assurez-vous que les résultats sont aussi récents que possible et que le cache est actualisé avec ceux résultats.

Annonce 12-26. Utilisation des propriétés du mode cache

```
Liste publique <Stock> findExpensiveStocks (double seuil) {
    TypedQuery <Stock> q = em.createQuery (
        "SELECT s FROM Stock s WHERE s.price>: amount",
        Stock.class);
    q.setHint ("javax.persistence.cache.retrieveMode",
                    CacheRetrieveMode.BYPASS);
    q.setHint ("javax.persistence.cache.storeMode",
                    CacheStoreMode.REFRESH);
    q.setParameter ("montant", seuil);
    return q.getResultList ();
}
```

Au début, il peut sembler un peu étrange que le cache soit contourné lors de la récupération mode, mais en cours de rafraîchissement en mode magasin. Il suppose que toutes les requêtes ne être en contournant le cache, donc l'actualisation donnera aux hits de cache ultérieurs un accès au de nouvelles données.

Épisode 601

Notez que l'option REFRESH n'est pas nécessaire lorsque les entités ont simplement été
mis à jour dans une transaction. Pendant la validation, l'option par défaut du mode de stockage USE provoquera
l'entrée de cache partagé à mettre à jour avec les modifications de la transaction. L'ajout
La valeur de REFRESH s'applique uniquement aux lectures de base de données. C'est pourquoi REFRESH n'est pas nécessaire
dans le cas où la base de données est essentiellement dédiée à l'application JPA. Je tombe
les mises à jour de la base de données passent par l'application JPA, sa fabrique de gestionnaires d'entités partagées
le cache aurait toujours les données les plus à jour et il n'y aurait jamais de raison
pour rafraîchir le cache.

Une option de mode de stockage peut être passée dans une méthode à laquelle le mode de récupération fait
ne s'applique pas. La sémantique de la méthode refresh () du gestionnaire d'entités est que l'entité
l'instance dans le contexte de persistance est actualisée avec l'état de la base de données. Qui passe
dans un mode de récupération de USE a une valeur discutable car le point de rafraîchissement était de
obtenir les dernières données, et vous ne les trouverez que dans la base de données. Cependant, le rafraîchissement ()
la sémantique de la méthode n'inclut pas la mise à jour du cache partagé avec la nouvelle base de données
Etat. Pour que cela se produise, vous devez également inclure l'option REFRESH du mode de stockage en tant que
propriété argument de la méthode.

```
Accessoires HashMap = nouveau HashMap ();
props.put ("javax.persistence.cache.storeMode",
                CacheStoreMode.REFRESH);
em.refresh (emp, accessoires);
```

Conseil Pour les applications qui nécessitent REFRESH comme valeur par défaut, de nombreux fournisseurs
fournir un support pour le définir au niveau de l'unité de persistance en tant qu'unité de persistance
propriété. Parce que l'actualisation peut être plus nécessaire pour des classes d'entités spécifiques
que d'autres, les options REFRESH peuvent également être prises en charge au niveau de l'entité
class, ce qui signifie que toutes les entités d'un type donné sont automatiquement actualisées dans le
cache lors de la lecture de la base de données.

L'utilisation des options de mise en cache dynamique est préférable à la simple désactivation du cache,
soit globalement, soit par classe. Il vous donne un contrôle précis sur les performances
et la cohérence, et offre toujours au fournisseur la possibilité d'optimiser pour les cas
lorsque l'optimisation est appropriée.

Épisode 602

Classes d'utilité

Une poignée de méthodes sont disponibles sur deux interfaces utilitaires, PersistenceUnitUtil et
PersistenceUtil dans le package javax.persistence. Ces méthodes ne seront pas utilisées
souvent par une application au moment de l'exécution, mais peut être utile principalement pour les fournisseurs d'outils ou
cadres d'application.

PersistanceUtil

Une instance de PersistenceUtil est obtenue à partir de la classe Persistence dans les deux Java
Environnements SE et Java EE. La méthode statique getPersistenceUtil () est la seule
méthode sur la classe Persistence qu'une application gérée ou basée sur un conteneur
normalement utiliser dans un environnement géré. Il n'exporte que deux méthodes, les deux variantes
de déterminer si l'état est chargé ou non. La méthode isLoaded (Object)
retourne si l'entité transmise a tout son état non paresseux chargé. Par exemple, le

suivant peut retourner faux:

```
Persistence.getPersistenceUtil (). IsLoaded (
        em.getReference (Employee.class, 42));
```

Nous disons «pourrait» car le fournisseur est libre de charger certains ou tous les champs ou propriétés de l'instance Employee renvoyée; il n'est tout simplement pas obligé de le faire.

La deuxième variante, isLoaded (Object, String), accepte un paramètre String supplémentaire décrivant un attribut nommé de l'entité et retourne si cet attribut a été chargé dans l'instance d'entité transmise. Il renverra false si isLoaded (Object) est false, ou si l'attribut est marqué comme paresseux et n'a pas été chargé. En supposant une définition d'une entité Employee qui a un attribut de relation phoneNumbers marqué comme paresseux, le suivant renverra probablement faux:

```
Persistence.getPersistenceUtil (). IsLoaded (
        em.find (Employee.class, 42), "phoneNumbers");
```

La classe PersistenceUtil ne serait utilisée que du côté client, ou dans un autre couche d'application de la persistance, lorsque le gestionnaire d'entités ou l'usine associée à l'entité n'est pas connue. Chacune des méthodes entre dans une phase de résolution du fournisseur avant appeler la méthode correspondante sur le fournisseur approprié. Cela peut ajouter des frais généraux,

Épisode 603

en fonction de la mise en œuvre, de la mise en cache des fournisseurs et de la fréquence des appels. Sur côté serveur, vous pouvez savoir quel gestionnaire d'entités ou quelle fabrique utiliser pour l'entité, et dans ce cas, la classe PersistenceUnitUtil la plus efficace, décrite dans la section suivante, devrait être utilisé à la place.

PersistenceUnitUtil

Une instance PersistenceUnitUtil peut être obtenue à partir d'une fabrique de gestionnaires d'entités via la méthode getPersistenceUnitUtil (). Il sert de classe d'utilité pour le unité de persistance, et bien qu'elle ne contienne pas beaucoup de méthodes maintenant, à l'avenir plus de fonctions utilitaires seront ajoutées.

Les deux mêmes méthodes isLoaded () sont définies sur la classe PersistenceUnitUtil tels que définis dans la classe PersistenceUtil. La différence est que les invoquer sur cette classe ne nécessite pas de résolution du fournisseur. L'interface PersistenceUnitUtil est implémentée par le fournisseur, de sorte que l'utilisateur appelle déjà une classe de fournisseur qui est supposé avoir une connaissance approfondie des mappages du modèle de domaine et des Implémentation JPA.

Une méthode supplémentaire nommée getIdentifier () renvoie la valeur de l'identifiant attribut si l'entité a un identifiant simple ou incorporé. Si l'entité a plusieurs identifiant, une instance de la classe d'identifiant sera renvoyée. Cette méthode permet à une couche d'obtenir dynamiquement l'identifiant d'une entité donnée sans avoir pour tout savoir sur les mappages d'entités, ou même sur leur type. C'est un assez typique situation dans laquelle se trouve un framework. La méthode du Listing 12-27 recueille tous les identifiants pour une liste d'entités donnée.

Annonce 12-27. Collecte des identifiants d'entité

```
public List <Object> getEntityIdentifiers (List <T> entités) {
    PersistenceUnitUtil util = emf.getPersistenceUnitUtil ();
    List <Object> result = new ArrayList <Object> ();
    pour (T entité: entités) {
        result.add (util.getIdentifier (entité));
```

```
        }
        résultat de retour;
}
```


Épisode 604

Résumé

Ce chapitre a couvert un certain nombre de sujets divers, des événements à la mise en cache. Pas tout que nous avons décrit sera immédiatement utilisable dans une nouvelle application, mais certaines fonctionnalités, tels que le verrouillage optimiste, sont susceptibles de jouer un rôle de premier plan dans de nombreuses entreprises applications.

La section des rappels de cycle de vie a présenté le cycle de vie d'une entité et montré les points auxquels une application peut surveiller les événements qui sont déclenchés lors du déplacement d'une entité à travers différentes étapes de son cycle de vie. Nous avons examiné deux approches différentes pour implémentation de méthodes de rappel: sur la classe d'entité et dans le cadre d'un auditeur séparé classe.

Nous avons introduit la validation et donné un aperçu de ce qu'elle était, comment elle pouvait être utilisée, et comment il pourrait être étendu pour fournir des contraintes de validation spécifiques à l'application. Nous avons montré comment cela peut nous éviter d'avoir à écrire du code explicite pour les conditions d'erreur et vérification des limites. Nous avons apporté plus de contexte à la validation en expliquant comment c'est intégré à JPA et comment les points d'intégration peuvent être configurés pour répondre aux besoins de votre application.

Dans notre discussion sur le verrouillage et la gestion des versions, nous avons introduit le verrouillage optimiste et décrit le rôle vital qu'il joue dans de nombreuses applications, en particulier celles qui utilisent entités détachées. Nous avons également examiné les différents types d'options de verrouillage supplémentaires et quand ils peuvent être utilement appliqués. Nous avons expliqué leur correspondance avec l'isolement niveaux dans la base de données et la mesure dans laquelle ils devraient être invoqués. Nous avons décrit le difficultés de récupération après des échecs de verrouillage et quand il est approprié de rafraîchir l'état d'une entité gérée. Nous sommes allés au verrouillage pessimiste et à son impact sur l'évolutivité. nous décrit le mode pessimiste principal et deux autres modes moins répandus. Nous avons montré comment les délais d'expiration peuvent être configurés et mis en évidence les conditions dans lesquelles les deux différents types d'exceptions pessimistes sont lancés et peuvent être traités.

Nous avons examiné la mise en cache et passé du temps à examiner comment le cache partagé peut être géré et contrôlé à l'aide des paramètres de cache global et des propriétés du mode de cache local. Nous avons discuté de la façon dont les propriétés du mode cache affectent les requêtes et offert des conseils sur quels modes utiliser et quand les utiliser. Enfin, nous avons découvert quelques-uns des JPA classes d'utilité qui fournissent des fonctionnalités supplémentaires, telles que la capacité de déterminer si un L'entité JPA a été entièrement chargée et a obtenu l'identifiant de toute instance d'entité.

Dans le chapitre suivant, nous examinons le fichier de mappage XML, montrant comment utiliser XML avec ou au lieu d'annotations, et comment les métadonnées d'annotation peuvent être remplacées.

Épisode 605

CHAPITRE 13

Fichiers de mappage XML

Au début, après la sortie de Java SE 5, il y avait un silence, et parfois
pas si calme, débat sur la question de savoir si les annotations étaient meilleures ou pires que XML. le
les défenseurs des annotations ont vigoureusement proclamé à quel point les annotations sont tellement plus simples
et fournissez des métadonnées intégrées qui sont co-localisées avec le code qu'elles décrivent. le
affirmait que cela évite d'avoir à reproduire les informations inhérentes au
contexte du code source où s'appliquent les métadonnées. Les partisans du XML ont alors rétorqué
que les annotations couplent inutilement les métadonnées au code, et cela change en
les métadonnées ne doivent pas nécessiter de modifications du code source.

La vérité est que les deux parties avaient raison et qu'il y a des moments appropriés pour utiliser
les métadonnées d'annotation et d'autres moments d'utilisation de XML. Quand les métadonnées sont vraiment
couplé au code, il est logique d'utiliser des annotations car les métadonnées sont
juste un autre aspect du programme. Par exemple, la spécification du champ d'identifiant de
une entité n'est pas seulement une information pertinente pour le fournisseur, mais aussi une
détail connu et assumé par le code de l'application référençant. Autres types de métadonnées,
telle que la colonne à laquelle un champ est mappé, peut être modifiée en toute sécurité sans avoir besoin
pour changer le code. Ces métadonnées s'apparentent aux métadonnées de configuration et peuvent être
mieux exprimé en XML, où il peut être configuré selon le modèle d'utilisation ou
environnement d'exécution.

Les arguments avaient également tendance à compartimenter injustement la question parce que
réalité, cela va plus loin que de simplement décider quand il serait logique d'utiliser un type
de métadonnées ou autre. Dans de nombreuses discussions et forums avant la sortie de JPA 1.0
spécification, nous avons demandé aux gens s'ils prévoyaient d'utiliser des annotations ou du XML, et
nous avons constamment vu qu'il y avait une scission. La raison était qu'il y avait d'autres facteurs
qui n'ont rien à voir avec ce qui est meilleur, comme les processus de développement existants,
systèmes de contrôle de source, expérience des développeurs, etc.

Maintenant que les développeurs ont quelques années d'utilisation d'annotations sous leur ceinture, il y a
pas presque la même hésitation à incorporer des annotations dans leur code comme il y en avait autrefois. Dans
En fait, la plupart des gens sont parfaitement satisfaits des annotations, ce qui rend leur acceptation jolie

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_13

Épisode 606

beaucoup un fait accompli. Néanmoins, il existe encore des cas d'utilisation de XML, donc nous
continuez à décrire et à illustrer comment les métadonnées de mappage peuvent être spécifiées dans
l'un ou l'autre format. En fait, l'utilisation du mappage XML est définie pour permettre l'utilisation des annotations et
puis remplacé par XML. Cela offre la possibilité d'utiliser des annotations pour certaines choses
et XML pour les autres, ou pour utiliser des annotations pour une configuration attendue mais fournir ensuite
un fichier XML de remplacement pour convenir à un environnement d'exécution particulier. Le fichier XML peut être
sparse et ne fournissez que les informations qui sont remplacées. Vous verrez plus tard dans
ce chapitre que la granularité avec laquelle ces métadonnées peuvent être spécifiées offre une bonne
beaucoup de flexibilité de mappage objet-relationnel.

Les fichiers et schémas de mappage persistence.xml et orm.xml ont été mis à jour dans JPA
version 2.2.

- Le fichier persistence.xml définit une unité de persistance et se trouve dans
  le répertoire META-INF de la racine de l'unité de persistance.
- Le fichier orm.xml, contenu dans le répertoire META-INF de la racine de
  l'unité de persistance, inclut les classes de persistance gérées utilisées
  prendre la forme d'annotations du mapping objet-relationnel
  information. Le fichier de mappage orm.xml ou un autre fichier de mappage sera
  chargé en tant que ressource par le fournisseur de persistance.

Remarque Les versions JPA 2.1 et 2.2 demandent que les mappages de fichiers XML, tels que persistence.xml et orm.xml, se trouvent dans le chemin de classe Java.

La version JPA 2.2 dit:

> «Un fichier XML de mappage objet / relationnel nommé orm.xml peut être spécifié dans le répertoire META-INF à la racine de la persistance unité ou dans le répertoire META-INF de tout fichier JAR référencé par le persistence.xml. »

Notez que nous pouvons ajouter d'autres fichiers de mappage qui peuvent être présents n'importe où sur le classpath et le ClassLoader peuvent les charger en tant que ressources.

Au cours de ce chapitre, nous décrivons la structure et le contenu du fichier de mappage et son lien avec les annotations de métadonnées. Nous discutons également de la façon dont XML les métadonnées de mappage peuvent se combiner avec les métadonnées d'annotation et les remplacer. Nous avons

Épisode 607

essayé de structurer le chapitre dans un format qui lui permettra d'être utilisé à la fois comme source de informations et une référence pour le format de fichier de mappage.

Le puzzle des métadonnées

Les règles de XML et d'utilisation et de remplacement des annotations peuvent être un peu déroutantes pour dire que le moins, surtout compte tenu de l'espace de permutation du mélange d'annotations avec XML. L'astuce comprendre la sémantique et être capable de spécifier correctement les métadonnées de la manière que vous souhaitez qu'il soit spécifié, c'est de comprendre le processus de collecte de métadonnées. Une fois que vous aurez une solide compréhension de ce que fait le processeur de métadonnées, vous serez en bonne voie pour comprendre ce que vous devez faire pour obtenir un résultat spécifique.

Le fournisseur peut choisir d'effectuer le processus de collecte de métadonnées de quelque manière que ce soit il choisit, mais le résultat est qu'il doit respecter les exigences de la spécification. Les développeurs comprennent les algorithmes, nous avons donc décidé qu'il serait plus facile de comprendre si nous avons présenté la fonctionnalité logique comme un algorithme, même si le l'implémentation pourrait ne pas l'implémenter de cette façon. L'algorithme suivant peut être considéré comme la logique simplifiée pour obtenir les métadonnées pour l'unité de persistance:

1. Traitez les annotations. L'ensemble des entités, superclasses mappées, et les objets embarqués (nous appelons cet ensemble E) sont découverts par à la recherche de @Entity, @MappedSuperclass et @Embeddable annotations. Les annotations de classe et de méthode dans toutes les classes dans l'ensemble E sont traités et les métadonnées résultantes sont stockées dans l'ensemble C. Toutes les métadonnées manquantes qui n'étaient pas explicitement spécifiées dans le les annotations sont vides.

2. Ajoutez les classes définies en XML. Recherchez toutes les entités, mappées superclasses et objets incorporés définis dans le mapper les fichiers et les ajouter à E. Si nous constatons que l'une des classes existe déjà en E, nous appliquons les règles de priorité pour le niveau de classe métadonnées que nous avons trouvées dans le fichier de mappage. Ajoutez ou ajustez le métadonnées au niveau de la classe en C selon les règles de priorité.

Épisode 608

3. Ajoutez les mappages d'attributs définis dans XML. Pour chaque classe en E,
   regardez les champs ou propriétés dans le fichier de mappage et essayez d'ajouter le
   métadonnées de la méthode à C.Si le champ ou la propriété existe déjà,
   appliquer les règles de priorité pour les métadonnées de mappage au niveau des attributs.

4. Appliquez les valeurs par défaut. Déterminez toutes les valeurs par défaut selon le
   règles de portée et où les valeurs par défaut ont pu être définies
   (voir ce qui suit pour la description des règles par défaut). Les classes,
   mappages d'attributs et autres paramètres qui n'ont pas encore été renseignés
   dans sont assignées des valeurs et mises en C.

Certains des cas suivants peuvent entraîner une légère modification de cet algorithme, mais en général, c'est ce qui se passe logiquement lorsque le fournisseur a besoin d'obtenir le mappage des métadonnées.

Vous avez déjà appris dans les chapitres sur le mappage que les annotations peuvent être rares et que ne pas annoter un attribut persistant le fera généralement mapper par défaut comme une cartographie de base. D'autres valeurs par défaut de mappage ont également été expliquées, et vous avez vu combien Ils ont facilité la configuration et le mappage des entités. Vous remarquerez dans notre algorithme que les valeurs par défaut sont appliquées à la fin, donc les mêmes valeurs par défaut que vous avez vues pour les annotations être également appliqué lors de l'utilisation de fichiers de mappage. Cela devrait être un peu réconfortant pour les utilisateurs XML que les fichiers de mappage peuvent être spécifiés de manière clairsemée de la même manière que les annotations. Ils aussi ont les mêmes exigences pour ce qui doit être spécifié; par exemple, un identifiant doit être spécifié, un mappage de relation doit avoir au moins sa cardinalité spécifiée, et ainsi de suite.

Le fichier de mappage

À ce stade, vous savez bien que si vous ne souhaitez pas utiliser XML pour le mappage, vous pas besoin d'utiliser XML. En fait, comme vous le verrez au chapitre14, n'importe quel nombre de mappage les fichiers, ou aucun, peuvent être inclus dans une unité de persistance. Si vous en utilisez un, cependant, chacun Le fichier de mappage fourni doit être conforme et valide par rapport au *XSD (XML Schema Définition)* fichier de schéma.

Épisode 609

Remarque Lors de l'utilisation d'une implémentation JPA 1.0, JPA 2.0 ou JPA 2.1, le schéma sera respectivement orm_1_0.xsd, orm_2_0.xsd ou orm_2_1.xsd, situé à http://xmlns.jcp.org/xml/ns/persistence/ . Lors de l'utilisation de JPA 2.2, le schéma sera à la place nommé persistence_2_2.xsd. il sera situé comme avant à http://xmlns.jcp.org/xml/ns/persistence/.

Ce schéma définit un espace de noms appelé http://xmlns.jcp.org/xml/ns/
persistence / orm qui inclut tous les éléments ORM qui peuvent être utilisés dans un fichier de mappage.
Un en-tête XML typique pour un fichier de mappage est présenté dans le Listing 13-1. Vous remarquerez qu'il
fait référence au fait que le fichier de schéma XML peut être nommé META-INF / orm.xml dans la persistance
archive ou il peut être nommé un autre nom, qui serait utilisé pour localiser le fichier comme
resource sur le chemin de classe.

Annonce 13-1. En-tête XML pour le fichier de mappage

```
<? xml version = "1.0" encoding = "UTF-8"?>
<persistence xmlns = "http://xmlns.jcp.org/xml/ns/persistence"
      xmlns: xsi = "http://www.w3.org/2001/XMLSchema-instance"
      xsi: schemaLocation = "http://xmlns.jcp.org/xml/ns/persistence
      http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd "
      version = "2.2">

<xsd: annotation>
    <xsd: documentation>
        @ (#) orm_2_2.xsd 2.2 7 juillet 2017
          </ xsd: documentation>
</ xsd: annotation>

<xsd: annotation>
    <xsd: documentation> <! [CDATA [
    Il s'agit du schéma XML de l'objet de persistance / du fichier de mappage relationnel.
    Le fichier peut être nommé "META-INF / orm.xml" dans la persistance
    archive ou il peut être nommé un autre nom qui serait
    utilisé pour localiser le fichier en tant que ressource sur le chemin de classe.
    Les fichiers de mappage objet / relationnel doivent indiquer l'objet / relationnel
    schéma de fichier de mappage à l'aide de l'espace de noms de persistance:
```

Épisode 610


```
    http://xmlns.jcp.org/xml/ns/persistence/orm
    et indiquez la version du schéma par
    en utilisant l'élément version comme indiqué ci-dessous:

    <entity-mappings xmlns = "http://xmlns.jcp.org/xml/ns/persistence/orm"
        xmlns: xsi = "http://www.w3.org/2001/XMLSchema-instance"
        xsi: schemaLocation = "http://xmlns.jcp.org/xml/ns/persistence/orm
        http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd "
        version = "2.2">
        ...
    </entity-mappings>
      ]]> </ xsd: documentation>
</ xsd: annotation>
```

L'élément racine du fichier de mappage est appelé mappages d'entités. Tout objet-relationnel
Les métadonnées XML sont contenues dans cet élément et, comme le montre l'exemple, le
les informations d'en-tête sont également spécifiées comme attributs dans cet élément. Les sous-éléments
des mappages d'entités peuvent être classés en quatre principaux périmètres et
groupes: valeurs par défaut des unités de persistance, valeurs par défaut des fichiers de mappage, requêtes et générateurs, et
classes et mappages gérés. Il existe également un paramètre spécial qui détermine si
les annotations doivent être prises en compte dans les métadonnées de l'unité de persistance. Ces groupes
sont abordés dans les sections suivantes. Par souci de brièveté, nous n'inclurons pas le
les informations d'en-tête dans les exemples XML de ces sections.

Désactivation des annotations

Pour ceux qui sont parfaitement satisfaits de XML et ne ressentent pas le besoin d'annotations, il existe des moyens d'ignorer la phase de traitement des annotations (étape 1 de la précédente algorithme). L'élément xml-mapping-metadata-complete et metadata-complete l'attribut fournit un moyen pratique de réduire la surcharge requise pour découvrir et traiter toutes les annotations sur les classes dans l'unité de persistance. C'est aussi un moyen de désactiver efficacement toutes les annotations existantes. Ces options entraîneront le processeur pour les ignorer complètement comme s'ils n'existaient pas du tout.

Épisode 611

xml-mapping-metadata-complete

Lorsque l'élément xml-mapping-metadata-complete est spécifié, toutes les annotations du l'unité de persistance entière sera ignorée, et seuls les fichiers de mappage dans la persistance l'unité sera considérée comme l'ensemble total de métadonnées fournies. Uniquement les entités, mappées les superclasses et les objets incorporés qui ont des entrées dans un fichier de mappage seront ajoutés à l'unité de persistance.

L'élément xml-mapping-metadata-complete ne doit figurer que dans l'un des fichiers de mappage s'il existe plusieurs fichiers de mappage dans l'unité de persistance. Il est précisé comme sous-élément vide de l'élément persistence-unit-metadata, qui est le premier[1] sous-élément de mappages d'entités. Un exemple d'utilisation de ce paramètre est présenté dans Référencement 13-2.

Liste 13-2. Désactivation des métadonnées d'annotation pour l'unité de persistance

```
<entity-mappings>
    <persistence-unit-metadata>
        <xml-mapping-metadata-complete />
    </persistence-unit-metadata>
    ...
</entity-mappings>
```

S'il est activé, il n'y a aucun moyen de remplacer ce paramètre de manière portable. Il s'appliquera globalement à l'unité de persistance, indépendamment du fait qu'un attribut de métadonnées complet soit défini sur false dans une entité.

métadonnées complètes

L'attribut metadata-complete est un attribut de l'entité, mapped-superclass, et éléments intégrables. Si spécifié, toutes les annotations sur la classe spécifiée et sur tout les champs ou les propriétés de la classe seront ignorés, et seules les métadonnées du fichier de mappage sera considéré comme l'ensemble des métadonnées de la classe.

[1] Techniquement, il existe un élément de description dans de nombreux éléments, tout comme dans la plupart des les schémas standard de Java EE, mais ils ont peu de valeur fonctionnelle et ne seront pas mentionnés ici. Ils peuvent être d'une certaine utilité pour les outils qui analysent les schémas XML et utilisent les descriptions pour info-bulles et actions similaires.

Épisode 612

> Attention Les annotations définissant des requêtes, des générateurs ou des mappages d'ensembles de résultats sont ignorés s'ils sont définis sur une classe marquée comme métadonnée complète dans un fichier de mappage XML.

Lorsque la saisie semi-automatique des métadonnées est activée, les mêmes règles que celles que nous avons appliquées sont annotées les entités s'appliqueront toujours lors de l'utilisation d'entités mappées XML. Par exemple, l'identifiant doit être mappé, et toutes les relations doivent être spécifiées avec leur correspondant mappages de cardinalité à l'intérieur de l'élément entité.

Un exemple d'utilisation de l'attribut metadata-complete est présenté dans la liste 13-3 . le les mappages d'entités dans la classe annotée sont désactivés par l'attribut metadata-complete, et comme les champs ne sont pas mappés dans le fichier de mappage, le mappage par défaut les valeurs seront utilisées. Les champs de nom et de salaire seront mappés avec le NOM et le SALAIRE colonnes, respectivement.

Liste 13-3. Désactivation des annotations pour une classe gérée

```
@Entité
Employé de classe publique {
    @Id id int privé;
    @Column (nom = "EMP_NAME")
    nom de chaîne privé;
    @Column (nom = "SAL")
    long salaire privé;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    ...
    <entity class = "examples.model.Employee"
                metadata-complete = "true">
```


Épisode 613


```
            <attributs>
                <id name = "id" />
            </attributes>
        </entity>
        ...
</ entity-mappings
```

Valeurs par défaut de l'unité de persistance

L'une des conditions d'utilisation des métadonnées d'annotation est que nous devons avoir quelque chose

pour annoter. Si nous voulons définir des métadonnées pour une unité de persistance, nous sommes dans le position malheureuse de ne rien avoir à annoter car une unité de persistance est
juste un regroupement logique de classes Java, essentiellement une configuration. Cela nous ramène à
la discussion que nous avons eue plus tôt lorsque nous avons décidé que si les métadonnées ne sont pas couplées au code,
peut-être que cela ne devrait pas vraiment figurer dans le code. Voici les raisons pour lesquelles l'unité de persistance
les métadonnées ne peuvent être spécifiées que dans un fichier de mappage XML.

En général, une unité de persistance par défaut signifie que chaque fois qu'une valeur pour ce paramètre est
non spécifié à une portée plus locale, la valeur par défaut de l'unité de persistance s'appliquera. C'est un
moyen pratique de définir des valeurs par défaut qui s'appliqueront à toutes les entités, superclasses mappées,
et les objets incorporés dans l'ensemble de l'unité de persistance, qu'ils soient dans l'un des fichiers de mappage
ou classes annotées. Les valeurs par défaut ne seront pas appliquées si une valeur est présente à tout
niveau inférieur à l'unité de persistance. Cette valeur peut être sous la forme d'un fichier de mappage par défaut
valeur, une valeur dans un élément d'entité ou une annotation sur l'une des classes gérées
ou champs ou propriétés persistants.

L'élément qui renferme toutes les valeurs par défaut au niveau de l'unité de persistance est le
élément nommé persistence-unit-defaults. C'est l'autre sous-élément du
élément persistence-unit-metadata (après xml-mapping-metadata-complete). Si plus
qu'un seul fichier de mappage existe dans une unité de persistance, un seul des fichiers doit contenir ces
éléments.

Il y a six paramètres qui peuvent être configurés pour avoir des valeurs par défaut pour la persistance
unité. Ils sont spécifiés à l'aide du schéma, du catalogue, des identificateurs délimités, de l'accès,
éléments cascade-persist et entity-listeners.

Épisode 614

schéma

L'élément schema est utile si vous ne souhaitez pas avoir à spécifier un schéma dans chaque
@Table, @SecondaryTable, @JoinTable, @CollectionTable ou @TableGenerator
annotation; ou table, table secondaire, table de jointure, table de collection ou table
Élément XML générateur dans l'unité de persistance. Lorsqu'il est défini ici, il s'appliquera à toutes les tables
dans l'unité de persistance, qu'ils aient été réellement définis ou définis par défaut par le fournisseur.
La valeur de cet élément peut être remplacée par l'un des éléments suivants:

- élément de schéma défini dans les valeurs par défaut du fichier de mappage (voir le
  Section "Mappage des valeurs par défaut des fichiers")
- attribut de schéma sur n'importe quelle table, table secondaire, table de jointure,
  table-collection, générateur de séquence ou générateur de table
  élément dans un fichier de mappage
- schéma défini dans une @Table, @SecondaryTable, @JoinTable,
  @CollectionTable, @SequenceGenerator ou @TableGenerator
  annotation; ou dans une annotation @TableGenerator
  (sauf si xml- mapping- metadata-complete est défini)

Référencement 13-4 montre un exemple de configuration du schéma pour toutes les tables du
unité de persistance dont le schéma n'est pas encore défini.

Annonce 13-4. Définition du schéma d'unité de persistance par défaut

```
<entity-mappings>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <schema> RH </schema>
```

```
            </persistence-unit-defaults>
        </persistence-unit-metadata>
        ...
</entity-mappings>
```


Épisode 615

catalogue

L'élément de catalogue est exactement analogue à l'élément de schéma, mais c'est pour les bases de données qui prennent en charge les catalogues. Il peut être utilisé indépendamment, que le schéma soit spécifié ou not, a le même comportement que schema et est remplacé exactement de la même manière. le exactement les mêmes règles peuvent être appliquées au fichier de mappage de catalogue par défaut, comme décrit dans le section de schéma précédente.

identifiants délimités

L'élément délimité-identificateurs provoque la table de base de données, le schéma et la colonne les identificateurs utilisés dans l'unité de persistance, définis sous forme d'annotation, XML ou par défaut, à délimiter lors de son envoi à la base de données (voir chapitre 10 pour en savoir plus sur délimité identifiants). Il ne peut pas être désactivé localement, il est donc important d'avoir une compréhension complète des conséquences avant d'activer cette option. Si une annotation ou un élément XML est délimités localement par des guillemets, ils seront traités comme faisant partie du nom de l'identifiant.

Aucune valeur ni aucun texte n'est inclus dans l'élément délimité-identificateurs. Seul le vide L'élément doit être spécifié dans l'élément persistence-unit-defaults pour activer délimitation de l'identifiant d'unité de persistance.

accès

L'élément d'accès défini dans la section persistence-unit-defaults est utilisé pour définir le type d'accès pour toutes les classes gérées dans l'unité de persistance qui ont XML entrées mais ne sont pas annotées. Sa valeur peut être FIELD ou PROPERTY, indiquant comment le fournisseur doit accéder à l'état persistant.

Le paramètre d'accès est une valeur par défaut légèrement différente qui n'affecte aucun des classes gérées qui ont des champs ou des propriétés annotés. C'est une commodité pour quand XML est utilisé et évite d'avoir à spécifier l'accès pour toutes les entités répertoriées dans tous les Fichiers de mappage XML.

Cet élément affecte uniquement les classes gérées définies dans les fichiers de mappage car une classe avec des champs ou des propriétés annotés est considérée comme ayant remplacé le mode d'accès en raison de ses annotations placées sur ses champs ou propriétés. Si l'élément xml- mapping- metadata-complete est activé, l'accès à l'unité de persistance la valeur par défaut sera appliquée à ces classes annotées qui ont des entrées en XML. En d'autres termes, les annotations qui auraient autrement remplacé le mode d'accès ne seraient plus pris en compte, et les valeurs par défaut XML, y compris le mode d'accès par défaut, seraient appliquées.

Épisode 616

La valeur de cet élément peut être remplacée par un ou plusieurs des éléments suivants:

- élément d'accès défini dans les valeurs par défaut du fichier de mappage (voir le Section "Mappage des valeurs par défaut des fichiers")
- attribut d'accès sur n'importe quelle entité, superclasse mappée ou intégrable élément dans un fichier de mappage
- attribut d'accès sur n'importe quel élément de base, id, id intégré, intégré, plusieurs-à-un, un-à-un, un-à-plusieurs, plusieurs-à-plusieurs, une collection d'éléments ou élément de version dans un fichier de mappage
- Annotation @Access sur n'importe quelle entité, superclasse mappée ou classe intégrable
- Annotation @Access sur n'importe quel champ ou propriété d'une entité, mappée superclasse ou objet incorporé
- Un champ annoté ou une propriété dans une entité, une superclasse mappée ou objet incorporé

Référencement 13-5 montre un exemple de réglage du mode d'accès sur PROPRIÉTÉ pour tous les classes gérées dans l'unité de persistance qui n'ont pas de champs annotés.

Annonce 13-5. Définition du mode d'accès par défaut pour l'unité de persistance

```
<entity-mappings>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <access> PROPRIÉTÉ </access>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    ...
</entity-mappings>
```

cascade-persister

L'élément cascade-persist est unique d'une manière différente. Quand la cascade vide l'élément persist est spécifié, cela revient à ajouter l'option de cascade PERSIST à tous les relations dans l'unité de persistance. Reportez-vous au chapitre 6 pour une discussion sur la options en cascade sur les relations.

Épisode 617

Le terme «persistance par joignabilité» est souvent utilisé pour signifier que lorsqu'un objet est persisté, tous les objets accessibles à partir de cet objet sont également persistants automatiquement. L'élément cascade-persist fournit la sémantique de persistance par joignabilité que certaines personnes ont l'habitude d'avoir. Ce paramètre ne peut actuellement pas être remplacé, le l'hypothèse étant que lorsque quelqu'un est habitué à la persistance, par l'accessibilité sémantique, ils ne veulent normalement pas le désactiver. Si un contrôle plus fin sur la mise en cascade de l'opération persist est nécessaire, cet élément ne doit pas être spécifié et le les relations doivent avoir l'option de cascade PERSIST spécifiée localement.

Un exemple d'utilisation de l'élément cascade-persist est présenté dans le Listing 13-6 .

Annonce 13-6. Configuration de la sémantique de persistance par joignabilité

```
<entity-mappings>
    <persistence-unit-metadata>
        <persistence-unit-defaults>
```

```
                <cascade-persist />
            </persistence-unit-defaults>
        </persistence-unit-metadata>
        ...
</entity-mappings>
```

écouteurs d'entités

C'est le seul endroit où une liste d'écouteurs d'entités par défaut peut être spécifiée. Un défaut l'écouteur d'entité est un écouteur qui sera appliqué à chaque entité de l'unité de persistance. Ils seront appelés dans l'ordre dans lequel ils sont répertoriés dans cet élément, avant tout autre auditeur ou la méthode de rappel est appelée sur l'entité. C'est l'équivalent logique de l'ajout du écouteurs de cette liste au début de la liste @EntityListeners dans la superclasse racine. nous discuté des écouteurs d'entités dans le dernier chapitre, alors reportez-vous au chapitre 12 pour revoir l'ordre des invocation si vous en avez besoin. Une description de la manière de spécifier un écouteur d'entité est donnée dans le Section «Entity Listeners» de ce chapitre.

Épisode 618

L'élément d'entité-écouteurs est composé de zéro ou plusieurs écouteurs d'entité éléments qui définissent chacun un écouteur d'entité. Ils peuvent être remplacés ou désactivés dans l'une des deux manières suivantes:

- élément exclude-default-listeners dans une entité ou élément de fichier de mappage mapped- superclass
- Annotation @ExcludeDefaultListeners sur une entité ou mappée superclass (sauf si xml-mapping-metadata-complete est défini)

Mappage des valeurs par défaut des fichiers

Le niveau suivant de valeurs par défaut, après ceux définis pour l'ensemble de l'unité de persistance, sont ceux qui concernent uniquement les entités, les superclasses mappées et les objets incorporés qui sont contenu dans un fichier de mappage particulier. En général, s'il existe une unité de persistance par défaut définie pour le même paramètre, cette valeur remplacera l'unité de persistance par défaut pour le classes gérées dans le fichier de mappage. Contrairement aux valeurs par défaut de l'unité de persistance, le mappage les valeurs par défaut des fichiers n'affectent pas les classes gérées qui sont annotées et non définies dans le fichier de mappage. En termes de notre algorithme, les valeurs par défaut de cette section s'appliquent à tous les classes de C qui ont des entrées dans le fichier de mappage.

Les valeurs par défaut du fichier de mappage se composent de quatre sous-éléments facultatifs du Élément entity-mappings. Il s'agit du package, du schéma, du catalogue et de l'accès; et ils suivez l'élément persistence-unit-metadata.

paquet

L'élément package est destiné à être utilisé par les développeurs qui ne veulent pas pour répéter le nom de classe complet dans toutes les métadonnées du fichier de mappage. Ça peut être remplacé dans le fichier de mappage en qualifiant complètement un nom de classe dans n'importe quel élément ou attribut dans lequel un nom de classe est attendu. Ce sont les suivants:

- attribut class de la classe id, de l'entité-écouteur, de l'entité, mappé - superclasse, ou éléments intégrables

• attribut d'entité cible plusieurs-à-un, un-à-un, un-à-plusieurs,
et éléments plusieurs-à-plusieurs

• attribut de classe cible de l'élément de collection d'élément

Épisode 619

• attribut result-class de l'élément de requête native nommée

• attribut de classe d'entité de l'élément de résultat d'entité

Un exemple d'utilisation de cet élément est présenté dans l'extrait 13-7. Nous définissons la valeur par défaut mapper le nom du package de fichier à examples.model pour le fichier de mappage entier et peut simplement utilisez les noms de classe Employee et EmployeePK non qualifiés dans tout le fichier. le le nom du package ne sera pas appliqué à OtherClass, car il est déjà entièrement spécifié.

Annonce 13-7. Utilisation de l'élément package

```
<entity-mappings>
    <package> examples.model </package>
    ...
    <entity class = "Employee">
        <id-class class = "EmployeePK" />
        ...
    </entity>
    <entity class = "examples.tools.OtherClass">
        ...
    </entity>
    ...
</entity-mappings>
```

schéma

L'élément schema définira un schéma par défaut à assumer pour chaque table, table secondaire, table de jointure ou générateur de table définie ou par défaut dans le mappage fichier. Cet élément peut être remplacé par la spécification de l'attribut schema sur tout table, table secondaire, table de jointure, table de collection, générateur de séquence ou élément générateur de table dans le fichier de mappage.

Référencement 13-8 montre le schéma de fichier de mappage défini par défaut sur HR, donc la table EMP qui L'employé est mappé à est supposé être dans le schéma HR.

Épisode 620

Annonce 13-8. Utilisation de l'élément schema

```
<entity-mappings>
    <package> examples.model </package>
    <schema> RH </schema>
    ...
    <entity class = "Employee">
        <nom de la table = "EMP" />
        ...
    </entity>
    ...
</entity-mappings>
```

Le schéma par défaut du fichier de mappage affectera également la @Table, @SecondaryTable, @JoinTable, @CollectionTable, @SequenceGenerator et @TableGenerator annotations sur les classes qui ont des entrées dans le fichier de mappage. Par exemple, parce que L'employé est répertorié dans le fichier de mappage, il devient partie de l'ensemble de classes auquel le la valeur par défaut s'applique. S'il y avait un @TableGenerator (name = "EmpGen", table = "IDGEN") annotation sur Employee, le fichier de mappage par défaut lui sera appliqué, et l'IDGEN sera supposé être dans le schéma HR.

catalogue

L'élément de catalogue est à nouveau exactement analogue à l'élément de schéma, mais il est pour bases de données prenant en charge les catalogues. Il peut être utilisé indépendamment du fait que le schéma soit spécifié ou non, a le même comportement que le schéma au niveau par défaut du fichier de mappage, et est remplacé exactement de la même manière. Comme nous l'avons mentionné dans la section des unités de persistance, les mêmes règles peuvent être appliquées à la valeur par défaut du fichier de mappage de catalogue, comme décrit dans la section par défaut du fichier de mappage de schéma.

Épisode 621

accès

La définition d'un mode d'accès particulier comme valeur par défaut du fichier de mappage affecte uniquement classes gérées définies dans le fichier de mappage. L'accès au fichier de mappage par défaut Le mode peut être remplacé par un ou plusieurs des éléments suivants:

- attribut d'accès sur n'importe quelle entité, superclasse mappée ou intégrable élément dans un fichier de mappage
- attribut d'accès sur n'importe quel élément de base, id, intégré-id, intégré, plusieurs-à-un, un-à-un, un-à-plusieurs, plusieurs-à-plusieurs, element-collection ou élément de version dans un fichier de mappage
- Annotation @Access sur n'importe quelle entité, superclasse mappée ou classe intégrable
- Annotation @Access sur n'importe quel champ ou propriété d'une entité, mappée superclasse ou objet incorporé
- Un champ annoté ou une propriété dans une entité, une superclasse mappée ou objet incorporé

Requêtes et générateurs

Certains artefacts de persistance, tels que les générateurs d'ID et les requêtes, sont définis comme des annotations sur une classe même si elles sont en fait globales à l'unité de persistance dans la portée car ce sont des annotations et il n'y a pas d'autre endroit pour les mettre que sur une classe. Plus tôt, nous avons souligné le caractère inapproprié d'exprimer les métadonnées des unités de persistance comme des annotations sur une classe aléatoire, mais les générateurs et les requêtes créent quelque chose de concret, au lieu d'être juste des paramètres. Néanmoins, ce n'est toujours pas idéal, et en XML ce global les métadonnées liées aux requêtes n'ont pas besoin d'être placées arbitrairement dans une classe mais peuvent être défini au niveau des sous-éléments de l'élément entity-mappings.

Les éléments de métadonnées de la requête globale sont constitués d'éléments de générateur et de requête qui incluent le générateur de séquence, le générateur de table, la requête nommée, la requête nommée native, et sql-result-set-mapping. Pour des raisons historiques, ces éléments peuvent apparaître dans contextes différents, mais ils sont néanmoins toujours limités à l'unité de persistance. Il y a trois espaces de noms d'unité de persistance différents, un pour les requêtes, un pour les générateurs et un pour les mappages d'ensembles de résultats utilisés pour les requêtes natives. Lorsque l'un des éléments que nous

Épisode 622

juste listés sont définis dans le fichier de mappage, les artefacts qu'ils définissent seront ajoutés dans le espace de noms d'unité de persistance auquel ils s'appliquent.

Les espaces de noms contiendront déjà tous les artefacts d'unité de persistance existants qui peuvent avoir été définis dans des annotations ou dans un autre fichier de mappage. Parce que ces artefacts partagent le même type d'espace de noms d'unité de persistance globale, lorsque l'un des les artefacts définis dans XML partagent le même nom que celui qui existe déjà dans le espace de noms du même type, il est considéré comme un remplacement. L'artefact défini en XML remplace celui qui a été défini par l'annotation. Il n'y a pas de concept de remplacer des requêtes, des générateurs ou des mappages d'ensembles de résultats au sein du même ou de différents fichiers de mappage. Si un ou plusieurs fichiers de mappage contiennent l'un de ces objets définis avec le même nom, il n'est pas défini, ce qui remplace l'autre car l'ordre dans lequel ils sont traités dans n'est pas spécifié. »

générateur de séquence

L'élément générateur de séquence est utilisé pour définir un générateur utilisant une base de données séquence pour générer des identifiants. Il correspond à l'annotation @SequenceGenerator (voir chapitre 4) et peut être utilisé pour définir un nouveau générateur ou remplacer un générateur du même nom que celui défini par une annotation @SequenceGenerator dans n'importe quelle classe de l'unité de persistance. Il peut être spécifié soit au niveau global comme un sous-élément de mappages d'entités, au niveau de l'entité en tant que sous-élément d'entité, ou au niveau du champ ou de la propriété level en tant que sous-élément de l'élément de mappage d'id.

Les attributs du générateur de séquence correspondent exactement aux éléments du Annotation @SequenceGenerator. Référencement13-9 montre un exemple de définition d'une séquence Générateur.

Annonce 13-9. Définition d'un générateur de séquence

```
<entity-mappings>
    ...
    <sequence-generator name = "empGen" sequence-name = "empSeq" />
    ...
</entity-mappings>
```

[2] Il est possible, et même probable, que les fournisseurs traitent les fichiers de mappage dans l'ordre ils sont répertoriés, mais ce n'est ni obligatoire ni normalisé.

Épisode 623

table-générateur

L'élément table-generator définit un générateur qui utilise une table pour générer identifiants. Son équivalent d'annotation est l'annotation @TableGenerator (voir Chapitre 4). Cet élément peut définir un nouveau générateur ou il peut remplacer un générateur défini par une annotation @TableGenerator. Comme le générateur de séquence élément, il peut être défini dans n'importe quel élément de mappage d'entité, d'entité ou d'id.

Les attributs de table-generator correspondent également à l'annotation @TableGenerator éléments. Le Listing 13-10 montre un exemple de définition d'un générateur de séquence dans formulaire d'annotation mais le remplaçant pour être un générateur de table en XML.

Annonce 13-10. Remplacement d'un générateur de séquence avec un générateur de table

```
@Entité
Employé de classe publique {
    @SequenceGenerator (nom = "empGen")
    @Id @GeneratedValue (générateur = "empGen")
    id int privé;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    ...
    <table-generator name = "empGen" table = "ID_GEN" pk-column-value = "EmpId" />
    ...
</entity-mappings>
```

requête nommée

Les requêtes statiques ou nommées peuvent être définies à la fois sous forme d'annotation à l'aide de @NamedQuery (reportez-vous au chapitre 7) ou dans un fichier de mappage utilisant l'élément named-query. Une requête nommée élément du fichier de mappage peut également remplacer une requête existante du même nom qui a été défini comme une annotation. Cela a du sens, bien sûr, lors du remplacement d'une requête pour le remplacer uniquement par une requête ayant le même type de résultat, qu'il s'agisse d'une entité, de données ou projection de données. Sinon, tout le code qui exécute la requête et traite le les résultats ont de bonnes chances de se briser.

Épisode 624

Un élément de requête nommée peut apparaître comme un sous-élément de mappages d'entités ou comme sous-élément d'entité. Quel que soit l'endroit où il est défini, il sera saisi par son nom dans l'espace de noms de requête d'unité de persistance.

Le nom de la requête est spécifié comme attribut de l'élément de requête nommée, tandis que la chaîne de requête va dans un sous-élément de requête en son sein. N'importe lequel des énumérés

Les constantes LockModeType peuvent être incluses. N'importe quel nombre d'indices de requête peut également être fourni en tant que sous-éléments d'indication.

Référencement 13-11 montre un exemple de deux requêtes nommées, dont l'une utilise un indice qui contourne le cache.

Annonce 13-11. Requête nommée dans un fichier de mappage

```
<entity-mappings>
    ...
    <named-query name = "findEmpsWithName">
        <query> SELECT e FROM Employee e WHERE e.name LIKE: empName </query>
        <hint name = "javax.persistence.cacheRetrieveMode"
                value = "CacheRetrieveMode.BYPASS" />
    </named-query>
    <named-query name = "findEmpsWithHigherSalary">
        <query> <! [CDATA [SELECT e FROM Employee e WHERE e.salary>
        : salaire]]> </query>
    </named-query>
    ...
</entity-mappings>
```

Les chaînes de requête peuvent également être exprimées en CDATA dans l'élément de requête. Vous pouvez voir dans la liste 13-11 , que cela est utile dans les cas où la requête comprend des caractères XML tel que> qui aurait autrement besoin d'être échappé.

query-native-nommée

Le SQL natif peut également être utilisé pour les requêtes nommées en définissant un @NamedNativeQuery annotation (reportez-vous au chapitre 11) ou en spécifiant un élément de requête native fichier de mappage. Les requêtes nommées et les requêtes natives partagent le même espace de noms de requête dans l'unité de persistance, donc en utilisant soit l'élément named-query ou named-native-query fera que cette requête écrase toute requête du même nom défini dans le formulaire d'annotation.

Épisode 625

Les requêtes natives sont identiques aux requêtes nommées dans la mesure où la requête nommée native L'élément peut apparaître comme un sous-élément de mappages d'entités ou comme un sous-élément d'entité. Le nom est spécifié à l'aide de l'attribut name et la chaîne de requête utilise une requête sous-élément. Les indices sont également spécifiés de la même manière. La seule différence est que deux des attributs supplémentaires ont été ajoutés à named-native-query pour fournir la classe de résultat ou le mappage de l'ensemble de résultats.

Un cas d'utilisation pour le remplacement des requêtes est lorsque le DBA vient à vous et exige que votre requête s'exécute d'une certaine manière sur une certaine base de données. Vous pouvez laisser la requête générique JP QL pour les autres bases de données, mais il s'avère que, par exemple, la base de données Oracle peut très bien faire cette chose en utilisant la syntaxe native. En mettant cette requête dans un fichier XML spécifique à la base de données, il sera beaucoup plus facile à gérer à l'avenir. Référencement13-12 a un exemple de requête nommée vanille dans JP QL qui est remplacée par un SQL natif requete.

Annonce 13-12. Remplacer une requête JP QL avec SQL

```
@NamedQuery (nom = "findAllManagers"
                query = "SELECT e FROM Employee e WHERE e.directs N'EST PAS VIDE")
@Entité
Employé de classe publique {...}
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    ...
    <named-native-query name = "findAllManagers"
                                result-class = "examples.model.Employee">
        <requête>
            SELECT / * + FULL (m) * / e.id, e.name, e.salary,
                    e.manager_id, e.dept_id, e.address_id
            DE EMP e,
                    (SELECT DISTINCT manager_id AS id FROM emp) m
            O e.id = m.id
        </query>
    </named-native-query>
    ...
</entity-mappings>
```

Épisode 626

requête-procédure-stockée-nommée

Une procédure stockée peut être représentée par une requête nommée en définissant un @NamedStoredProcedureQuery annotation (reportez-vous au chapitre 11) ou en spécifiant un Élément named-stored-procedure-query dans un fichier de mappage. Comme les autres requêtes nommées, les requêtes de procédure stockée nommées peuvent être spécifiées comme sous-éléments de mappages d'entités ou entité. Bien qu'ils puissent partager le même espace de noms de requête d'unité de persistance avec tous les autres requêtes nommées, la méthode de création d'API d'une requête de procédure stockée nommée renvoie une instance de StoredProcedureQuery. Cela signifie qu'en pratique, ils sont différents types de requêtes nommées et vous ne pourrez pas, par exemple, remplacer un JP Requête nommée QL avec une requête de procédure stockée. Vous pouvez cependant remplacer un requête de procédure définie sous forme d'annotation avec une requête de procédure stockée définie en utilisant l'élément de requête de procédure stockée nommée.

Semblable à d'autres requêtes nommées, le nom de la requête est spécifié à l'aide du nom et n'importe quel nombre de sous-éléments d'indication peuvent être utilisés pour fournir des indicateurs de requête. Cependant, étant donné que plusieurs jeux de résultats peuvent être renvoyés à partir de requêtes de procédure stockée, au lieu des attributs result-class et result-set-mapping qui existaient sur named-native-query, plusieurs sous-éléments de mappage de résultats et de classes de résultats peut être spécifié sous l'élément named-stored-procedure-query. Bien que le schema ne l'interdit pas, en utilisant à la fois result-class et result-set-mapping les sous-éléments pour différents ensembles de résultats dans la même requête ne sont pas autorisés.

Les parties uniques d'une requête de procédure stockée sont le nom de procédure supplémentaire attribut pour spécifier le nom de la procédure stockée dans la base de données et le paramètre sous-éléments pour définir les noms et les types de paramètres. Chaque sous-élément de paramètre a un attribut de nom, de classe et de mode pour indiquer le nom du paramètre, la classe JDBC et si le paramètre est un paramètre IN, OUT, INOUT ou REF_CURSOR. Les paramètres doivent être définis dans l'ordre dans lequel ils apparaissent dans la définition de procédure stockée réelle.

Référencement 13-13 montre un exemple d'utilisation d'un élément de requête de procédure stockée nommée pour ajouter la requête de procédure stockée du chapitre 11 .

Annonce 13-13. Définition d'une requête de procédure stockée nommée

```
@NamedStoredProcedureQuery (
    name = "fetch_emp",
    procedureName = "fetch_emp",
    paramètres = {
```

Épisode 627


```
        @StoredProcedureParameter (nom = "empList", type = void.class,
                                   mode = ParameterMode.REF_CURSOR)
    },
    resultClasses = Employee.class
)
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    ...
    <named-stored-procedure-query name = "fetch_emp" procedure-name = "fetch_emp">
        <parameter name = "empList" class = "void" mode = "REF_CURSOR" />
        <result-class> Employé </result-class>
    </ named-stored-procedure-query>
    ...
</entity-mappings>
```

sql-result-set-mapping

Un mappage d'ensemble de résultats est utilisé par les requêtes natives ou les requêtes de procédure stockée pour indiquer le fournisseur de persistance comment mapper les résultats. L'élément sql-result-set-mapping correspond à l'annotation @SqlResultSetMapping. Le nom du jeu de résultats le mappage est spécifié dans l'attribut name de l'élément sql-result-set-mapping. Le résultat peut être mappé comme un ou plusieurs types d'entités, types Java non-entité, des données de projection, ou une combinaison de celles-ci. Tout comme @SqlResultSetMapping englobe tableaux de @EntityResult, @ConstructorResult et @ColumnResult, de même que L'élément sql-result-set-mapping contient plusieurs résultats d'entité, les éléments constructor-result et column-result. De même, parce que chacun @EntityResult contient un tableau de @FieldResult, l'élément de résultat d'entité peut contenir plusieurs éléments de résultat de champ. L'autre entityClass et discriminatorColumn éléments de la mappe d'annotations @EntityResult directement Attributs de classe d'entité et de colonne de discrimination de l'élément de résultat d'entité. De même, un @ConstructorResult contient un tableau de @ColumnResult, donc le le sous-élément constructeur-résultat contient un nombre arbitraire de sous-éléments de colonne, avec un attribut de classe cible pour spécifier le nom de la classe non-entité à construire.

Chaque sql-result-set-mapping peut définir un nouveau mappage ou remplacer un un du même nom qui a été défini par une annotation. Il n'est pas possible de remplacer

Épisode 628

seulement une partie du mappage de l'ensemble de résultats. Si vous remplacez une annotation, l'ensemble l'annotation sera remplacée et les composants du mappage d'ensemble de résultats définis par l'élément sql-result-set-mapping s'appliquera.

Après avoir dit tout cela à propos de la neutralisation, il n'est pas vraiment utile de remplacer une @SqlResultSetMapping car ils sont utilisés pour structurer le format de résultat à partir d'un requête de procédure native ou stockée. Comme nous l'avons mentionné précédemment, les requêtes ont tendance à être exécutées avec une certaine attente du résultat renvoyé. Les mappages d'ensemble de résultats sont généralement défini dans un fichier de mappage, car c'est aussi généralement là où la requête la définition du résultat est définie.

Référencement 13-14 montre le mappage d'ensemble de résultats DepartmentSummary que nous avons défini dans Chapitre 11 et son formulaire de fichier de mappage XML équivalent.

Annonce 13-14. Spécification d'un mappage d'ensemble de résultats

```
@SqlResultSetMapping (
    name = "DepartmentSummary",
    entités = {
        @EntityResult (entityClass = Department.class,
                       fields = @ FieldResult (name = "name", column = "DEPT_NAME")),
        @EntityResult (entityClass = Employee.class)
    },
    colonnes = {@ ColumnResult (name = "TOT_EMP"),
                @ColumnResult (nom = "AVG_SAL")}
)
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    ...
    <sql-result-set-mapping name = "DepartmentSummary">
        <entity-result entity-class = "examples.model.Department">
            <field-result name = "name" column = "DEPT_NAME" />
        </entity-result>
        <entity-result entity-class = "examples.model.Employee" />
        <column-result name = "TOT_EMP" />
```



```
        <column-result name = "AVG_SAL" />
    </sql-result-set-mapping>
    ...
</entity-mappings>
```

Classes et mappages gérés

La partie principale de chaque fichier de mappage sera généralement les classes gérées dans le unité de persistance qui sont l'entité, la superclasse mappée et les éléments intégrables et leur état et leurs relations. Chacun d'eux a sa classe spécifiée en tant que classe attribut de l'élément et son type d'accès spécifié dans un attribut d'accès. L'accès L'attribut n'est requis que lorsqu'il n'y a pas d'annotations sur la classe gérée ou lorsque metadata-complete (ou xml-mapping-metadata-complete) a été spécifié pour le classe. Si aucune de ces conditions ne s'applique et que des annotations existent sur la classe, le Le paramètre d'attribut d'accès doit correspondre à l'accès utilisé par les annotations.

Pour les entités, un attribut facultatif pouvant être mis en cache peut également être défini sur une valeur booléenne. Cette L'attribut correspond à l'annotation @Cacheable et, lorsqu'il est spécifié, remplacera le valeur de l'annotation. Comme l'annotation, elle dicte si le cache partagé est utilisé pour les instances de la classe d'entité, et n'est applicable que lorsque le mode de cache partagé (voir chapitre 14) est réglé sur l'un des modes sélectifs. L'attribut pouvant être mis en cache est hérité par les sous-classes et est remplacé par l'annotation @Cacheable sur la sous-classe, ou l'attribut pouvant être mis en cache dans l'élément de sous-classe.

Les requêtes et les générateurs peuvent être spécifiés dans un élément d'entité. Générateurs peut également être défini à l'intérieur d'un élément id dans une entité ou une superclasse mappée. Ils ont déjà décrit dans la section précédente «Requêtes et générateurs».

Les attributs

Malheureusement, le mot «attribut» est largement surchargé. Cela peut être un terme général pour un champ ou propriété dans une classe, il peut s'agir d'une partie spécifique d'un élément XML qui peut être insérée dans l'étiquette d'élément, ou il peut s'agir d'un terme générique faisant référence à une caractéristique. Tout au long de ces sections, nous y avons généralement fait référence dans le contexte du deuxième sens car nous avons beaucoup parlé des éléments XML. Dans cette section, cependant, il fait référence à la première définition d'un attribut d'état sous la forme d'un champ ou d'une propriété.

Épisode 630

L'élément attributes est un sous-élément de l'entité, mapped-superclass, et éléments intégrables. C'est un élément englobant qui regroupe tout le mapping sous-éléments pour les champs ou propriétés de la classe gérée. Parce que ce n'est qu'un élément de regroupement, il n'a pas d'annotation analogue. Il dicte les mappages sont autorisés pour chaque type de classe gérée.

Dans les éléments entity et mapped-superclass, il existe un certain nombre de mapping sous-éléments qui peuvent être spécifiés. Pour les identifiants, soit plusieurs sous-éléments d'identifiant, soit un seul sous-élément Embedded-id peut être inclus. La version basique simple, et Des sous-éléments de mappage transitoire peuvent également être spécifiés, ainsi que le plusieurs-à-un, sous-éléments d'association un-à-un, un-à-plusieurs et plusieurs-à-plusieurs. La cartographie mix est complété par les sous-éléments incorporés et élément-collection. Un L'élément intégrable n'est pas autorisé à contenir un id, un identifiant intégré ou un mappage de version sous-éléments. Ces éléments seront tous traités séparément dans leurs propres sections plus tard, mais ils ont tous une chose en commun. Chacun a un attribut de nom (dans le XML Attribut sense) qui est nécessaire pour indiquer le nom de l'attribut (dans ce cas, nous voulons dire champ ou propriété) qu'il mappe.

Un commentaire général sur le remplacement des mappages d'attributs est que le remplacement les annotations avec XML se font au niveau du nom de l'attribut (champ ou propriété). Notre algorithme s'appliquera à ces mappages car ils sont saisis par nom d'attribut, et Les remplacements XML seront appliqués uniquement par nom d'attribut. Toute la cartographie annotée les informations de l'attribut seront remplacées dès qu'un élément de mappage pour cela le nom d'attribut est défini en XML.

Il n'y a rien pour arrêter le type de mappage d'attribut défini dans le formulaire d'annotation d'être remplacé dans XML pour être un type de mappage différent. Le fournisseur est responsable uniquement pour mettre en œuvre les règles primordiales et n'empêchera probablement pas ce genre de comportement. Cela nous amène à notre deuxième commentaire sur le remplacement, à savoir que lors du remplacement annotations, vous devez utiliser le mappage XML correct et compatible. Il y a quelques cas où il peut être valide de mapper un attribut différemment en XML, mais ces les cas sont rares et principalement pour des types exceptionnels de test ou de débogage.

Par exemple, on pourrait imaginer remplacer un champ mappé sous forme d'annotation comme un mappage de base avec un mappage transitoire en XML. Ce serait tout à fait légal, mais pas forcément une bonne idée. À un moment donné, un client de l'entité pourrait en fait essayer de accéder à cet état, et s'il n'est pas persistant, le client peut être assez confus et échouent de manière curieuse et difficile à déboguer. Une propriété d'association d'adresses mappée car un mappage plusieurs à un pourrait éventuellement être remplacé pour être stocké en série en tant que

Épisode 631

blob, mais cela pourrait non seulement interrompre l'accès client, mais également se répandre pour briser d'autres zones comme les requêtes JP QL qui traversent l'adresse.

La règle de base est que les mappages doivent être remplacés principalement pour changer les informations de mappage au niveau des données. Cela devrait normalement être fait lorsque, pour Par exemple, une application est développée sur une base de données mais déployée sur une autre ou doit déployer sur plusieurs bases de données différentes en production. Dans ces cas, les mappages XML serait probablement xml-mapping-metadata-complete de toute façon, et les métadonnées XML seraient être utilisé dans son intégralité plutôt que de bricoler des bits d'annotations et des bits de XML et en essayant de garder tout droit sur plusieurs configurations de mappage XML de base de données.

les tables

La spécification de tables en XML fonctionne à peu près de la même manière que sous forme d'annotation. Les mêmes valeurs par défaut sont appliquées dans les deux cas. Il y a deux éléments pour spécifier la table informations pour une classe gérée: table et table secondaire.

table

Un élément de table peut apparaître comme un sous-élément d'entité et décrit la table que le l'entité est mappée à. Il correspond à l'annotation @Table (voir chapitre 4) et a attributs de nom, de catalogue et de schéma. Un ou plusieurs sous-éléments de contrainte unique peuvent être incluses si des contraintes de colonne uniques doivent être créées dans la table pendant génération de schéma.

Si une annotation @Table existe sur l'entité, l'élément de table remplacera le table définie par l'annotation. Le remplacement d'un tableau est généralement accompagné de les mappages remplacés de l'état persistant à la table remplacée. Annonce 13-15 montre comment une entité peut être mappée à une table différente de celle à laquelle elle est mappée par un annotation.

Annonce 13-15. Remplacer une table

```
@Entité
@Table (nom = "EMP", schéma = "HR")
Employé de classe publique {...}
```


Épisode 632

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    <table name = "EMP_REC" schema = "HR" />
    ...
</entity>
```

table secondaire

N'importe quel nombre de tables secondaires peut être ajouté à l'entité en ajoutant une ou plusieurs sous-éléments de table secondaire à l'élément entité. Cet élément correspond à l'annotation @SecondaryTable (reportez-vous au chapitre 10), et s'il est présent dans une entité élément, il remplacera toutes les tables secondaires définies dans les annotations sur la classe d'entité. L'attribut name est obligatoire, tout comme le nom est obligatoire dans

l'annotation. Les attributs de schéma et de catalogue et la contrainte unique
des sous-éléments peuvent être inclus, tout comme avec l'élément table.

Chaque table secondaire doit être jointe à la table primaire via une table principale
colonne de jointure par clé (reportez-vous au chapitre dix). L'élément primary-key-join-column est un
sous-élément de l'élément de table secondaire et correspond au
Annotation @PrimaryKeyJoinColumn. Comme pour l'annotation, cela n'est requis que si le
La colonne de clé primaire de la table secondaire est différente de celle de la table primaire.
Si la clé primaire est une clé primaire composée, plusieurs
Les éléments de la colonne primary-key-join- column peuvent être spécifiés.

Référencement 13-16 compare la spécification des tables secondaires en annotation et
Formulaire XML.

Annonce 13-16. Spécification des tables secondaires

```
@Entité
@Table (nom = "EMP")
@SecondaryTables ({
    @SecondaryTable (nom = "EMP_INFO"),
    @SecondaryTable (nom = "EMP_HIST",
                        pkJoinColumns = @ PrimaryKeyJoinColumn (nom = "EMP_ID"))
 })
```

Épisode 633


```
Employé de classe publique {
    @Id id int privé;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    <nom de la table = "EMP" />
    <nom de la table secondaire = "EMP_INFO" />
    <nom de la table secondaire = "EMP_HIST">
        <primary-key-join-column name = "EMP_ID" />
    </secondary-table>
    ...
</entity>
```

Mappages d'identificateurs

Les trois types différents de mappages d'identificateurs peuvent également être spécifiés en XML. Primordial
s'applique aux informations de configuration dans un type d'identifiant donné, mais l'identifiant
le type d'une classe gérée ne devrait presque jamais être modifié.

id

L'élément id est la méthode la plus couramment utilisée pour indiquer l'identifiant d'une entité.
Il correspond à l'annotation @Id mais encapsule également les métadonnées pertinentes
aux identifiants. Cela inclut un certain nombre de sous-éléments, dont le premier est la colonne
sous-élément. Il correspond à l'annotation @Column qui pourrait accompagner un
Annotation @Id sur le champ ou la propriété. Lorsqu'il n'est pas spécifié, le nom de la colonne par défaut
sera supposé même si une annotation @Column existe sur le champ ou la propriété. Comme nous
discuté dans la section «Attributs», c'est parce que le mappage XML de l'attribut

remplace l'ensemble du groupe de métadonnées de mappage sur le champ ou la propriété.

Un élément de valeur générée correspondant à l'annotation @GeneratedValue peut également être inclus dans l'élément id. Il est utilisé pour indiquer que l'identifiant sera avoir sa valeur automatiquement générée par le fournisseur (voir chapitre 4). Cette L'élément de valeur générée a des attributs de stratégie et de générateur qui correspondent à ceux sur l'annotation. Le générateur nommé peut être défini n'importe où dans l'unité de persistance.

Épisode 634

Les générateurs de séquence et de table peuvent également être définis dans l'élément id. C'étaient discuté dans la section «Requêtes et générateurs».

Un exemple de remplacement d'un mappage d'identifiant est de changer le générateur pour un base de données (voir l'extrait 13-17).

Annonce 13-17. Remplacement d'un générateur d'ID

```
@Entité
Employé de classe publique {
    @Id @GeneratedValue (strategy = GenerationType.TABLE, generator = "empTab")
    @TableGenerator (nom = "empTab", table = "ID_GEN")
    identifiant long privé;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    ...
    <attributs>
        <id name = "id">
            <generated-value strategy = "SEQUENCE" generator = "empSeq" />
            <sequence-generator name = "empSeq" sequence-name = "mySeq" />
        </id>
        ...
    </attributes>
</entity>
```

identifiant intégré

Un élément Embedded-id est utilisé lorsqu'une classe de clé primaire composée est utilisée comme identifiant (voir chapitre 10). Il correspond à l'annotation @EmbeddedId et est vraiment juste mapper une classe incorporée comme identificateur. Tout l'état est en fait cartographié dans l'objet incorporé, il n'y a donc que des remplacements d'attribut disponibles dans le Élément embedded-id. Comme nous le verrons dans la section «Mappages d'objets intégrés», les remplacements d'attributs permettent le mappage du même objet incorporé dans plusieurs entités. Le zéro ou plusieurs éléments de remplacement d'attribut dans la propriété ou le mappage de champ du

Épisode 635

entity fournit les remplacements locaux qui s'appliquent à la table d'entité. Référencement13-18 montre comment pour spécifier un identifiant intégré sous forme d'annotation et XML.

Annonce 13-18. Spécification d'un ID intégré

```
@Entité
Employé de classe publique {
    @EmbeddedId ID de EmployeePK privé;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    ...
    <attributs>
        <embedded-id name = "id" />
        ...
    </attributes>
</entity>
```

id-class

Une classe ID est une stratégie qui peut être utilisée pour une clé primaire composée (reportez-vous à Chapitre dix). Le sous-élément id-class d'une entité ou d'un élément mappé-superclass correspond à l'annotation @IdClass, et lorsqu'elle est spécifiée en XML, elle remplacera toute annotation @IdClass sur la classe. Le remplacement de la classe ID ne devrait normalement pas être fait dans la pratique car le code qui utilise les entités supposera généralement un classe d'identifiant.

Le nom de la classe est indiqué comme la valeur de l'attribut class de l'id-class élément, comme indiqué dans la liste 13-19 .

Annonce 13-19. Spécification d'une classe d'ID

```
@Entité
@IdClass (EmployeePK.class)
Employé de classe publique {...}
```


Épisode 636

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    ...
    <id-class = "examples.model.EmployeePK" />
    ...
</entity>
```

Mappages simples

Un mappage simple prend un attribut et le mappe à une seule colonne dans une table. le la majorité des états persistants mappés par une entité seront composés de mappages simples. Dans cette section, nous discutons des mappages de base et couvrons également les métadonnées pour la gestion des versions et les attributs transitoires.

de base

Les mappages de base ont été discutés en détail dans la première partie du livre; ils cartographient un simple

champ d'état ou propriété à une colonne de la table. L'élément de base fournit cette même capacité en XML et correspond à l'annotation @Basic. Contrairement à l'annotation @Basic (décrit au chapitre 4) rarement utilisé, l'élément de base est requis lors du mappage état persistant dans une colonne spécifique. Tout comme avec les annotations, lorsqu'un champ ou une propriété est non mappé, il sera supposé être un mappage de base et sera défini par défaut comme tel. Cette se produira si le champ ou la propriété n'est pas annoté ou n'a pas d'entrée de sous-élément nommé dans l'élément attributs.

En plus d'un nom, l'élément de base a des attributs d'extraction et facultatifs qui peuvent être utilisé pour le chargement paresseux et les options. Ils ne sont pas obligatoires et pas très utiles au niveau d'un champ ou d'une propriété. Le seul autre attribut de l'élément de base est l'accès attribut. Lorsqu'il est spécifié, cela entraînera l'accès à l'état en utilisant le mode prescrit.

Le sous-élément le plus important et le plus utile de basic est l'élément de colonne. Un de quatre autres sous-éléments peuvent éventuellement être inclus à l'intérieur de l'élément de base. Ils servent principalement à indiquer le type à utiliser lors de la communication avec le pilote JDBC. Le premier est un élément lob vide qui correspond à l'annotation @Lob. Ceci est utilisé lorsque la colonne cible est un type d'objet volumineux. Que ce soit un caractère ou un objet binaire dépend du type de champ ou de propriété.

Épisode 637

Le second est l'élément temporel qui contient l'un des éléments DATE, TIME ou TIMESTAMP comme contenu. Il correspond à l'annotation @Temporal et est utilisé pour champs de type java.util.Date ou java.util.Calendar.

Le troisième est utilisé si le champ ou la propriété est un type énuméré, et le les valeurs doivent être mappées à l'aide de chaînes au lieu d'ordinaires. Dans ce cas, l'énumération l'élément doit être utilisé. Il correspond à l'annotation @Enumerated et contient ORDINAL ou STRING comme contenu.

Enfin, si l'attribut doit être converti à l'aide d'un convertisseur, l'annotation de conversion peut être ajouté. L'élément de conversion est abordé plus loin dans la section «Convertisseurs».

Référencement 13-20 montre quelques exemples de mappages de base. En ne spécifiant pas la colonne dans le mappage d'élément de base pour le champ de nom, la colonne est remplacée par l'utilisation la colonne EMP_NAME annotée est définie par défaut sur NAME. Le champ des commentaires, cependant, est remplacé par l'utilisation de la valeur par défaut pour être mappé à la colonne COMM. C'est aussi stocké dans une colonne d'objet de grand caractère (CLOB) en raison de la présence de l'élément lob et le fait que le champ est une chaîne. Le champ type est remplacé pour être mappé au STR_TYPE, et le type énuméré de STRING est spécifié pour indiquer que le les valeurs doivent être stockées sous forme de chaînes. Le champ salaire ne contient pas non plus de métadonnées sous forme d'annotation ou XML et continue d'être mappé au nom de colonne par défaut de SALAIRE.

Annonce 13-20. Remplacer les mappages de base

```
@Entité
Employé de classe publique {
    // ...
    @Column (nom = "EMP_NAME")
    nom de chaîne privé;
    commentaires de chaîne privés;
    type EmployeeType privé;
    long salaire privé;
    // ...
}
```

Épisode 638

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    ...
    <attributs>
        ...
        <basic name = "name" />
        <basic name = "comments">
            <nom de la colonne = "COMM" />
            <lob />
        </basic>
        <nom de base = "type">
            <nom de la colonne = "STR_TYPE" />
            <enumerated> STRING </enumerated>
        </basic>
        ...
    </attributes>
</entity>
```

transitoire

Un élément transitoire marque un champ ou une propriété comme étant non persistant. C'est équivalent à l'annotation @Transient ou ayant un qualificatif transitoire sur le champ ou la propriété. Référencement 13-21 montre un exemple de la manière de définir un champ comme transitoire.

Annonce 13-21. Définition d'un champ transitoire dans un fichier de mappage

```
<entity-mappings>
    <entity class = "examples.model.Employee">
        <attributs>
            <transient name = "cacheAge" />
            ...
        </attributes>
    </entity>
</entity-mappings>
```


Épisode 639

version

L'élément version est utilisé pour mapper le champ de numéro de version dans l'entité. Il correspond à l'annotation @Version et est normalement mappé à un champ intégral

pour que le fournisseur s'incrémente lorsqu'il apporte des modifications persistantes à l'entité (reportez-vous à Chapitre 12). Le sous-élément de colonne spécifie la colonne qui stocke les données de version.

Un seul champ de version doit exister pour chaque entité. Référencement13-22 montre comment une version Le champ est spécifié dans les annotations et XML.

Annonce 13-22. Spécification de la version

```
@Entité
Employé de classe publique {
    // ...
    @Version
    version int privée;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    <entity class = "examples.model.Employee">
        <attributs>
            ...
            <nom de la version = "version" />
            ...
        </attributes>
    </entity>
    ...
</entity-mappings>
```

Mappages de relations et de collections

Comme leurs homologues d'annotation, la relation XML et les éléments de collection sont utilisé pour mapper les associations et les collections d'éléments.

Nous sommes à nouveau confrontés au problème d'un terme surchargé. Tout au long de ce chapitre, nous avons utilisé le terme «élément» pour désigner un jeton XML

Épisode 640

(la chose avec des crochets inclinés autour d'elle). Mais au chapitre5 nous avons introduit la notion de une collection d'éléments, un mappage qui désigne une collection d'objets simples ou des objets intégrables. Les sections suivantes décrivent chacune des relations et types de mappage de collection d'éléments qui existent dans XML.

plusieurs à un

Pour créer un mappage plusieurs-à-un pour un champ ou une propriété, l'élément plusieurs-à-un peut être spécifié. Cet élément correspond à l'annotation @ManyToOne et, comme la base mapping, a des attributs d'extraction, facultatifs et d'accès.

Normalement, l'entité cible est connue du fournisseur car le champ ou la propriété est presque toujours du type d'entité cible, mais sinon, l'attribut d'entité cible doit également être spécifié. Lorsque la clé étrangère plusieurs-à-un contribue à l'identificateur de l'entité et l'annotation @MapsId décrite au chapitre 10 s'applique, puis le L'attribut maps-id serait utilisé. La valeur, lorsqu'elle est requise, est le nom du attribut intégrable de la classe d'ID intégrée qui mappe la relation de clé étrangère. Si, par contre, la relation fait partie de l'identifiant mais un simple @Id serait appliqué au champ ou à la propriété de relation, l'attribut booléen id du plusieurs-à-un L'élément doit être spécifié et défini sur true.

Un élément join-column peut être spécifié comme un sous-élément du plusieurs-à-un élément lorsque le nom de la colonne est différent de celui par défaut. Si l'association est à un

entité avec une clé primaire composée, plusieurs éléments de colonne de jointure seront nécessaires.
Le mappage d'un attribut à l'aide d'un élément plusieurs-à-un entraîne les annotations de mappage qui
aurait pu être présent sur cet attribut pour être ignoré. Toutes les informations cartographiques pour
la relation, y compris les informations de la colonne de jointure, doit être spécifiée ou définie par défaut
dans l'élément XML plusieurs-à-un.

Au lieu d'une colonne de jointure, il est possible d'avoir un plusieurs-à-un ou un-à-plusieurs
relation qui utilise une table de jointure. C'est dans ce cas qu'un élément join-table peut
être spécifié comme un sous-élément de l'élément plusieurs-à-un. L'élément join-table
correspond à l'annotation @JoinTable et contient une collection de join-column
éléments qui se joignent à l'entité propriétaire, qui est normalement le côté plusieurs-à-un.
Un deuxième ensemble de colonnes de jointure joint la table de jointure au côté inverse de la relation.
Ils sont appelés éléments de colonne de jointure inverse. En l'absence d'un ou des deux,
les valeurs par défaut seront appliquées.

Épisode 641

La possibilité de mettre en cascade les opérations entre elles est unique dans les relations. La cascade
les paramètres d'une relation déterminent les opérations en cascade vers l'entité cible de
le mappage plusieurs-à-un. Pour spécifier comment la cascade doit se produire, un élément en cascade
doit être inclus en tant que sous-élément de l'élément plusieurs-à-un. Dans la cascade
élément, nous pouvons inclure notre choix de vide cascade-all, cascade-persist,
sous-éléments de fusion en cascade, de suppression en cascade, d'actualisation en cascade ou de détachement en cascade
qui dictent que les opérations données soient en cascade. Bien sûr, en spécifiant cascade
éléments en plus de l'élément cascade-tout est simplement redondant.

Maintenant, nous arrivons à une exception à la règle que nous avons donnée plus tôt lorsque nous avons dit que
le remplacement des mappages concerne généralement les remplacements de données physiques. Quand cela vient à
relations, il y a des moments où vous voudrez tester les performances d'un
et aimerait pouvoir définir certaines relations à charger avec empressement ou paresseux.
Vous ne voudrez pas parcourir le code et devrez continuer à modifier ces paramètres
et en avant, cependant. Il serait plus pratique d'avoir les mappages que vous réglez
en XML et modifiez-les simplement si nécessaire. [3] Annonce13-23 montre le remplacement de deux
relations plusieurs à un à charger paresseusement.

Annonce 13-23. Remplacer le mode de récupération

```
@Entité
Employé de classe publique {
    // ...
    @ManyToOne
    adresse d'adresse privée;
    @ManyToOne
    @JoinColumn (nom = "MGR")
    gestionnaire d'employé privé;
    // ...
}
```

[3] Certains ont fait valoir que ces types d'exercices de réglage expliquent précisément pourquoi XML devrait être utilisé pour commencer avec.

Épisode 642

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    ...
    <attributs>
        ...
        <many-to-one name = "address" fetch = "LAZY" />
        <many-to-one name = "manager" fetch = "LAZY">
            <join-column name = "MGR" />
        </many-to-one>
        ...
    </attributes>
</entity>
```

un-à-plusieurs

Un mappage un-à-plusieurs est créé à l'aide d'un élément un-à-plusieurs. Cet élément correspond à l'annotation @OneToMany et a la même entité cible facultative, extraire et accéder aux attributs décrits dans le mappage plusieurs à un. Il a un attribut supplémentaire appelé mappé par, qui indique le champ ou la propriété du propriétaire 4) et un attribut de suppression d'orphelin pour spécifier que les orphelins les entités doivent être supprimées automatiquement (reportez-vous au chapitre 10).

Un mappage un-à-plusieurs est une association à valeur de collection, et la collection peut être une liste, une carte, un ensemble ou une collection. S'il s'agit d'une liste, les éléments peuvent être remplis dans un order en spécifiant un sous-élément order-by. Cet élément correspond au @OrderBy annotation et fera ordonner le contenu de la liste par le champ spécifique ou nom de propriété spécifié dans le contenu de l'élément. Alternativement, une liste peut avoir son la commande a persisté à l'aide d'un sous-élément de colonne de commande, qui fournit toutes les fonctionnalités de son équivalent d'annotation @OrderColumn.

Le sous-élément map- key peut être spécifié pour indiquer le nom du champ ou de la propriété à utiliser comme clé. Cet élément correspond à l'annotation @MapKey et sera par défaut le champ ou la propriété de clé primaire lorsqu'aucun n'est spécifié. Si le type de clé est une entité, un map- key- join-column peut être utilisé, alors que si la clé est un type de base, une map-key-column

Épisode 643

map- key- enumerated ou map-key-temporal peut être utilisé si le type de base est un

map-key-attribute- override peut être spécifié pour remplacer où un champ incorporé ou la propriété du type intégrable est mappée. Le sous-élément map-key-class est inclus pour indiquer le type de la clé lorsque la carte n'est pas typée de manière générique et que la clé n'est pas un champ ou une propriété de l'entité cible. L'élément map-key-convert peut être utilisé si le key est un type de base et doit être converti à l'aide d'un convertisseur. L'élément map-key-convert contient les mêmes attributs que l'élément de conversion décrit dans la section «Convertisseurs».

Une table de jointure est utilisée par défaut pour mapper une association un-à-plusieurs unidirectionnelle qui ne stocke pas de colonne de jointure dans l'entité cible. Pour utiliser ce type de cartographie, l'attribut mappé par est omis et l'élément join-table est inclus. Si la le mappage est un unidirectionnel un-à-plusieurs avec la clé étrangère dans la table cible, un ou plus de sous-éléments de colonne de jointure sont utilisés à la place de la table de jointure. La colonne de jointure les éléments s'appliquent à la table d'entités cible, mais pas à la table d'entités source (reportez-vous à Chapitre dix).

Enfin, la cascade dans la relation est spécifiée via une cascade facultative 13-24 montre un mappage bidirectionnel un-à-plusieurs, tous deux dans les annotations et XML.

Annonce 13-24. Spécification d'un mappage un-à-plusieurs

```
@Entité
Employé de classe publique {
    // ...
    @OneToMany (mappedBy = "manager")
    @Commandé par
    la liste privée <Employee> dirige;
    @ManyToOne
    gestionnaire d'employé privé;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    ...
    <attributs>
```

Épisode 644


```
        ...
        <one-to-many name = "dirige" mapped-by = "manager">
            <ordre par />
        </one-to-many>
        <many-to-one name = "manager" />
        ...
    </attributes>
</entity>
```

Un par un

Pour mapper une association un-à-un, l'élément un-à-un doit être utilisé. Cet élément 4 et a le même target-entity, fetch, optional, access, maps-id et id attributs que le plusieurs-à-un

mappage un-à-plusieurs pour faire référence à l'entité propriétaire et provoquer des entités cibles orphelines pour être automatiquement supprimé.

Un élément one-to-one peut contenir un élément join-column s'il est le propriétaire du relation ou il peut avoir plusieurs éléments de colonne de jointure si l'association est à un entité avec une clé primaire composée. Dans certains systèmes hérités, il est mappé à l'aide d'une jointure table, donc un élément join-table doit être utilisé dans ce cas.

Lorsque l'association un-à-un est jointe à l'aide des clés primaires des deux entités tables, au lieu d'utiliser les attributs maps-id ou id, l'élément one-to-one

clé primaire, plusieurs éléments de colonne de jointure de clé primaire seront présents. L'un ou l'autre des

Les éléments primary-key-join-column ou join-column peuvent être présents, mais pas les deux.

Remarque L'utilisation de colonnes de jointure de clé primaire pour une clé primaire un-à-un

maps-id a été introduit, et c'est la méthode préférée pour l'avenir.

Les classes annotées et les équivalents du fichier de mappage XML pour un mappage un-à-un
13-25 .

Épisode 645

Annonce 13-25. Association de clé primaire un-à-un

```
@Entité
Employé de classe publique {
    // ...
    @OneToOne (mappedBy = "employé")
    Parking privéEspace de stationnementEspace;
    // ...
}
@Entité
public class ParkingSpace {
    @Id
    id int privé;
    // ...
    @OneToOne @MapsId
    employé privé;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    <entity class = "examples.model.Employee">
        <attributs>
            ...
            <one-to-one name = "parkingSpace" mapped-by = "employee" />
            ...
        </attributes>
    </entity>
    <entity class = "examples.model.ParkingSpace">
        <attributs>
            ...
            <one-to-one name = "employee" maps-id = "true" />
            ...
        </attributes>
    </entity>
</entity-mappings>
```

Épisode 646

plusieurs à plusieurs

La création d'une association plusieurs-à-plusieurs se fait par l'utilisation d'un plusieurs-à-plusieurs
4)
et a les mêmes attributs facultatifs d'entité cible, d'extraction, d'accès et de mappage que
ont été décrits dans la cartographie un-à-plusieurs.

En outre, étant une association à valeur de collection comme le mappage un-à-plusieurs, il prend en charge le même ordre par, colonne de commande, clé de carte, classe de clé de carte, colonne de clé de carte, clé de carte-join-column, map-key-enumerated, map-key-temporal, map-key-attribute-override,

13-26 montre un exemple de classe d'entité et XML équivalent, avec un
Exemple de relation plusieurs-à-plusieurs.

Annonce 13-26. Annotations de mappage plusieurs-à-plusieurs et XML

```
@Entité
Employé de classe publique {
    // ...
    @Plusieurs à plusieurs
    @MapKey (nom = "nom")
    @JoinTable (nom = "EMP_PROJ",
            joinColumns = @ JoinColumn (nom = "EMP_ID"),
            inverseJoinColumns = @ JoinColumn (name = "PROJ_ID"))
    projets Map <String, Project> privés;
    // ...
}

@Entité
Projet de classe publique {
    // ...
    nom de chaîne privé;
    @ManyToMany (mappedBy = "projects") Collection privée <Employee>
des employés;
    // ...
}
```


Épisode 647

Voici un extrait de code orm.xml:

```
<entity-mappings>
    <entity class = "examples.model.Employee">
        <attributs>
            ...
            <many-to-many name = "projets">
                <map-key name = "name" />
                <join-table name = "EMP_PRJ">
                    <join-column name = "EMP_ID" />
                    <inverse-join-column name = "PROJ_ID" />
                </join-table>
```

```
                </many-to-many>
                ...
            </attributes>
        </entity>
        <entity class = "examples.model.Project">
            <attributs>
                ...
                <many-to-many name = "employee" mapped-by = "projects" />
                ...
            </attributes>
        </entity>
    </entity-mappings>
```

élément-collection

Une collection d'objets de base ou incorporables est mappée à l'aide d'une collection d'éléments élément, qui correspond à l'annotation @ElementCollection (reportez-vous au chapitre 5) et a les mêmes attributs facultatifs d'entité cible, d'extraction et d'accès que ceux décrit dans les sections de mappage un-à-plusieurs et plusieurs-à-plusieurs.

Si la collection est une liste, l'une des colonnes order-by ou order-column peut être spécifiée comme sous-élément. Si la collection est une carte et contient des éléments incorporables comme valeurs, une clé de carte élément peut être utilisé pour indiquer qu'un champ ou une propriété dans la valeur intégrable doit être utilisé comme clé de carte. Alternativement, les différentes colonnes de clé de carte, colonne de jointure de clé de carte, map-key-enumerated, map-key-temporal, map-key-attribute-override, map-key-convert et map-key-class peuvent être utilisés comme décrit dans la section un-à-plusieurs.

Épisode 648

Les éléments incorporables peuvent contenir des mappages de base ainsi que des relations, afin de remplacer les colonnes et joindre les colonnes auxquelles les objets incorporables sont mappés, le Les sous-éléments de remplacement d'attribut et de remplacement d'association sont utilisés.

Si la valeur est un type de base, le sous-élément de colonne - avec la possibilité de l'un des sous-éléments temporels, lob, énumérés ou convertis - peuvent être inclus. Tout cela se réfère aux valeurs de base de la collection et l'élément de colonne fait référence à la colonne dans le table de collection qui stocke les valeurs.

Enfin, les collections d'éléments sont stockées dans une table de collection, donc la table de collection sous-élément sera évidemment un élément commun. Cela correspond à la @CollectionTable annotation et fait référence à la table qui stocke les objets de base ou incorporables dans le collection ainsi que les clés qui les indexent si la collection est une Map. Un exemple de La collection d'éléments de carte est celle qui stocke le nombre d'heures travaillées par rapport à un nom du projet, comme indiqué dans le listing 13-27.

Annonce 13-27. Collection d'éléments d'entiers avec des clés de chaîne

```
@Entité
Employé de classe publique {
    // ...
    @ElementCollection (targetClass = java.lang.Integer)
    @MapKeyClass (nom = "java.lang.String")
    @MapKeyColumn (nom = "PROJ_NAME")
    @Column (nom = "HOURS_WORKED")
    @CollectionTable (nom = "PROJ_TIME")
    Carte privée projectHours;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    <attributs>
        ...
        <element-collection name = "projectHours" target-class = "java.lang.Integer">
            <map-key-class name = "java.lang.String" />
            <map-key-column name = "PROJ_NAME" />
```


Épisode 649


```
            <nom de la colonne = "HOURS_WORKED" />
            <collection-table name = "PROJ_TIME" />
        </element-collection>
    </attributes>
</entity>
```

Mappages d'objets intégrés

Un objet incorporé est une classe qui dépend de son entité parente pour son identité.
Les objets incorporés sont spécifiés en XML à l'aide de l'élément incorporé et sont personnalisés
en utilisant l'élément de remplacement d'attribut.

embarqué

Un élément incorporé est utilisé pour mapper un objet incorporé contenu dans un champ
ou propriété (reportez-vous au chapitre 4). Il correspond à l'annotation @Embedded et permet
un attribut d'accès à spécifier pour déterminer si l'état doit être accédé à l'aide d'un
champ ou propriété. Étant donné que l'état persistant est mappé dans l'objet incorporé,
seuls les sous-éléments de remplacement d'attribut, de remplacement d'association et de conversion sont
autorisé dans l'élément incorporé.

Il doit y avoir une entrée de classe intégrable dans un fichier de mappage pour le
objet, ou il doit être annoté comme @Embeddable. Un exemple de remplacement d'un élément intégré
L'adresse est indiquée dans le listing 13-28.

Annonce 13-28. Mappages intégrés dans les annotations et XML

```
@Entité
Employé de classe publique {
    // ...
    @Embedded
    adresse d'adresse privée;
    // ...
}

@Embeddable
Adresse de classe publique {
    rue privée String;
```


Épisode 650

```
    ville privée de String;
    état de chaîne privé;
    zip de chaîne privé;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    <entity class = "examples.model.Employee">
        <attributs>
            ...
            <Embedded name = "adresse" />
            ...
        </attributes>
    </entity>
    <embeddable class = "examples.model.Address" />
</entity-mappings>
```

Le sous-élément de conversion est utilisé pour appliquer ou remplacer la conversion pour un champ particulier ou propriété dans l'objet incorporé. Consultez la section «Convertisseurs» pour en savoir plus sur la L'élément convert peut être utilisé pour remplacer la conversion.

remplacement d'attribut

Lorsqu'un objet incorporé est utilisé par plusieurs types d'entités, il est probable que certains des les mappages de base dans l'objet incorporé devront être remappés par un ou plusieurs des les entités (voir le chapitre 4). L'élément de remplacement d'attribut peut être spécifié en tant que sous-élément des éléments incorporés, imbriqués-id et collection d'éléments pour accueillir ce cas.

L'annotation qui correspond à l'élément de remplacement d'attribut est le Annotation @AttributeOverride. Cette annotation peut être sur la classe d'entité ou sur un champ ou propriété qui stocke un objet incorporé, une collection d'objets incorporés ou identifiant intégré. Lorsqu'une annotation @AttributeOverride est présente dans l'entité, elle être remplacé uniquement par un élément de remplacement d'attribut dans l'entrée du fichier de mappage d'entité qui spécifie le même champ ou propriété nommé. Notre algorithme précédent tient toujours si nous considérez les remplacements d'attribut comme étant indexés par le nom du champ ou de la propriété qu'ils

Épisode 651

sont primordiaux. Tous les remplacements d'annotations pour une entité sont rassemblés, et tout le XML les remplacements pour la classe sont appliqués en plus des remplacements d'annotations. S'il y a un remplacement en XML pour le même champ ou propriété nommé, il écrasera celui annoté. le les remplacements non chevauchants restants des annotations et du XML seront également appliqués.

L'élément de remplacement d'attribut stocke le nom du champ ou de la propriété dans son nom attribut et la colonne que le champ ou la propriété mappe en tant que sous-élément de colonne. Référencement 13-29 revisite le Listing 13-28 et remplace les champs state et zip du adresse intégrée.

Annonce 13-29. Utilisation des remplacements d'attributs

```
@Entité
Employé de classe publique {
    // ...
    @Embedded
    @AttributeOverrides ({
```

```
        @AttributeOverride (nom = "état", colonne = @ Colonne (nom = "PROV")),
        @AttributeOverride (name = "zip", column = @ Column (name = "PCODE"))})
    adresse d'adresse privée;
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    <attributs>
        ...
        <Embedded name = "adresse">
            <attribute-override name = "state">
                <nom de la colonne = "PROV" />
            </attribute-override>
            <attribute-override name = "zip">
                <nom de la colonne = "PCODE" />
            </attribute-override>
        </embedded>
        ...
    </attributes>
</entity>
```


Épisode 652

remplacement d'association

Les objets incorporables prennent également en charge les mappages de relations, bien que ce soit moins courant exigence. Lorsqu'une relation plusieurs-à-un ou un-à-un est présente dans un intégrable, une colonne de jointure est mappée soit explicitement, soit par défaut par l'association dans l'objet incorporé. Réutilisation du type intégrable dans une autre classe d'entité signifie qu'il est possible que la colonne de jointure doive être remappée. le élément association-override, qui correspond à @AssociationOverride annotation (reportez-vous au chapitre dix), peut être inclus en tant que sous-élément de l'élément intégré, Embedded-id et éléments de collection d'élément pour s'adapter à ce cas.

L'élément de remplacement d'association mappe le nom du champ ou de la propriété dans son l'attribut name et les colonnes de jointure auxquelles le champ ou la propriété mappe comme un ou plusieurs sous-éléments de colonne de jointure. Si le mappage remplacé utilise une table de jointure, le Le sous-élément join-table est utilisé à la place de join-column.

Les mêmes règles d'annotation de remplacement XML s'appliquent que celles décrites pour l'attribut remplace.

Référencement 13-30 revisite à nouveau notre exemple intégré, mais cette fois l'emporte sur la ville association dans l'adresse intégrée.

Annonce 13-30. Utilisation des remplacements d'association

```
@Entité
Employé de classe publique {
    // ...
    @Embedded
    @AssociationOverride (nom = "ville", joinColumns = @ JoinColumn (nom = "CITY_ID"))
    adresse d'adresse privée;
    // ...
}

@Embeddable
Adresse de classe publique {
    rue privée String;
    @ManyToOne
```

```
    @JoinColumn (nom = "VILLE")
    ville privée;
    // ...
}
```

640

Épisode 653

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    <attributs>
        ...
        <Embedded name = "adresse">
            <association-override name = "city">
                <join-column name = "CITY_ID" />
            </association-override>
        </embedded>
        ...
    </attributes>
</entity>
```

Mappages d'héritage

Une hiérarchie d'héritage d'entité est mappée à l'aide de l'héritage, discriminateur-des éléments de colonne et de valeur discriminante. Si la stratégie d'héritage est modifiée, elle doit être remplacé pour toute la hiérarchie d'entités.

héritage

L'élément d'héritage est spécifié pour indiquer la racine d'une hiérarchie d'héritage. Il correspond à l'annotation @Inheritance et indique le mappage d'héritage stratégie qui doit être utilisée. Lorsqu'il est inclus dans l'élément entité, il remplacera tout stratégie d'héritage définie ou par défaut dans l'annotation @Inheritance sur le classe d'entité.

Changer la stratégie d'héritage peut avoir des répercussions qui se répercutent sur d'autres zones. Par exemple, changer une stratégie de table unique en table jointe nécessitera probablement ajouter un tableau à chacune des entités en dessous. L'exemple dans Listing13-31 remplace un hiérarchie d'entités de l'utilisation d'une table unique à l'utilisation d'une stratégie jointe.

Annonce 13-31. Remplacer une stratégie d'héritage

```
@Entité
@Table (nom = "EMP")
@Héritage
@DiscriminatorColumn (nom = "TYPE")
```

641

Épisode 654


```
Classe abstraite publique Employé {...}

@Entité
```

```
@DiscriminatorValue ("FT")
classe publique FullTimeEmployee {...}

@Entité
@DiscriminatorValue ("PT")
classe publique PartTimeEmployee {...}
```

Voici un extrait de code orm.xml:

```
<entity-mappings>
    <entity class = "examples.model.Employee">
        <nom de la table = "EMP" />
        <stratégie d'héritage = "JOINED" />
        ...
    </entity>
    <entity class = "examples.model.FullTimeEmployee">
        <nom de la table = "FT_EMP" />
        ...
    </entity>
    <entity class = "examples.model.PartTimeEmployee">
        <table name = "PT_EMP" />
        ...
    </entity>
</entity-mappings>
```

discriminateur-colonne

Les colonnes discriminantes stockent des valeurs qui différencient les sous-classes d'entités concrètes dans une hiérarchie d'héritage (reportez-vous au chapitre 10). L'élément discriminateur-colonne est un sous-élément de l'entité ou des éléments de résultat d'entité et est utilisé pour définir ou remplacer la colonne du discriminateur. Il correspond à et remplace la @DiscriminatorColumn annotation et a des attributs qui incluent le nom, le type de discriminateur, columnDefinition et length. C'est un élément vide qui n'a pas de sous-éléments.

Épisode 655

L'élément discriminator-column n'est généralement pas utilisé pour remplacer une colonne sur son propre mais en conjonction avec d'autres substitutions d'héritage et de table. Annonce 13-32 montre la spécification d'une colonne discriminante.

Annonce 13-32. Spécification d'une colonne discriminante

```
@Entité
@Héritage
@DiscriminatorColumn (nom = "TYPE")
Classe abstraite publique Employé {...}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    <héritage />
    <discriminator-column name = "TYPE" />
    ...
</ entité>
```

valeur-discriminante

Un élément de valeur discriminante est utilisé pour déclarer la valeur qui identifie le béton
sous-classe d'entité stockée dans une ligne de base de données (reportez-vous au chapitre dix). Il n'existe qu'en tant que
sous-élément de l'élément entité. La valeur du discriminateur est indiquée par le contenu de
l'élément. Il n'a ni attributs ni sous-éléments.

L'élément discriminator-value correspond à @DiscriminatorValue
l'annotation et la remplace lorsqu'elle existe sur la classe d'entité. Comme avec l'autre
remplace l'héritage, il est rarement utilisé comme un remplacement. Même lorsqu'une hiérarchie est remappée
vers une base de données ou un ensemble de tables différent, il ne sera normalement pas nécessaire de remplacer la valeur.
Référencement 13-33 montre comment spécifier une valeur de discriminateur sous forme d'annotation et XML.

Annonce 13-33. Spécification d'une colonne discriminante

```
@Entité
@DiscriminatorValue ("FT")
classe publique FullTimeEmployee étend Employee {...}
```

Épisode 656

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.FullTimeEmployee">
    <discriminator-value> FT </discriminator-value>
    ...
</ entité>
```

remplacement d'attribut et remplacement d'association

Les mappages et associations simples peuvent être remplacés par l'utilisation de l'attribut
remplacements et remplacements d'associations, mais uniquement dans le cas d'une entité qui est la sous-classe
d'une superclasse mappée. État persistant simple ou état d'association hérité de
une superclasse d'entité ne peut pas être remplacée de manière portative.

Un exemple de remplacement de deux mappages de champs persistants de nom et de salaire simples, et
une association de gestionnaire avec une clé primaire composée, est affichée dans la liste 13-34 .

Annonce 13-34. Utilisation des remplacements d'attributs et d'associations avec l'héritage

```
@MappedSuperclass
@IdClass (EmployeePK.class)
Classe abstraite publique Employé {
    @Id nom de chaîne privé;
    @Id private java.sql.Date dob;
    long salaire privé;
    @ManyToOne
    gestionnaire d'employé privé;
    // ...
}

@Entité
@Table (nom = "PT_EMP")
@AttributeOverrides ({
    @AttributeOverride (nom = "nom", colonne = @ Colonne (nom = "EMP_NAME")),
    @AttributeOverride (nom = "salaire", colonne = @ Colonne (nom = "SAL"))})
@AssociationOverride (nom = "manager",
    joinColumns = {
        @JoinColumn (name = "MGR_NAME", referencedName = "EMP_NAME"),
```

Épisode 657


```
        @JoinColumn (name = "MGR_DOB", referencedName = "DOB")})
classe publique PartTimeEmployee étend Employee {...}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.PartTimeEmployee">
    ...
    <attribute-override name = "name">
        <column name = "EMP_NAME" />
    </attribute-override>
    <attribute-override name = "salaire">
        <nom de la colonne = "SAL" />
    </attribute-override>
    <association-override name = "manager">
        <join-column name = "MGR_NAME" referenced-column-name = "EMP_NAME" />
        <join-column name = "MGR_DOB" referenced-column-name = "DOB" />
    </association-override>
    ...
</entity>
```

Événements du cycle de vie

Tous les événements du cycle de vie pouvant être associés à une méthode dans un écouteur d'entité peut également être associé directement à une méthode dans une entité ou une superclasse mappée (voir Chapitre 12). La pré-persistance, post-persistance, pré-mise à jour, post-mise à jour, pré-suppression, Les méthodes post-remove et post-load sont toutes des sous-éléments valides de l'entité ou éléments de superclasse mappés. Chacun d'eux ne peut se produire qu'une seule fois dans chaque classe. Chaque L'élément d'événement de cycle de vie remplacera toute méthode de rappel d'entité du même type d'événement qui pourrait être annoté dans la classe d'entité.

Avant que quiconque ne sort et remplace toutes ses méthodes de rappel annotées avec Remplacements XML, nous devons mentionner que le cas d'utilisation pour faire une telle chose sur, sinon complètement tombe, l'inexistant. Un exemple de spécification d'une entité La méthode de rappel dans les annotations et en XML est affichée dans la liste 13-35 .

Épisode 658

Annonce 13-35. Spécification des méthodes de rappel du cycle de vie

```
@Entité
Employé de classe publique {
    // ...
    @PrePersist
    @PostLoad
```

```
    public void initTransientState () {...}
    // ...
}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    ...
    <pre-persist method-name = "initTransientState" />
    <post-load method-name = "initTransientState" />
    ...
</entity>
```

Écouteurs d'entités

Les méthodes de rappel du cycle de vie définies sur une classe autre que la classe d'entité sont appelées écouteurs d'entité. Les sections suivantes décrivent comment configurer les écouteurs d'entité dans XML à l'aide de l'élément entity-listeners et comment exclure les écouteurs hérités et par défaut.

écouteurs d'entités

Une ou plusieurs classes d'écouteurs d'entités ordonnées peuvent être définies dans un @EntityListeners annotation sur une entité ou une superclasse mappée (reportez-vous au chapitre 12). Quand un cycle de vie se déclenche, les écouteurs qui ont des méthodes pour l'événement seront appelés dans l'ordre dans dont ils sont répertoriés. L'élément entity-listeners peut être spécifié comme un sous-élément d'une entité ou d'un élément de superclasse mappée pour accomplir exactement la même chose. Ce sera ont également pour effet de remplacer les écouteurs d'entité définis dans un @EntityListeners annotation avec celles définies dans l'élément entity-listeners.

Épisode 659

Un élément d'entité-écouteurs comprend une liste d'écouteurs d'entités ordonnés sous-éléments, dont chacun définit une classe d'entité-écouteur dans son attribut de classe. Pour chaque écouteur, les méthodes correspondant aux événements du cycle de vie doivent être indiquées comme événements de sous-élément. Les événements peuvent être un ou plusieurs événements pré-persistants, post-persistants, pré-mise à jour, post-mise à jour, pré-suppression, post-suppression et post-chargement, qui correspondent à @PrePersist, @PostPersist, @PreUpdate, @PostUpdate, @PreRemove, @PostRemove, et annotations @PostLoad, respectivement. Chacun des sous-éléments d'événements a un attribut method-name qui nomme la méthode à invoquer lorsque son événement de cycle de vie est déclenché. La même méthode peut être fournie pour plusieurs événements, mais pas plus d'un Un événement du même type peut être spécifié sur une seule classe d'écouteur.

L'élément entity-listeners peut être utilisé pour désactiver tous les écouteurs d'entité définis sur une classe ou ajoutez simplement un auditeur supplémentaire. La désactivation des écouteurs n'est pas recommandée, de bien sûr, car les écouteurs définis sur une classe ont tendance à être assez couplés à la classe elle-même, et leur désactivation pourrait introduire des bogues dans la classe ou dans le système dans son ensemble.

Référencement 13-36 montre que le fichier de mappage XML remplace les écouteurs d'entité sur la classe Employé. C'est garder les existants, mais aussi en ajouter un à la fin de l'ordre de notifier au service informatique de supprimer les comptes utilisateurs d'un employé lorsqu'il ou elle quitte l'entreprise.

Annonce 13-36. Remplacer les écouteurs d'entité

```
@Entité
@EntityListeners ({EmployeeAuditListener.class, NameValidator.class})
```

```
Employé de classe publique {...}
public class EmployeeAuditListener {
    @PostPersist
    public void employeeCreated (Employee emp) {...}
    @PostUpdate
    public void employeeUpdated (Employee emp) {...}
    @PostRemove
    public void employeeRemoved (Employee emp) {...}
}
Public class NameValidator {
    @PrePersist
    public void validateName (Employé emp) {...}
}
```


Épisode 660


```
public class EmployeeExitListener {
    public void notifyIT (Employé emp) {...}
}
```

Voici un extrait de code orm.xml:

```
<entity class = "examples.model.Employee">
    ...
    <entités-auditeurs>
        <entity-listener class = "examples.listeners.EmployeeAuditListener">
            <post-persist method-name = "employeeCreated" />
            <post-update method-name = "employeeUpdated" />
            <post-remove method-name = "employeeRemoved" />
        </entity-listener>
        <entity-listener class = "examples.listeners.NameValidator">
            <pre-persist method-name = "validateName" />
        </entity-listener>
        <entity-listener class = "examples.listeners.EmployeeExitListener">
            <post-remove method-name = "notifyIT" />
        </entity-listener>
    </entity-listeners>
    ...
</entity>
```

Notez que nous avons entièrement spécifié chacun des écouteurs de rappel d'entité en XML. Certains les fournisseurs trouveront les annotations d'événement de cycle de vie sur EmployeeAuditListener et Classes d'écouteur d'entité NameValidator, mais ce n'est pas un comportement obligatoire. Être portable, les méthodes d'événement de cycle de vie doivent être spécifiées dans chacun des écouteurs d'entité éléments.

exclude-default-listeners

L'ensemble des écouteurs d'entités par défaut qui s'appliquent à toutes les entités est défini dans le Sous-élément entity- listeners de l'élément persistence-unit-defaults (voir le section d'entités-écouteurs). Ces écouteurs peuvent être désactivés ou désactivés pour un entité ou hiérarchie d'entités en spécifiant un exclude-default-listeners vide élément dans l'entité ou l'élément mappé-superclass. Cela équivaut au

Épisode 661

@ExcludeDefaultListeners annotation, et si l'un ou l'autre est spécifié pour une classe, les écouteurs par défaut sont désactivés pour cette classe. Notez que exclude-default-listeners est un élément vide, pas un booléen. Si les écouteurs d'entité par défaut sont désactivés pour une classe par une annotation @ExcludeDefaultListeners, il n'y a actuellement aucun moyen de les réactiver via XML.

exclure-les-auditeurs-de-superclasse

Les écouteurs d'entité définis sur la superclasse d'une entité seront normalement déclenchés avant le les écouteurs d'entité définis sur la classe d'entité elle-même sont déclenchés (reportez-vous au chapitre 12). À désactiver les écouteurs définis sur une superclasse d'entité ou une superclasse mappée, un vide L'élément exclude-superclass-listeners peut être fourni à l'intérieur d'une entité ou Élément mappé-superclass. Cela désactivera les écouteurs de superclasse pour les class et toutes ses sous-classes.

L'élément exclude-superclass-listeners correspond au @ExcludeSuperclassListeners annotation et, comme les exclude-default-listeners / @ExcludeDefaultListeners paire, l'un des deux peut être spécifié afin de désactiver les écouteurs de superclasse pour l'entité ou la superclasse mappée et ses sous-classes.

Graphiques d'entités nommées

Les graphiques d'entités nommées agissent comme des plans de récupération pour les requêtes d'objet et peuvent remplacer la récupération mode des mappages de champs ou de propriétés pour les types d'entité et incorporables (voir le chapitre 11). L'élément named-entity-graph est équivalent à l'annotation @NamedEntityGraph (voir chapitre 11) et ne peut se produire qu'en tant que sous-élément de l'élément entité. Tout nommé les graphes d'entités définis en XML seront ajoutés aux graphes d'entités nommés définis dans formulaire d'annotation. Si un graphe d'entité nommé XML porte le même nom que celui défini par un annotation, la version XML remplacera la définition d'annotation.

Épisode 662

Un graphe d'entité nommée a deux attributs facultatifs: un attribut name pour déclarer son nom et un attribut booléen includeAllAttributes qui sert de raccourci formulaire pour inclure chaque champ ou propriété de l'entité. Trois sous-éléments peuvent se multiplient dans le graphe d'entité nommée:

- Un sous-élément de nœud d'attribut nommé est ajouté pour chaque champ ou propriété à récupérer et sa structure est similaire à celle du Annotation @NamedAttributeNode. Il a un attribut de nom obligatoire et les attributs de chaîne optionnels de sous-graphe et de sous-graphe-clé auxquels se référer un élément de sous-graphe.
- Le sous-élément de sous-graphe de named-entity-graph est l'analogue

à l'annotation @NamedSubgraph et est utilisé pour spécifier un type
modèle pour une entité ou un type intégrable. Il a un nom obligatoire
attribut et un attribut de classe qui n'est utilisé que lorsque le sous-graphe
est pour une classe dans une hiérarchie d'héritage. Un nœud d'attribut nommé
un sous-élément doit être spécifié pour chaque champ ou propriété à
inclus dans le sous-graphe.

• Le sous-élément subclass-subgraph de named-entity-graph peut
être spécifié de la même manière qu'un élément de sous-graphe, avec le nom et la classe
attributs et un sous-élément de nœud d'attribut nommé pour chaque champ ou
propriété à récupérer. L'élément subclass-subgraph n'est utilisé que
pour spécifier des sous-graphes pour les sous-classes de l'entité qui est la racine de
le graphe d'entité nommé.

Un exemple permettra de voir beaucoup plus facilement comment le XML se compare avec le annotation. Référencement13-37 commence par montrer une annotation du chapitre 11 qui définit un graphe d'entité nommé avec plusieurs définitions de type de sous-graphe. L'équivalent XML que utilise plusieurs sous-éléments de nœud d'attribut nommé et de sous-graphe suivant.

Annonce 13-37. Graphique d'entité nommée avec plusieurs sous-graphiques

```
@NamedEntityGraph (
    attributeNodes = {
        @NamedAttributeNode ("nom"),
        @NamedAttributeNode ("salaire"),
        @NamedAttributeNode (valeur = "adresse"),
        @NamedAttributeNode (valeur = "téléphones", subgraph = "téléphone"),
```


Épisode 663


```
        @NamedAttributeNode (value = "manager", subgraph = "namedEmp"),
        @NamedAttributeNode (value = "department", subgraph = "dept")},
    sous-graphes = {
        @NamedSubgraph (nom = "téléphone",
            attributeNodes = {
                @NamedAttributeNode ("nombre"),
                @NamedAttributeNode ("type"),
                @NamedAttributeNode (value = "employee", subgraph = "namedEmp")}),
        @NamedSubgraph (nom = "namedEmp",
            attributeNodes = {
                @NamedAttributeNode ("nom")}),
        @NamedSubgraph (nom = "dept",
            attributeNodes = {
                @NamedAttributeNode ("nom")})
    })
```

Voici un extrait de code orm.xml:

```
<named-entity-graph>
    <named-attribute-node name = "nom" />
    <named-attribute-node name = "salaire" />
    <named-attribute-node name = "adresse" />
    <named-attribute-node name = "phones", subgraph = "phone" />
    <named-attribute-node name = "manager", subgraph = "namedEmp" />
    <named-attribute-node name = "department", subgraph = "dept" />
    <subgraph name = "phone">
        <named-attribute-node name = "nombre" />
        <named-attribute-node name = "type" />
```

```
        <named-attribute-node name = "employee" subgraph = "namedEmp" />
    </subgraph>
    <subgraph name = "namedEmp">
        <named-attribute-node name = "nom" />
    </subgraph>
    <subgraph name = "dept">
        <named-attribute-node name = "nom" />
    </subgraph>
</named-entity-graph>
```


Épisode 664

Convertisseurs

Les convertisseurs sont un moyen de transformer par programme les données dans un champ mappé de base ou propriété sous une autre forme avant qu'elle ne soit enregistrée dans la base de données, puis inversez le transformation à nouveau lorsque les données sont lues de la base de données dans l'entité.

Il y a deux parties à la conversion: premièrement, définir un convertisseur et deuxièmement, l'appliquer aux champs d'entité ou aux propriétés. Cette section traite de chacun de ces éléments en XML.

convertisseur

Un convertisseur est une classe gérée qui peut être déclarée soit à l'aide de @Converter annotation ou dans le fichier de mappage à l'aide du sous-élément convertisseur de mappages d'entités. L'élément convertisseur peut être spécifié au même niveau que l'entité, intégrable, ou des éléments de superclasse mappés. Il n'a que deux attributs: classe et application automatique. L'attribut class fait référence à la classe qui implémente javax.persistence. Interface AttributeConverter <X, Y> pendant que l'option booléenne d'application automatique dicte si le convertisseur doit être automatiquement appliqué aux champs d'entité ou aux propriétés de type X dans toute l'unité de persistance. Voir le chapitre10 pour plus de détails sur l'utilisation du fonction d'application automatique.

> Remarque JPA version 2.2 a ajouté la prise en charge de l'injection de CDi dans le AttributeConverter, qui nous permettent principalement d'injecter notre réutilisable implémentation de la conversion dans AttributeConverter.

C'est indéfini[4] pour avoir plus d'une classe de convertisseur déclarée à l'aide du Annotation @Converter à appliquer automatiquement au même champ cible ou type de propriété. Cependant, un convertisseur annoté peut être remplacé en utilisant l'élément convertisseur pour déclarer une classe de convertisseur différente (non annotée) appliquée automatiquement au même type de cible. Par exemple, si nous avions une classe SecureURLConverter qui implémentait AttributeConver ter <URL, String>, nous pourrions alors remplacer l'URLConverter annoté défini dans Référencement 13-5 avec la version sécurisée en la déclarant dans un élément convertisseur, comme indiqué dans Référencement 13-38 .

[4] Le fournisseur peut en choisir un au hasard ou lancer une exception au démarrage et l'interdire tout à fait.

Épisode 665

Annonce 13-38. Déclaration d'un convertisseur appliqué automatiquement

```
<entity-mappings>
    ...
    <convertisseur class = "examples.SecureURLConverter" auto-apply = "true" />
    ...
</entity-mappings>
```

convertir

Si un convertisseur n'est pas appliqué automatiquement, il doit être explicitement appliqué à un champ ou à une propriété pour qu'il prenne effet. Un convertisseur peut être appliqué à un champ ou à une propriété soit par l'annotation avec l'annotation @Convert ou l'ajout d'un sous-élément de conversion au fichier de base ou élément de collection d'élément qui mappe le champ ou la propriété.

L'élément convert contient trois attributs. Le premier attribut, convertisseur, est utilisé pour indiquer le nom de la classe de convertisseur à appliquer. Il est utilisé à chaque conversion est explicitement appliqué à un champ ou à une propriété. Le listing 13-39 montre comment appliquer un convertisseur en un attribut d'entité à l'aide de l'élément convert.

Annonce 13-39. Application de la conversion à un attribut d'entité

```
<entity-mappings>
    ...
    <entity class = "examples.model.Employee">
        <attributs>
            ...
            <basic name = "homePage">
                <convert converter = "URLConverter" />
            </basic>
        </attributes>
    </entity>
    ...
</entity-mappings>
```

Deux autres attributs sont utilisés pour remplacer ou ajouter une conversion à un ou champ ou propriété hérité. L'un est l'attribut nom-attribut, qui contient le nom du champ d'entité ou de la propriété à remplacer ou auquel appliquer un convertisseur. L'attribut suivant

Épisode 666

est un attribut de valeur booléenne appelé disable-conversion et il est utilisé pour remplacer conversion afin de l'empêcher de se produire.

Un champ ou une propriété peut être marqué pour la conversion ou remplacé pour ne pas être converti, lorsque l'élément convert est utilisé dans une entité ou un élément incorporé. Référencement 13- 40 montre comment FTEmployee, une sousclasseemployés, peut passer outre le convertisseur qui a été appliqué à homePage dans le listing 13-39.

Annonce 13-40. Remplacement d'un convertisseur dans un attribut d'entité hérité

```
<entity-mappings>
    ...
    <entity class = "examples.model.FTEmployee">
        ...
        <convert converter = "SecureURLConverter" attribute-name = "homePage" />
    </entity>
```

...
</entity-mappings>

Résumé

Avec toutes les informations de mappage XML sous votre ceinture, vous devriez maintenant être en mesure de mapper entités utilisant des annotations, XML ou une combinaison des deux. Dans ce chapitre, nous sommes allés sur tous les éléments du fichier de mappage et les a comparés avec leurs annotations. Nous avons discuté de la façon dont chacun des éléments est utilisé, de ce qu'ils remplacent et comment ils sont remplacés. Nous les avons également utilisés dans quelques brefs exemples.

Les valeurs par défaut peuvent être spécifiées dans les fichiers de mappage à différents niveaux, à partir du niveau d'unité de persistance au niveau du fichier de mappage. Nous avons couvert ce que chacun des portées et comment elles ont été appliquées.

Le chapitre suivant montre comment empaqueter et déployer des applications qui utilisent JPA. Nous examinons également comment les fichiers de mappage XML sont référencés dans le cadre d'une unité de persistance configuration.

CHAPITRE 14

Emballage et déploiement

La configuration d'une application de persistance implique de spécifier les bits d'informations, en plus du code, que l'environnement d'exécution ou la plateforme de persistance peut require pour que le code fonctionne comme une application d'exécution. Moyens d'emballage assembler toutes les pièces d'une manière qui a du sens et peut être correctement interprétée et utilisé par l'infrastructure lorsque l'application est déployée dans une application serveur ou exécuté dans une JVM autonome. Le déploiement est le processus d'obtention de l'application dans un environnement d'exécution et l'exécuter.

On pourrait afficher les métadonnées de mappage dans le cadre de la configuration globale de une application, mais nous ne couvrirons pas cela dans ce chapitre car il a déjà été discuté dans les chapitres précédents. Dans ce chapitre, nous aborderons les principaux fichier de configuration de persistance d'exécution, persistence.xml, qui définit la persistance unités. Nous détaillerons comment spécifier les différents éléments de ce fichier, lorsque ils sont nécessaires et quelles devraient être les valeurs.

Une fois l'unité de persistance configurée, nous emballerons une unité de persistance avec quelques-unes des unités de déploiement les plus courantes, telles que les archives EJB, les archives Web, et les archives d'application dans un serveur Java EE. Le package résultant sera alors déployable dans un serveur d'applications conforme. Nous passerons également à travers l'emballage et règles de déploiement pour les applications Java SE.

La génération de schéma est le processus de génération des tables de schéma vers lesquelles le les entités sont mappées. Nous listerons les propriétés qui activent la génération de schéma et

décrire les différentes formes qu'il peut prendre, comme la création de tables dans la base de données ou générer du DDL dans des fichiers de script. Nous décrirons ensuite toutes les annotations qui jouent un rôle dans ce qui est généré.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_14

Épisode 668

Configuration des unités de persistance

L'unité de persistance est l'unité principale de la configuration d'exécution. Il définit le diverses informations que le fournisseur a besoin de connaître pour gérer le classes persistantes pendant l'exécution du programme et est configuré dans une persistance. xml fichier. Il peut y avoir un ou plusieurs fichiers persistence.xml dans une application, et chaque Le fichier persistence.xml peut définir plusieurs unités de persistance. Il y aura le plus souvent un seul, cependant. Puisqu'il y a une EntityManagerFactory pour chaque unité de persistance, vous pouvez considérer la configuration de l'unité de persistance comme la configuration du usine pour cette unité de persistance.

Un fichier de configuration commun contribue grandement à normaliser le runtime configuration, et le fichier persistence.xml offre exactement cela. Alors que certains fournisseurs peut encore nécessiter un fichier de configuration supplémentaire spécifique au fournisseur, la plupart prennent en charge leurs propriétés spécifiées dans la section des propriétés (décrite dans le Section «Ajout des propriétés du fournisseur») du fichier persistence.xml.

Le fichier persistence.xml est la première étape de la configuration d'une unité de persistance. Tous les les informations requises pour l'unité de persistance doivent être spécifiées dans la persistance. xml fichier. Une fois qu'une stratégie de packaging a été choisie, le fichier persistence.xml doit être placé dans le répertoire META-INF de l'archive choisie.

Chaque unité de persistance est définie par un élément d'unité de persistance dans le fichier persistence.xml. Toutes les informations pour cette unité de persistance sont incluses dans cet élément. Les sections suivantes décrivent les métadonnées qu'une unité de persistance peut définir lors du déploiement sur un serveur Java EE.

Nom de l'unité de persistance

Chaque unité de persistance doit avoir un nom qui l'identifie de manière unique dans le cadre de son emballage. Nous discuterons des différentes options d'emballage plus tard, mais en général, si une unité de persistance est définie dans un module Java EE, il ne doit pas y avoir d'autre unité de persistance du même nom dans ce module. Par exemple, si une unité de persistance named EmployeeService est défini dans un JAR EJB nommé emp_ejb.jar, il devrait ne pas être une autre unité de persistance nommée EmployeeService dans emp_ejb.jar. Il peut être des unités de persistance nommées EmployeeService dans un module Web ou même dans un autre EJB module dans l'application, cependant.

Épisode 669

Remarque En ce qui concerne l'empaquetage Jpa, nous recommandons que les unités de persistance

être emballés dans des fichiers Jar séparés pour les rendre plus accessibles et réutilisables.

Nous avons vu dans certains des exemples des chapitres précédents que le nom du
l'unité de persistance est simplement un attribut de l'élément unité de persistance, comme dans l'exemple suivant:

```
<persistence-unit name = "EmployeeService" />
```

Cet élément d'unité de persistance vide est la définition d'unité de persistance minimale.
C'est peut-être tout ce qui est nécessaire si le serveur utilise par défaut les informations restantes, mais pas toutes
les serveurs le feront. Certains peuvent nécessiter la présence d'autres métadonnées d'unité de persistance,
telle que la source de données à accéder.

Type de transaction

La fabrique utilisée pour créer des gestionnaires d'entités pour une unité de persistance donnée
générer des gestionnaires d'entités d'un type transactionnel spécifique. Nous sommes entrés dans les détails dans
Chapitre 6 sur les différents types de gestionnaires d'entités, et l'une des choses que nous avons vu était
que chaque gestionnaire d'entité doit utiliser JTA ou des transactions locales aux ressources. Normalement,
lors de l'exécution dans un environnement de serveur géré, le mécanisme de transaction JTA est
utilisé. C'est le type de transaction par défaut qu'un serveur assumera quand aucun n'est spécifié
pour une unité de persistance et est généralement la seule dont la plupart des applications auront besoin,
ainsi, en pratique, le type de transaction n'aura pas besoin d'être spécifié très souvent.

Si la source de données est requise par le serveur, comme cela sera souvent le cas, des données compatibles JTA
la source doit être fournie (voir la section «Source de données»). Spécifier une source de données qui
n'est pas compatible JTA peut effectivement fonctionner dans certains cas, mais les opérations de base de données
ne pas participer à la transaction JTA globale ou être nécessairement atomique par rapport à
cette transaction.

Dans des situations telles que celles décrites au chapitre 6, lorsque vous souhaitez utiliser la ressource-
transactions locales au lieu de JTA, l'attribut de type transaction de la persistance
l'élément unit est utilisé pour déclarer explicitement le type de transaction de RESOURCE_LOCAL ou JTA,
comme dans l'exemple suivant:

```
<persistence-unit name = "EmployeeService"
                                transaction-type = "RESOURCE_LOCAL" />
```


Épisode 670

Ici, nous remplaçons le type de transaction JTA par défaut pour qu'il soit local à la ressource, donc
tous les gestionnaires d'entités créés dans l'unité de persistance EmployeeService doivent utiliser le
Interface EntityTransaction pour contrôler les transactions.

Fournisseur de persistance

L'API Java Persistence possède une interface SPI (Service Provider Interface) enfichable qui permet à tout
Serveur Java EE compatible pour communiquer avec tout fournisseur de persistance JPA conforme
la mise en oeuvre. Cependant, les serveurs ont normalement un fournisseur par défaut, natif du
serveur, ce qui signifie qu'il est implémenté par le même fournisseur ou livré avec le serveur.
Dans la plupart des cas, ce fournisseur par défaut sera utilisé par le serveur, et aucune métadonnée spéciale
sera nécessaire de le spécifier explicitement.

Pour passer à un autre fournisseur, la classe fournie par le fournisseur qui implémente
l'interface javax.persistence.spi.PersistenceProvider doit être répertoriée dans le
élément de fournisseur. Le listing 14-1 montre une simple unité de persistance qui définit explicitement
la classe de fournisseur EclipseLink. La seule exigence est que les JAR du fournisseur soient sur le
chemin de classe du serveur ou de l'application et accessible à l'application en cours d'exécution lors du déploiement
temps. L'élément d'en-tête de persistance complet est également inclus dans le Listing 14-1 , mais sera

ne pas être inclus dans les exemples XML suivants.

Annonce 14-1. Spécification d'un fournisseur de persistance

```
<persistence xmlns = "http://xmlns.jcp.org/xml/ns/persistence"
             xmlns: xsi = "http://www.w3.org/2001/XMLSchema-instance"
             xsi: schemaLocation = "http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd "
             version = "2.2">
    <persistence-unit name = "EmployeeService">
        <provider> org.eclipse.persistence.jpa.PersistenceProvider </provider>
    </persistence-unit>
</persistence>
```


Épisode 671

La source de données

Une partie fondamentale des métadonnées de l'unité de persistance est la description de l'endroit où le fournisseur doit obtenir des connexions à la base de données pour lire et écrire des données d'entité. La cible la base de données est spécifiée en termes de nom d'une source de données JDBC qui se trouve dans le serveur Java Espace JNDI (Naming and Directory Interface). Cette source de données doit être accessible globalement car le fournisseur y accède lorsque l'application de persistance est déployée.

Le cas typique est que les transactions JTA sont utilisées, donc c'est dans le jta-data-source élément que le nom de la source de données JTA doit être spécifié. De même, si le le type de transaction de l'unité de persistance est la ressource locale, la source de données non jta l'élément doit être utilisé.

Bien que JPA définisse les éléments standard dans lesquels spécifier les noms de source de données, il ne dicte pas le format. Dans le passé, une source de données était mise à disposition dans JNDI par en cours de configuration dans un fichier de configuration ou une console de gestion spécifique au serveur. le nom n'était pas officiellement portable mais dans la pratique, ils étaient généralement de la forme jdbc / SomeDataSource. Référencement14-2 montre comment une source de données serait spécifiée à l'aide d'un nom JNDI à l'échelle de l'application. Cet exemple suppose que le fournisseur est défini par défaut.

Liste 14-2. Spécification de la source de données JTA

```
<persistence-unit name = "EmployeeService">
    <jta-data-source> java: app / jdbc / EmployeeDS </jta-data-source>
</persistence-unit>
```

NAMESPACES JAVA EE

de nombreuses applications utilisent l'approche de dénomination à l'ancienne qui suppose une portée du composant name (par exemple, jdbc / SomeDataSource), mais à partir de Java ee 6, trois nouveaux espaces de noms existent pour permettre aux noms de faire référence à la portée globale, d'application ou de module. En utilisant le correspondant préfixes d'espace de noms standard de java: global, java: app ou java: module, une ressource peut être mis à la disposition d'autres composants dans une portée plus large que le simple composant, et le name serait portable dans les implémentations de conteneurs.

Nous utiliserons l'espace de noms de l'application dans nos exemples car nous pensons à l'application portée comme étant la portée la plus utile et la plus raisonnable pour créer une source de données

disponible.

659

Épisode 672

Notez que la spécification Java ee définit six ressources par défaut, que le produit fournit dans sa configuration par défaut. Nous pouvons configurer une ressource par défaut Java ee fournisseur en liant le nom Jndi de la ressource par défaut au nom Jndi du ressource configurée.

À partir de Java EE 7, les conteneurs fournissent une source de données par défaut (disponible au nom JNDI java: comp / DefaultDataSource), et si le fournisseur est une implémentation native pour le serveur, il peut utiliser cette valeur par défaut. Dans d'autres cas, la source de données devra être spécifiée.

Le tableau 14-1 présente les ressources par défaut de Java EE.

Tableau 14-1. *Ressources par défaut de Java EE*

Classe de ressources	Nom JNDI Java EE
javax.sql.DataSource	java: comp / DefaultDataSource
javax.enterprise.concurrent. ContextService	java: comp / DefaultContextService
javax.enterprise.concurrent. ManagedExecutorService	java: comp / DefaultManagedExecutorService
javax.enterprise.concurrent. ManagedScheduledExecutorService	java: comp / DefaultManagedScheduledExecutorService
javax.enterprise.concurrent. ManagedThreadFactory	java: comp / DefaultManagedThreadFactory
javax.jms.ConnectionFactory	java: comp / DefaultJMSConnectionFactory

Le lien suivant contient les composants du schéma XML pour le schéma Java EE 8:

http://xmlns.jcp.org/xml/ns/javaee/ namespace.

Considérez que de nombreuses API requises par la plate-forme Java EE 8 sont incluses dans Java Platform SE 8 et sont donc disponibles pour les applications Java EE.

660

Épisode 673

Dans Java EE 8, l'API JNDI fournit des fonctionnalités de dénomination et de répertoire. Ce sera permettre aux applications d'accéder à plusieurs services de dénomination et d'annuaire, tels que LDAP, DNS et NIS.

L'API JNDI fournit aux applications des méthodes pour exécuter un répertoire standard opérations, comme par exemple l'association d'attributs à des objets et la recherche d'objets

en utilisant leurs attributs.

Un composant Java EE 8 pourra également localiser son contexte de dénomination d'environnement en en utilisant les interfaces JNDI.

Un composant peut:

- Créez un objet javax.naming.InitialContext.
- Recherchez le contexte de dénomination de l'environnement dans InitialContext sous le nom java: comp / env.

L'environnement de dénomination d'un composant est stocké directement dans la dénomination de l'environnement contexte ou dans l'un de ses sous-contextes directs ou indirects.

Certains fournisseurs offrent une lecture haute performance via des connexions de base de données ne sont pas associés à la transaction JTA en cours. Les résultats de la requête sont ensuite renvoyés et rendu conforme au contenu du contexte de persistance. Cela améliore le évolutivité de l'application car la connexion à la base de données n'est pas inscrite dans la transaction JTA jusqu'à ce qu'elle soit absolument nécessaire, généralement au moment de la validation. Pour activer ces types de lectures évolutives, la valeur de l'élément non-jta-data-source serait être fourni en plus de l'élément jta-data-source. Un exemple de spécification de ces deux est dans l'extrait 14-3.

Annonce 14-3. Spécification de sources de données JTA et non-JTA

```
<persistence-unit name = "EmployeeService">
    <jta-data-source> java: app / jdbc / EmployeeDS </jta-data-source>
    <non-jta-data-source> java: app / jdbc / NonTxEmployeeDS </ non-jta-data- source>
</persistence-unit>
```

Notez que EmployeeDS est une source de données régulièrement configurée qui accède au base de données des employés, mais NonTxEmployeeDS est une source de données distincte configurée pour accéder la même base de données d'employés mais pas être enrôlé dans les transactions JTA.

Épisode 674

Fichiers de mappage

Au chapitre 13 , nous avons utilisé des fichiers de mappage XML pour fournir des métadonnées de mappage. Tout ou partie du les métadonnées de mappage pour l'unité de persistance peuvent être spécifiées dans les fichiers de mappage. L'Union de tous les fichiers de mappage (et les annotations en l'absence de xml-mapping-metadata-complete) seront les métadonnées appliquées à l'unité de persistance.

Vous vous demandez peut-être pourquoi plusieurs fichiers de mappage pourraient être utiles. Il y a en fait de nombreux cas d'utilisation de plus d'un fichier de mappage dans une seule unité de persistance, mais se résume vraiment à la préférence et au processus. Par exemple, vous souhaiterez peut-être définir tous les artefacts au niveau de l'unité de persistance dans un fichier et toutes les métadonnées d'entité dans un autre fichier. Dans un autre cas, il peut être judicieux pour vous de regrouper toutes les requêtes dans un fichier pour les isoler du reste des mappages de bases de données physiques.

Peut-être convient-il au processus de développement d'avoir même un fichier pour chaque entité, soit pour les découpler les uns des autres ou pour réduire les conflits résultant du contrôle de version et système de gestion de la configuration. Cela peut être un choix populaire pour une équipe qui travailler sur différentes entités au sein de la même unité de persistance. Chacun peut vouloir changer les mappages pour une entité particulière sans gêner les autres membres de l'équipe qui modifient d'autres entités. Bien sûr, cela doit être négocié soigneusement quand il sont vraiment des dépendances entre les entités telles que les relations ou les objets incorporés.

Il est logique de regrouper les métadonnées d'entité lorsque les relations entre ils ne sont pas statiques ou lorsque le modèle d'objet peut changer. En règle générale, s'il y a couplage fort dans le modèle objet, le couplage doit être pris en compte dans la cartographie

modèle de configuration.

Certains pourraient simplement préférer avoir un seul fichier de mappage avec toutes les métadonnées contenues à l'intérieur. Il s'agit certainement d'un modèle de déploiement plus simple et facilite le packaging. Une prise en charge intégrée est disponible pour ceux qui souhaitent limiter leurs métadonnées à un fichier unique et prêt à le nommer orm.xml. Si un fichier de mappage nommé orm.xml existe dans un Répertoire META-INF sur le chemin de classe, par exemple à côté du fichier persistence.xml, il n'a pas besoin d'être explicitement répertorié. Le fournisseur recherchera automatiquement un tel fichier et utilisez-le s'il en existe un. Mappage de fichiers nommés différemment ou dans un autre l'emplacement doit être répertorié dans les éléments du fichier de mappage du fichier persistence.xml.

Les fichiers de mappage répertoriés dans les éléments du fichier de mappage sont chargés en tant que ressources Java (en utilisant méthodes telles que ClassLoader.getResource (), par exemple) à partir du chemin de classe, doit être spécifié de la même manière que toute autre ressource Java prévue être chargé comme tel. Le composant d'emplacement de répertoire suivi du nom de fichier de

Épisode 675

le fichier de mappage entraînera sa recherche, son chargement et son traitement au moment du déploiement. Par exemple, si nous mettons toutes nos métadonnées d'unité de persistance dans META-INF / orm.xml, toutes nos requêtes dans META-INF / employee_service_queries.xml, et toutes nos entités dans META-INF / employee_ service_entities.xml, nous devrions nous retrouver avec le niveau d'unité de persistance définition indiquée dans la liste 14-4. N'oubliez pas que nous n'avons pas besoin de spécifier le META-INF / orm. xml car il sera trouvé et traité par défaut. Les autres fichiers de mappage pourraient être dans n'importe quel répertoire, pas nécessairement uniquement le répertoire META-INF. Nous les mettons dans META-INF juste pour les conserver avec le fichier orm.xml.

Annonce 14-4. Spécification des fichiers de mappage

```
<persistence-unit name = "EmployeeService">
    <jta-data-source> java: app / jdbc / EmployeeDS </jta-data-source>
    <mapping-file> META-INF / employee_service_queries.xml </mapping-file>
    <mapping-file> META-INF / employee_service_entities.xml </mapping-file>
</persistence-unit>
```

Classes gérées

Les classes gérées sont toutes les classes qui doivent être traitées et prises en compte dans un unité de persistance, y compris les entités, les superclasses mappées, les éléments incorporables et le convertisseur Des classes. Les déploiements typiques placeront toutes les entités et autres classes gérées dans un JAR unique, avec le fichier persistence.xml dans le répertoire META-INF et un ou plusieurs les fichiers de mappage sont également ajoutés lorsque le mappage XML est utilisé. Le processus de déploiement est optimisé pour ces types de scénarios de déploiement afin de minimiser la quantité de métadonnées qu'un déployeur doit préciser.

L'ensemble des entités, des superclasses mappées, des objets incorporés et des classes de convertisseur qui sera gérée dans une unité de persistance particulière est déterminée par le fournisseur lorsque il traite l'unité de persistance. Au moment du déploiement, il peut obtenir des classes gérées de l'une des quatre sources. Une classe gérée sera incluse si elle fait partie des éléments suivants:

- *Classes locales:* les classes annotées de l'unité de déploiement dans lesquelles son fichier persistence.xml a été empaqueté.
- *Classes dans les fichiers de mappage:* les classes qui ont des entrées de mappage dans un Fichier de mappage XML.

Épisode 676

- *Classes explicitement répertoriées:* les classes répertoriées comme éléments de classe dans le fichier persistence.xml.
- *JAR supplémentaires des classes gérées:* les classes annotées dans un nommé JAR répertorié dans un élément jar-file du fichier persistence.xml.

En tant que déployeur, vous pouvez choisir d'utiliser l'un ou une combinaison de ces mécanismes pour que vos classes gérées soient incluses dans l'unité de persistance. Nous discuterons chacun à son tour.

Cours locaux

La première catégorie de cours incluse est celle qui est la plus simple et qui sera probablement être utilisé le plus souvent. Nous appelons ces classes des classes locales car elles sont locales au unité de déploiement. Lorsqu'un JAR est déployé avec un fichier persistence.xml dans META-INF répertoire, ce JAR sera recherché pour toutes les classes annotées avec @Entity, @ MappedSuperclass, @Embeddable ou @Converter. Cela sera vrai pour divers types de unités de déploiement que nous décrirons plus en détail plus loin dans le chapitre.

Cette méthode est clairement le moyen le plus simple de faire inclure une classe car tout il faut mettre les classes annotées dans un JAR et ajouter la persistance. xml dans le répertoire META-INF du JAR. Le prestataire se chargera de passer par les classes et trouver les entités. D'autres classes peuvent également être placées dans le JAR avec le entités et n'aura aucun effet sur le processus de recherche, sauf peut-être potentiellement ralentir le processus de recherche s'il existe de nombreuses classes de ce type.

Classes dans les fichiers de mappage

Toute classe qui a une entrée dans un fichier de mappage sera également considérée comme une classe gérée dans l'unité de persistance. Il ne doit être nommé que dans une entité, mapped-superclass, élément intégrable ou convertisseur dans l'un des fichiers de mappage. L'ensemble de toutes les classes de tous les fichiers de mappage répertoriés (y compris le fichier orm.xml traité implicitement) sera ajouté à l'ensemble des classes gérées dans l'unité de persistance. Rien de spécial ne doit être fait à part s'assurer que les classes nommées dans un fichier de mappage sont sur le chemin des classes de l'unité en cours de déploiement. S'ils se trouvent dans l'archive des composants déployés, ils probablement déjà sur le chemin des classes. S'ils ne le sont pas, ils doivent être explicitement inclus dans le classpath tout comme ceux explicitement listés (voir la section suivante «Explicitly Listed Section Classes »).

Épisode 677

Classes explicitement répertoriées

Lorsque l'unité de persistance est petite ou lorsqu'il n'y a pas un grand nombre d'entités, nous peut vouloir lister les classes explicitement dans les éléments de classe dans le fichier persistence.xml. Cette entraînera l'ajout des classes répertoriées à l'unité de persistance.

Puisqu'une classe locale à l'unité de déploiement sera déjà incluse, nous n'avons pas besoin pour le lister dans un élément de classe. La liste explicite des classes est vraiment utile dans trois cas principaux.

Le premier est lorsqu'il y a des classes supplémentaires qui ne sont pas locales à l'unité de déploiement POT. Par exemple, il existe une classe d'objets incorporée dans un autre JAR que nous voulons

utiliser dans une entité de notre unité de persistance. Nous listerions la classe pleinement qualifiée dans la classe élément dans le fichier persistence.xml. Nous devons également nous assurer que le JAR ou le répertoire qui contient la classe est sur le chemin de classe du composant déployé, par exemple, par en l'ajoutant au chemin de classe du manifeste du JAR de déploiement.

Dans le second cas, nous voulons exclure une ou plusieurs classes qui peuvent être annotées en tant que entité. Même si la classe peut être annotée avec @Entity, nous ne voulons pas qu'elle soit traitée en tant qu'entité dans ce contexte de déploiement particulier. Par exemple, il peut être utilisé comme transfert objet et doivent faire partie de l'unité de déploiement. Dans ce cas, nous devons utiliser un spécial élément appelé exclude-unlisted-classes dans le fichier persistence.xml, qui désactive classes locales d'être ajoutées à l'unité de persistance. Lorsque exclude-unlisted-classes est utilisé, aucune des classes de la catégorie des classes locales décrite précédemment ne sera incluse.

Notez qu'il y avait un bogue dans le schéma Jpa 1.0 persistence_1_0.xsd qui la valeur par défaut de l'élément exclude-unlisted-classes était false. cela signifiait qu'une valeur true devait être explicitement incluse comme contenu, tel comme <exclude-unlisted-classes> true <exclude-unlisted-classes />, au lieu de pouvoir simplement inclure l'élément vide pour signifier que seul le les classes répertoriées dans les éléments <class> doivent être considérées comme des entités. Certains les fournisseurs ont en fait contourné ce bogue en ne validant pas la persistance. xml par rapport au schéma, mais pour être portable dans Jpa 1.0, vous devez le définir explicitement à true lorsque vous souhaitez exclure les classes non répertoriées. le bogue a été corrigé dans le schéma Jpa 2.0 persistence_2_0.xsd. notez que Jpa 2.2 sera bien sûr utilisez le fichier nommé persistence_2_2.xsd, qui se trouve sur ce site Web lien: http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/ persistence / index.html # 2.2 .

Épisode 678

Le troisième cas est celui où nous prévoyons d'exécuter l'application dans un environnement Java SE et nous listons les classes explicitement car c'est le seul moyen portable de le faire dans Java SE. Nous expliquerons le déploiement dans l'environnement non serveur Java SE plus loin dans le chapitre.

JAR supplémentaires des classes gérées

La dernière façon d'obtenir des classes gérées incluses dans l'unité de persistance est de les ajouter à un autre JAR et spécifiez le nom du JAR dans un élément jar-file dans la persistance. xml. L'élément jar-file est utilisé pour indiquer au fournisseur un JAR pouvant contenir classes annotées. Le fournisseur traitera ensuite le JAR nommé comme s'il s'agissait d'un déploiement JAR, et il recherchera toutes les classes annotées et les ajoutera à l'unité de persistance. Il recherchera même un fichier orm.xml dans le répertoire META-INF du JAR et le traitera comme s'il s'agissait d'un fichier de mappage répertorié en plus.

Tout JAR répertorié dans une entrée de fichier JAR doit se trouver sur le chemin de classe de l'unité de déploiement. Nous devons le faire manuellement, car le serveur ne le fera pas automatiquement à notre place. Encore une fois, cela peut être fait en plaçant le JAR dans le répertoire lib de EAR (ou WAR si nous déployons un WAR), ajouter le JAR au chemin de classe manifeste du déploiement unité, ou par tout autre moyen spécifique au fournisseur.

Lors de la liste d'un JAR dans un élément jar-file, il doit être répertorié par rapport au parent de le fichier JAR dans lequel se trouve le fichier META-INF / persistence.xml. Cela correspond à quoi nous placerions l'entrée classpath dans le manifeste. Par exemple, supposons que l'entreprise archive (EAR), que nous appellerons emp.ear, est structurée comme indiqué dans l'extrait 14-5 .

Annonce 14-5. Entités dans un JAR externe

emp.ear

emp-ejb.jar
META-INF / persistence.xml
lib / emp-classes.jar
exemples / modèle / Employee.class

Le contenu du fichier persistence.xml doit être comme indiqué dans la liste 14-6 , avec l'élément jar-file contenant lib / emp-classes.jar pour référencer les classes emp. jar dans le répertoire lib du fichier EAR. Cela amènerait le fournisseur à ajouter le annoté les classes trouvées dans emp-classes.jar (Employee.class) à la persistance et parce que le JAR est dans le répertoire lib de l'EAR, il serait automatiquement sur le chemin de classe de l'application.

Épisode 679

Annonce 14-6. Contenu de persistence.xml

```
<persistence-unit name = "EmployeeService">
    <jta-data-source> java: app / jdbc / EmployeeDS </jta-data-source>
    <jar-file> lib / emp-classes.jar </jar-file>
</persistence-unit>
```

Mode de cache partagé

À la fin du chapitre 12 , nous sommes allés dans quelques détails sur la mise en cache et le cache qui est partagé par tous les gestionnaires d'entités obtenus de la même fabrique de gestionnaires d'entités. dans le Section «Configuration statique du cache» de ce chapitre, nous avons décrit les options pour définition du mode de cache partagé, mais nous résumerons ici comment le mode de cache partagé element fonctionne dans le fichier persistence.xml.

L'élément shared-cache-mode est facultatif, mais lorsqu'il est spécifié, il peut être défini sur un des cinq options répertoriées dans le tableau 14-2 .

Tableau 14-2. *Options de l'élément en mode cache partagé*

Valeur	La description
NON SPÉCIFIÉ	le fournisseur choisit l'option la plus appropriée pour ce fournisseur.
TOUT	Mettez en cache toutes les entités de l'unité de persistance.
AUCUN	ne mettez en cache aucune des entités de l'unité de persistance.
DISABLE_SELECTED	Cache toutes les entités à l'exception de celles annotées avec @Cacheable (false).
ENABLE_SELECTED	Ne cache aucune entité à l'exception de celles annotées avec @Cacheable (true).

Cela n'a pas beaucoup de sens de désigner explicitement NON SPÉCIFIÉ comme l'option car cela équivaut exactement à ne pas spécifier la valeur du tout et n'offre aucun réel information. Lorsqu'il n'est pas défini, l'élément sera défini par défaut par le fournisseur sur l'un des deux les quatre autres options sont les plus pertinentes pour ce fournisseur.

Les deux options suivantes, ALL et NONE, sont des options de «balayage», ce qui signifie qu'elles affectent toutes les entités de l'unité de persistance, sans exception. Toutes les annotations @Cacheable sera ignoré lorsque l'une de ces options est définie.

Épisode 680

Les options DISABLE_SELECTED et ENABLE_SELECTED sont des options «discrétionnaires», et sont utilisés conjointement avec l'annotation @Cacheable pour déterminer les entités qui sont mis en cache et ceux qui ne le sont pas. Si la valeur par défaut de votre fournisseur est l'une des options discrétionnaires et vous finissez par utiliser l'annotation @Cacheable pour déterminer les entités sont mises en cache, vous voudrez peut-être définir explicitement cet élément sur la valeur souhaitée / attendue mode au lieu de s'appuyer sur le comportement du fournisseur par défaut. Cela évitera toute confusion qui pourrait résulter du changement de fournisseur et de l'obtention d'une valeur par défaut différente qui ne considérez les annotations @Cacheable.

Mode de validation

L'élément validation-mode dans le fichier persistence.xml détermine si la validation est activée ou non (voir la section «Activation de la validation» dans le chapitre 12). Cela pourrait être défini sur AUTO, ce qui signifie que dans l'environnement du conteneur, la validation est activée, mais lorsqu'il n'est pas exécuté dans le conteneur, la validation ne sera activée que s'il y a une validation fournisseur disponible. Le paramétrer sur CALLBACK activera la validation et supposera qu'un le fournisseur de validation se trouve sur le chemin des classes.

La valeur par défaut est AUTO, qui active la validation, donc si vous n'avez pas l'intention d'utiliser validation, nous vous recommandons de le désactiver explicitement en définissant le mode validation élément à AUCUN. Cela contournera les vérifications du fournisseur de validation et vous empêchera de encourir des frais généraux de validation si, à un moment donné plus tard, un fournisseur montre sur le chemin des classes.

Ajout de propriétés

La dernière section du fichier persistence.xml est la section des propriétés. Les propriétés élément donne à un déployeur la possibilité de fournir des paramètres standard et spécifiques au fournisseur pour l'unité de persistance. Pour garantir la compatibilité d'exécution, un fournisseur doit ignorer propriétés qu'il ne comprend pas. Bien qu'il soit utile de pouvoir utiliser le même persistence.xml sur différents fournisseurs, il est également facile de taper par erreur une propriété de manière incorrecte et la faire ignorer involontairement et silencieusement. Un exemple de l'ajout de certaines propriétés du fournisseur est indiqué dans le Listing 14-7.

Épisode 681

Annonce 14-7. Utilisation des propriétés du fournisseur

```
<persistence-unit name = "EmployeeService">
        ...
        <propriétés>
                <property name = "eclipselink.logging.level"
                                value = "FINE" />
                <nom de propriété = "eclipselink.cache.size.default"
                                valeur = "500" />
        </properties>
</persistence-unit>
```

Construction et déploiement

L'un des gros avantages d'une API de persistance standard n'est pas seulement un runtime portable API mais aussi un moyen courant de composer, d'assembler et de configurer une application qui utilise la persistance. Dans cette section, nous décrivons certains des choix populaires et pratiques qui sont utilisées pour déployer des applications activées pour la persistance.

Chemin d'accès aux classes de déploiement

Dans certaines des sections précédentes, nous disons qu'une classe ou un JAR doit être sur le déploiement classpath. Lorsque nous disons cela, nous voulons dire que le JAR doit être accessible au JAR EJB, le l'archive Web (WAR) ou l'archive d'application d'entreprise (EAR). Ceci peut être réalisé en plusieurs façons.

La première consiste à placer le JAR dans le chemin de classe manifeste de l'EJB JAR ou WAR. Cette se fait en ajoutant une entrée classpath au fichier META-INF / MANIFEST.MF dans le JAR ou GUERRE. Un ou plusieurs répertoires ou JAR peuvent être spécifiés, à condition qu'ils soient séparés par les espaces. Par exemple, l'entrée de chemin de classe du fichier manifeste suivante ajoutera l'employé / emp-classes.jar et le répertoire employee / classes vers le chemin de classe du JAR qui contient le fichier manifeste:

Chemin de classe: employé / employé / classes emp-classes.jar

Épisode 682

Une meilleure façon d'obtenir un JAR dans le chemin de classe de l'unité de déploiement consiste à placer le JAR dans le répertoire de la bibliothèque de l'EAR. Lorsqu'un JAR est dans le répertoire de la bibliothèque, il sera automatiquement être sur le chemin de classe de l'application et accessible par tous les modules déployés dans l'oreille. Par défaut, le répertoire de la bibliothèque est le répertoire de la lib dans le fichier EAR, bien que il peut être configuré pour être n'importe quel répertoire utilisant l'élément library-directory dans le descripteur de déploiement application.xml. Le fichier application.xml aurait l'air quelque chose comme le squelette montré dans l'extrait 14-8.

Annonce 14-8. Définition du répertoire de la bibliothèque d'applications

```
<application ...>
    ...
    <library-directory> myDir / jars </library-directory>
</application>
```

Lorsque vous déployez un WAR et que vous souhaitez placer un JAR supplémentaire d'entités sur le classpath, vous pouvez placer le JAR dans le répertoire WEB-INF / lib du WAR. Ce qui provoque le JAR doit être sur le chemin des classes, et les classes qu'il contient sont accessibles à toutes les classes de la guerre.

Les fournisseurs fournissent généralement leur propre moyen spécifique aux développeurs d'ajouter des classes ou JAR au chemin de classe de déploiement. Ceci est généralement proposé au niveau de l'application et pas au niveau d'un JAR ou d'une GUERRE; cependant, certains peuvent fournir les deux.

Options d'emballage

L'un des principaux objectifs de l'API Java Persistence est son intégration avec la plate-forme Java EE. Non seulement il a été intégré de manière fine, par exemple en permettant l'injection d'entité gestionnaires dans les composants Java EE, mais il a également un statut spécial dans l'application Java EE emballage. Java EE permet de prendre en charge la persistance dans une variété d'empaquetages des configurations qui offrent flexibilité et choix. Nous allons les diviser en différents

types de modules dans lesquels l'application peut être déployée: modules EJB, modules Web,
et archives de persistance.

Épisode 683

JAR EJB

La logique métier modulaire s'est traditionnellement retrouvée dans les composants de session bean,
c'est pourquoi les session beans ont été conçus avec JPA pour être le principal composant Java EE
clients de persévérance. Les beans session ont traditionnellement été déployés dans un JAR EJB,
bien que depuis Java EE 6, ils puissent également être déployés dans un WAR avec des composants Web. Pour
une discussion sur le déploiement dans un WAR, voir la section suivante.

Nous supposons que le lecteur est familiarisé avec l'empaquetage et le déploiement des composants EJB
dans un fichier JAR EJB, mais si ce n'est pas le cas, il existe de nombreux livres et ressources disponibles pour en savoir plus.

Depuis EJB 3.0, nous n'avons plus besoin d'un descripteur de déploiement ejb-jar.xml,
mais si nous choisissons d'en utiliser un, il doit être dans le répertoire META-INF. Lors de la définition d'un
unité de persistance dans un JAR EJB, le fichier persistence.xml n'est pas facultatif. Ce doit être
créé et placé dans le répertoire META-INF du JAR à côté du fichier ejb-jar.xml
descripteur de déploiement, s'il existe. Bien que l'existence de persistence.xml soit
nécessaire, le contenu peut être très rare, dans certains cas, ne comprenant que le
nom de l'unité de persistance.

Le seul vrai travail pour définir une unité de persistance est de décider où nous voulons
entités et classes gérées à résider. Nous avons plusieurs options à notre disposition. le
L'approche la plus simple consiste simplement à vider nos classes gérées dans le JAR EJB avec le
Composants EJB. Comme nous l'avons décrit dans la section «Classes locales» plus haut dans le chapitre, comme
tant que les classes gérées sont correctement annotées, elles seront automatiquement découvertes
par le fournisseur au moment du déploiement et ajouté à l'unité de persistance. Référencement14-9 montre un
exemple de fichier d'archive d'application d'entreprise qui effectue cette opération.

Annonce 14-9. Empaquetage d'entités dans un JAR EJB

```
emp.ear
    emp-ejb.jar
        META-INF / persistence.xml
        META-INF / orm.xml
        exemples / ejb / EmployeeService.class
        exemples / modèle / Employee.class
        exemples / modèle / Phone.class
        exemples / modèle / Address.class
        exemples / modèle / Department.class
        exemples / modèle / Project.class
```


Épisode 684

Dans ce cas, le fichier orm.xml contient toutes les informations de mappage que nous pourrions avoir au niveau de l'unité de persistance, comme la définition du schéma de l'unité de persistance. Dans le fichier persistence.xml, nous aurions besoin de spécifier uniquement le nom de la persistance unité et la source de données. Référencement14-10 montre le fichier persistence.xml correspondant (sans l'en-tête de l'espace de noms).

Annonce 14-10. Fichier Persistence.xml pour les entités empaquetées dans un JAR EJB

```
<persistance ...>
    <persistence-unit name = "EmployeeService">
        <jta-data-source> java: app / jdbc / EmployeeDS </jta-data-source>
    </persistence-unit>
</persistence>
```

Si nous voulions séparer les entités des composants EJB, nous pourrions les mettre dans un autre JAR et faites référence à ce JAR dans une entrée de fichier jar du fichier persistence.xml. Nous avons montré un exemple simple de cette opération dans les "JAR supplémentaires des classes gérées" section, mais nous en montrons une à nouveau ici avec un fichier orm.xml supplémentaire et des emp-mappings. fichier de mappage xml. Le Listing 14-11 montre ce que la structure et le contenu de l'EAR ressembler.

Annonce 14-11. Empaquetage d'entités dans un JAR séparé

```
emp.ear
    emp-ejb.jar
        META-INF / persistence.xml
        exemples / ejb / EmployeeService.class
    lib / emp-classes.jar
        META-INF / orm.xml
        META-INF / emp-mappings.xml
        exemples / modèle / Employee.class
        exemples / modèle / Phone.class
        exemples / modèle / Address.class
        exemples / modèle / Department.class
        exemples / modèle / Project.class
```


Le fichier emp-classes.jar contenant les entités serait sur le chemin des classes car il est dans le répertoire de la bibliothèque de l'EAR, comme décrit dans la section «Chemin d'accès aux classes de déploiement». Outre le traitement des entités présentes dans le fichier emp-classes.jar, le fichier orm.xml dans le répertoire META-INF sera également détecté et traité automatiquement. Nous devons le faire lister explicitement le fichier de mappage emp_mappings.xml supplémentaire dans un élément de fichier de mappage, cependant, pour que le fournisseur le trouve en tant que ressource. La partie unité de persistance de le fichier persistence.xml est affiché dans la liste 14-12 .

Annonce 14-12. Fichier Persistence.xml pour les entités regroupées dans un JAR distinct

```
<persistence-unit name = "EmployeeService">
    <jta-data-source> java: app / jdbc / EmployeeDS </jta-data-source>
    <mapping-file> META-INF / emp-mappings.xml </mapping-file>
    <jar-file> lib / emp-classes.jar </jar-file>
</persistence-unit>
```

Archive Web

L'archive Web est devenue le moyen de déploiement d'applications le plus populaire depuis
presque tout ce dont une application Web typique a besoin peut y être hébergé. Artefacts Web
et les frameworks, les composants métier tels que les EJB, les beans CDI et les beans Spring, ainsi que
les entités persistantes peuvent toutes être déployées dans une archive Web sans avoir besoin d'un
module de déploiement. En utilisant le WAR comme véhicule de déploiement pour les trois niveaux de code, le
Les unités EJB JAR et EAR deviennent inutiles et le WAR devient le nouvel équivalent EAR.

L'inconvénient est qu'un WAR est un peu plus complexe que le JAR EJB, et l'apprentissage
empaqueter des unités de persistance dans des archives Web nécessite de comprendre la pertinence de la
Emplacement du fichier persistence.xml. L'emplacement du fichier persistence.xml détermine le
racine de l'unité de persistance. La racine de l'unité de persistance est définie comme le JAR ou le répertoire
qui contient le répertoire META-INF, où se trouve le fichier persistence.xml. Pour
exemple, dans un JAR EJB, le fichier persistence.xml se trouve dans le répertoire META-INF
de la racine du JAR, donc la racine de l'unité de persistance est toujours la racine du JAR EJB
fichier lui-même. Dans un WAR, la racine de l'unité de persistance dépend de l'emplacement de l'unité de persistance
situé dans la GUERRE. Le choix évident est d'utiliser le répertoire WEB-INF / classes
en tant que racine, ce qui nous amènerait à placer le fichier persistence.xml dans le WEB-INF /
répertoire classes / META-INF. Toutes les classes gérées annotées enracinées dans le WEB-INF /
Le répertoire classes sera détecté et ajouté à l'unité de persistance. De même, si un fichier orm.
xml se trouve dans WEB-INF / classes / META-INF, il sera traité.

Épisode 686

Les composants Web et les composants bean sont également placés dans les classes
annuaire. Un exemple d'empaquetage d'une unité de persistance dans le répertoire WEB-INF / classes,
avec les autres classes d'application qui l'accompagnent, est présenté dans l'extrait 14-13 . Nous avons inclus
le fichier web.xml, mais il n'est plus nécessaire si des annotations sur le servlet sont utilisées.

Annonce 14-13. Empaquetage des entités dans le répertoire WEB-INF / classes

```
emp.war
    WEB-INF / web.xml
    WEB-INF / classes / META-INF / persistence.xml
    WEB-INF / classes / META-INF / orm.xml
    WEB-INF / classes / exemples / web / EmployeeServlet.class
    WEB-INF / classes / exemples / ejb / EmployeeService.class
    WEB-INF / classes / examples / model / Employee.class
    WEB-INF / classes / exemples / modèle / Phone.class
    WEB-INF / classes / exemples / modèle / Address.class
    WEB-INF / classes / exemples / modèle / Department.class
    WEB-INF / classes / exemples / modèle / Project.class
```

Le fichier persistence.xml serait spécifié exactement de la même manière que celle indiquée
dans le listing 14-10. Si nous devons ajouter un autre fichier de mappage, nous pouvons le mettre n'importe où sur
le chemin de classe de l'unité de déploiement. Nous avons juste besoin d'ajouter un élément de fichier de mappage au
fichier persistence.xml. Si, par exemple, nous mettons emp-mapping.xml dans le WEB-INF / classes /
mapping, nous ajouterions l'élément suivant au fichier persistence.xml:

```
<mapping-file> mapping / emp-mapping.xml </mapping-file>
```

Le répertoire WEB-INF / classes étant automatiquement sur le chemin de classe du WAR,
le fichier de mappage est spécifié par rapport à ce répertoire.

Archive de persistance

Si nous voulons permettre à une unité de persistance d'être partagée ou accessible par plusieurs composants,
soit dans différents modules Java EE, soit dans un seul WAR, nous devrions utiliser une persistance
archiver. Il favorise également de bons principes de conception en conservant les classes de persistance

ensemble. Nous avons vu une simple archive de persistance au chapitre 2 lorsque nous étions
démarrer et observer comment il hébergeait le fichier persistence.xml et le fichier géré
classes qui faisaient partie de l'unité de persistance qui y était définie. En plaçant une persistance

Épisode 687

archive dans le répertoire lib d'un EAR, ou dans le répertoire WEB-INF / lib d'un WAR, nous
peut le mettre à disposition de tout composant fermé qui doit fonctionner sur les entités
défini par son unité de persistance contenue.

L'archive de persistance est simple à créer et à déployer. C'est simplement un JAR qui
contient un persistence.xml dans son répertoire META-INF et les classes gérées pour le
unité de persistance définie par le fichier persistence.xml.

Référencement 14-14 montre le contenu du WAR que nous avons montré dans la liste14-13, mais
dans ce cas, il utilise une archive de persistance simple, emp-persistence.jar, pour définir le
unité de persistance que nous avons utilisée dans les exemples précédents. Cette fois, nous devons
ne mettez l'archive de persistance que dans le répertoire WEB-INF / lib, et elle sera à la fois sur le
classpath et détecté comme unité de persistance.

Annonce 14-14. Empaquetage d'entités dans une archive de persistance

```
emp.war
    WEB-INF / web.xml
    WEB-INF / classes / exemples / web / EmployeeServlet.class
    WEB-INF / classes / exemples / ejb / EmployeeService.class
    WEB-INF / lib / emp-persistence.jar
        META-INF / persistence.xml
        META-INF / orm.xml
        exemples / modèle / Employee.class
        exemples / modèle / Phone.class
        exemples / modèle / Address.class
        exemples / modèle / Department.class
        exemples / modèle / Project.class
```

Si la partie JAR emp-persistence.jar du Listing 14-14 vous semble familière, c'est parce qu'elle
est pratiquement la même que la structure JAR EJB que nous avons montrée dans la liste 14-9 sauf qu'il
est un JAR d'archive de persistance au lieu d'un JAR EJB. Nous venons de changer le nom du JAR
et a sorti les classes de session bean. Le contenu du fichier persistence.xml peut être
exactement la même chose que ce qui est montré dans la liste 14-10 . Tout comme pour les autres types d'archives,
le fichier orm.xml dans le répertoire META-INF sera automatiquement détecté et traité,
et d'autres fichiers de mappage XML peuvent être placés dans le JAR et référencés par le
fichier persistence.xml comme entrée de fichier de mappage.

Épisode 688

Les classes gérées peuvent également être stockées dans un JAR distinct externe à la persistance
archive, tout comme ils pourraient l'être dans d'autres configurations d'archive d'empaquetage. L'externe
JAR serait référencé par le fichier persistence.xml comme une entrée de fichier jar avec le même
règles de spécification telles que décrites dans les autres cas. Ce n'est ni recommandé ni

utile, cependant, puisque l'archive de persistance elle-même est déjà séparée de l'autre
classes de composants. Il y aura rarement une raison de créer encore un autre JAR pour stocker le
classes gérées, mais il peut arriver que l'autre JAR soit préexistant et que vous
besoin de le référencer car vous ne pouvez pas ou ne voulez pas mettre le fichier persistence.xml dans
le JAR préexistant.

Les archives de persistance sont en fait une manière très ordonnée de conditionner une unité de persistance. Par
en les gardant autonomes (s'ils ne font pas référence à des JAR externes de classes utilisant jar-
entrées de fichier), ils ne dépendent d'aucun autre composant de l'application mais peuvent s'asseoir
en tant que couche sous ces composants à utiliser par eux.

Portée de l'unité de persistance

Pour simplifier, nous avons parlé d'une unité de persistance au singulier. La vérité c'est que
n'importe quel nombre d'unités de persistance peut être défini dans le même fichier persistence.xml et
utilisés dans le cadre dans lequel ils ont été définis. Vous avez vu dans les sections précédentes,
lorsque nous avons discuté de la manière dont les classes gérées sont incluses dans l'unité de persistance, cette
les classes de la même archive seront traitées par défaut. Si plusieurs unités de persistance
sont définis dans le même fichier persistence.xml, et exclude-unlisted-classes n'est pas
utilisé sur l'un ou l'autre, les mêmes classes seront ajoutées à toutes les unités de persistance définies.
Cela peut être un moyen pratique d'importer et de transformer des données d'une source de données vers
un autre: simplement en lisant les entités via une unité de persistance et en effectuant le
transformation sur eux avant de les écrire via une autre unité de persistance.

Maintenant que nous avons défini et empaqueté nos unités de persistance, nous devons décrire les
règles et façons de les utiliser. Il n'y en a que quelques-uns, mais il est important de les connaître.

- La première règle est que les unités de persistance ne sont accessibles que dans le
  portée de leur définition. Nous l'avons déjà mentionné au passage
  plusieurs fois, et nous y avons fait allusion à nouveau dans le "Persistence
  Section Archive ». Nous avons dit que l'unité de persistance définie dans
  une archive de persistance au niveau EAR était accessible à tous
  composants dans le fichier EAR, et qu'une unité de persistance définie dans un
  l'archive de persistance dans un WAR n'est accessible qu'aux composants

Épisode 689

  défini dans cette GUERRE. En fait, en général, une unité de persistance définie
  d'un JAR EJB est vu par les composants EJB définis par cet EJB
  JAR et une unité de persistance définie dans un WAR ne seront visibles que par
  les composants définis dans ce WAR. Unités de persistance définies
  dans une archive de persistance qui vit dans l'EAR sera vu par tous les
  composants de l'application.

- La partie suivante est que les noms des unités de persistance doivent être uniques
  dans leur champ d'application. Par exemple, il peut y avoir une seule persistance
  unité d'un nom donné dans le même JAR EJB. De même, là
  peut être une seule unité de persistance d'un nom donné dans le même
  WAR, ainsi qu'une seule unité de persistance du même nom en tout
  les archives de persistance au niveau EAR. Il peut y avoir un nommé
  nom d'unité de persistance dans un JAR EJB et un autre qui partage son
  nom dans un autre JAR EJB, ou il peut même y avoir une unité de persistance
  avec le même nom dans un JAR EJB que dans une archive de persistance.
  Cela signifie simplement que chaque fois qu'une unité de persistance est référencée soit
  dans un @PersistenceContext, une annotation @PersistenceUnit, ou
  une méthode createEntityManagerFactory (), la plus étendue localement
  un sera utilisé.

Un dernier commentaire sur la dénomination est que simplement parce qu'il est possible d'avoir plusieurs

des unités de persistance avec le même nom dans différents espaces de noms d'archives de composants, ne veut pas dire que c'est une bonne idée. En règle générale, vous devez toujours donner de la persistance noms uniques des unités dans l'application.

En dehors du serveur

Il existe des différences évidentes entre le déploiement sur un serveur Java EE et le déploiement à un environnement d'exécution Java SE. Par exemple, certains des services de conteneur Java EE ne sera pas présent, et cela se répand dans les informations de configuration d'exécution pour un unité de persistance. Dans cette section, nous décrivons les différences à prendre en compte lors de l'emballage et le déploiement dans un environnement Java SE.

Épisode 690

Configuration de l'unité de persistance

Comme précédemment, le point de départ est la configuration de l'unité de persistance, qui est principalement la création du fichier persistence.xml. Nous décrivons les différences entre la création d'un persistence.xml pour une application Java SE et en créer un pour une application Java EE.

Type de transaction

Lors de l'exécution dans un environnement de serveur, l'attribut de type de transaction dans le l'unité de persistance est par défaut JTA. La couche de transaction JTA a été conçue pour être utilisée au sein du serveur Java EE et est destiné à être entièrement intégré et couplé au serveur Composants. Compte tenu de ce fait, JPA ne prend pas en charge l'utilisation de JTA en dehors du serveur. Certains fournisseurs peuvent offrir ce support, mais il ne peut pas être utilisé de manière portative, et de Bien sûr, il repose sur la présence du composant JTA.

Le type de transaction n'a normalement pas besoin d'être spécifié lors du déploiement sur Java SE. Il sera juste par défaut RESOURCE_LOCAL, mais peut être spécifié explicitement pour faire le contrat de programmation plus clair.

La source de données

Lorsque nous avons décrit la configuration du serveur, nous avons illustré comment la source de données jta L'élément désigne l'emplacement JNDI de la source de données qui sera utilisée pour obtenir Connexions. Nous avons également vu que certains serveurs peuvent même utiliser la source de données par défaut.

L'élément non-jta-data-source est utilisé dans le serveur pour spécifier où resource- les connexions locales peuvent être obtenues dans JNDI. Il peut également être utilisé par les fournisseurs qui lecture optimisée via des connexions non JTA.

Lors de la configuration pour l'extérieur du serveur, non seulement nous ne pouvons pas compter sur JTA, car nous décrit dans la section type de transaction, mais nous ne pouvons pas du tout compter sur JNDI. Nous ne peut pas s'appuyer de manière portative sur l'un ou l'autre des éléments de source de données dans les configurations Java SE.

Lors de l'utilisation de transactions locales aux ressources en dehors du serveur, le fournisseur obtient connexions à la base de données directement distribuées par le pilote JDBC. Afin qu'il puisse obtenir ces connexions, il doit obtenir les informations spécifiques au pilote, qui incluent généralement le nom de la classe du pilote, l'URL que le pilote utilise pour se connecter à la base de données, et l'authentification de l'utilisateur et du mot de passe que le pilote transmet également à la base de données. Ces métadonnées peuvent être spécifiées de la manière que le fournisseur préfère qu'elles soient spécifiées, mais tous les fournisseurs doivent prendre en charge les propriétés JDBC standard dans la section des propriétés.

Épisode 691

Référencement 14-15 montre un exemple d'utilisation des propriétés standard pour se connecter au Derby base de données via le pilote Derby.

Annonce 14-15. Spécification des propriétés JDBC au niveau des ressources

```
<persistence-unit name = "EmployeeService">
    ...
    <propriétés>
        <nom de propriété = "javax.persistence.jdbc.driver"
            value = "org.apache.derby.jdbc.ClientDriver" />
        <nom de la propriété = "javax.persistence.jdbc.url"
            value = "jdbc: derby: // localhost: 1527 / EmpServDB; create = true" />
        <nom de la propriété = "javax.persistence.jdbc.user"
            value = "APP" />
        <property name = "javax.persistence.jdbc.password"
            value = "APP" />
    </properties>
</persistence-unit>
```

Fournisseurs

De nombreux serveurs auront un fournisseur par défaut ou natif qu'ils utiliseront lorsque le fournisseur n'est pas spécifié. Il appellera automatiquement ce fournisseur pour créer un EntityManagerFactory au moment du déploiement.

Lorsqu'elle n'est pas sur un serveur, la fabrique est créée par programme à l'aide de la fonction Persistence classe. Lorsque la méthode createEntityManagerFactory () est appelée, le paramètre Persistence class commencera un protocole de pluggability intégré qui sortira et trouvera le fournisseur qui est spécifié dans la configuration de l'unité de persistance. Si aucun n'a été spécifié, le premier qui il trouve sera utilisé. Les fournisseurs s'exportent via un service qui existe dans le fournisseur JAR qui doit se trouver sur le chemin de classe. Le résultat net est que l'élément fournisseur est non requis.

Dans la majorité des cas, lorsqu'un seul fournisseur sera sur le chemin de classe, le fournisseur sera détecté et utilisé par la classe Persistence pour créer un EntityManagerFactory pour une unité de persistance donnée. Si jamais vous vous trouvez dans une situation où vous avez deux prestataires sur le chemin de classe et que vous souhaitez en utiliser un en particulier, vous devez spécifier le fournisseur class dans l'élément provider. Pour éviter les erreurs d'exécution et de déploiement, le fournisseur L'élément doit être utilisé si l'application a une dépendance de code sur un fournisseur spécifique.

Épisode 692

Liste des classes gérées

L'un des avantages du déploiement à l'intérieur du serveur est qu'il s'agit d'un environnement structuré. Pour cette raison, le serveur peut prendre en charge le processus de déploiement d'une manière qui ne peut pas être réalisée par un simple environnement d'exécution Java SE. Le serveur doit déjà traiter toutes les unités de déploiement dans une application et peut faire des choses comme détecter tous les classes de persistance gérées dans un JAR EJB ou une archive de persistance. Ce genre de cours La détection fait des archives de persistance un moyen très pratique de regrouper une unité de persistance.

Le problème avec ce type de détection en dehors du serveur est que Java SE
l'environnement permet à toutes sortes de ressources de classe différentes d'être ajoutées au chemin de classe,
y compris les URL réseau ou tout autre type de ressource acceptable pour un chargeur de classe.
Il n'y a pas de limites d'unité de déploiement officielles dont le fournisseur a connaissance. Cette
rend difficile pour JPA d'exiger des fournisseurs qu'ils prennent en charge la détection automatique de
les classes gérées dans une archive de persistance. La position officielle de l'API est que
pour qu'une application soit portable chez tous les fournisseurs, elle doit répertorier explicitement tous les
classes dans l'unité de persistance à l'aide d'éléments de classe. Lorsqu'une unité de persistance est grande
et comprend un grand nombre de classes, cette tâche peut devenir assez onéreuse.

Dans la pratique, cependant, une partie du temps, les classes sont assises dans une persistance régulière
archiver le fichier JAR sur le système de fichiers, et le moteur d'exécution du fournisseur peut détecter que le
le serveur ferait l'affaire dans Java EE s'il pouvait simplement déterminer le JAR dans lequel effectuer la recherche. Pour cette raison,
bon nombre des principaux fournisseurs prennent en charge la détection des classes en dehors du
serveur. C'est vraiment un problème d'utilisabilité essentiel depuis la maintenance d'une classe
la liste serait si lourde qu'elle constituerait un goulot d'étranglement pour la productivité à moins que vous n'ayez un outil
gérer la liste pour vous.

Un corollaire de la directive officielle de portabilité pour utiliser des éléments de classe pour énumérer
la liste des classes gérées est que l'élément exclude-unlisted-classes n'est pas
garantie d'avoir un impact sur les unités de persistance Java SE. Certains fournisseurs peuvent autoriser
cet élément à utiliser en dehors du serveur, mais ce n'est pas vraiment très utile en SE
l'environnement de toute façon, étant donné la flexibilité du chemin de classe et les allocations d'emballage dans
cet environnement.

Spécification des propriétés au moment de l'exécution

L'un des avantages de l'exécution en dehors du serveur est la possibilité de spécifier les propriétés du fournisseur
à l'exécution. Ceci est disponible en raison de la surcharge de createEntityManagerFactory ()
méthode qui accepte une mappe de propriétés en plus du nom de l'unité de persistance.

Épisode 693

Les propriétés passées à cette méthode sont combinées avec celles déjà spécifiées,
normalement dans le fichier persistence.xml. Ils peuvent être des propriétés supplémentaires ou ils peuvent
remplacer la valeur d'une propriété déjà spécifiée. Cela peut ne pas sembler très
utile pour certaines applications, car mettre les informations de configuration d'exécution dans le code est
normalement pas considéré comme meilleur que de l'isoler dans un fichier XML. Cependant, on peut
imaginez que ce soit un moyen pratique de définir les propriétés obtenues à partir d'une entrée de programme,
comme la ligne de commande, comme mécanisme de configuration encore plus dynamique.

Dans la liste 14-16 est un exemple de prise des propriétés d'utilisateur et de mot de passe
à partir de la ligne de commande et en les transmettant au fournisseur lors de la création du
EntityManagerFactory.

Annonce 14-16. Utilisation des propriétés de persistance de la ligne de commande

```
public class EmployeeService {
    public static void main (String [] args) {
        Accessoires de carte = nouveau HashMap ();
        props.put ("javax.persistence.jdbc.user", args [0]);
        props.put ("javax.persistence.jdbc.password", args [1]);
        EntityManagerFactory emf = Persistance
            .createEntityManagerFactory ("EmployeeService", accessoires);
        // ...
        emf.close ();
    }
}
```

Classpath système

À certains égards, la configuration d'une unité de persistance dans une application Java SE est en fait plus facile que la configuration sur le serveur car le chemin de classe est simplement le chemin de classe du système. L'ajout de classes ou de fichiers JAR sur le chemin de classe système est un exercice trivial. Dans le serveur, nous pouvons doivent manipuler le chemin de classe du manifeste ou ajouter une application spécifique au fournisseur configuration du chemin de classe.

Épisode 694

Génération de schéma

La génération de schéma faisait simplement référence au processus de prise des mappages dans le unité de persistance et inférer un schéma possible de tables de base de données pour prendre en charge ces mappages. Cependant, il comprend désormais plus que la simple génération de tableaux. Schéma les propriétés de génération et les scripts peuvent désormais être utilisés pour créer et / ou supprimer les tables, et même provoquer le préchargement des données dans celles-ci avant d'exécuter une application.

Remarque, sauf indication contraire, le terme «génération de schéma» fait référence à la génération de tables pour un schéma de base de données préexistant, n'émettant pas nécessairement un Commande de base de données CREATE SCHEMA.

l'une des plaintes concernant la génération de schéma est que vous ne pouvez pas spécifier tout ce dont vous avez besoin pour pouvoir régler finement les schémas de table. c'était pas accidentel. il y a trop de différences entre les bases de données et trop différents paramètres pour essayer de mettre des options pour chaque type de base de données. si chaque base de données l'option de réglage a été exposée via Jpa, nous finirions par dupliquer les fonctionnalités du langage de définition de données (ddl) dans une API qui n'était pas censée être une base de données fonction de génération de schéma. comme nous l'avons mentionné précédemment, la majorité des applications se retrouvent dans un scénario de cartographie de rencontre du milieu dans tous les cas, et quand ils contrôlent le schéma, le schéma final sera généralement réglé par un administrateur de base de données ou par une personne possédant le niveau de base de données approprié expérience.

Astuce bien que de nombreux éléments d'annotation de génération de schéma aient été présente depuis Jpa 1.0, la spécification n'exigeait pas que les fournisseurs prennent en charge génération de tables jusqu'à Jpa 2.1. c'est aussi dans Jpa 2.1 que le schéma les propriétés de génération et les méthodes API ont été introduites. rien n'a été changé concernant la génération de schéma dans Jpa 2.2.

Épisode 695

Le processus de génération

Il existe un certain nombre d'aspects différents de la génération de schéma qui peuvent être spécifiés indépendamment ou en combinaison les uns avec les autres. Avant de nous y plonger, nous passons en revue certains des concepts de génération de schémas et le processus de base pour que vous puissiez obtenir un ressentez ce qui se passe. Ensuite, dans les sections suivantes, nous décrivons les propriétés et annotations qui peuvent être utilisées pour produire un résultat souhaité.

La génération de schéma est effectuée par une partie du fournisseur de persistance que nous appellerons le processeur de génération. Ce processeur est chargé de prendre certaines entrées et générer une ou plusieurs sorties. Les entrées peuvent être soit le domaine d'application les objets accompagnés de métadonnées de mappage (sous forme d'annotation ou XML), ou des scripts DDL préexistants accessibles au processeur (emballés dans le application ou référençable). Les sorties seront du DDL qui est soit exécuté dans la base de données par le processeur ou écrit dans des fichiers de script. Figure 14-1 montre un vue simple du processeur.

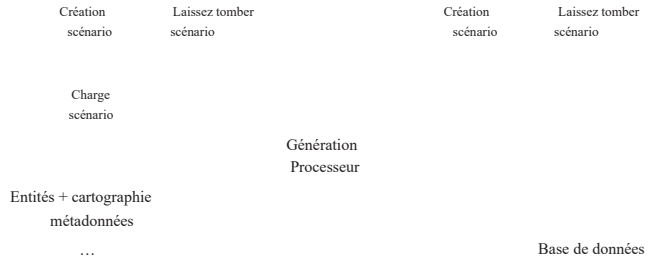

Figure 14-1. *Le processeur de génération*

Le processus de génération de schéma se produit généralement soit lorsque l'application est déployé (par exemple, dans un conteneur), ou lorsque la fabrique de gestionnaires d'entités obtient créé (lors d'un appel Persistence.createEntityManagerFactory () dans Java SE).

Épisode 696

Il existe trois opérations distinctes prises en charge par la génération de schéma. Vous pouvez créer les objets de schéma, déposez le schéma (supprimez les objets de schéma) ou préchargez les données dans un schéma. Bien que les opérations de création et de suppression de schéma soient spécifiées dans le même propriété, vous pouvez également mélanger et assortir les opérations que vous souhaitez exécuter. L'ordre dans lequel ils seront exécutés sera cependant dicté par ce qui est sensible. Pour Par exemple, si les trois sont spécifiés, l'ancien schéma sera d'abord supprimé, le nouveau sera être créé, puis les données seront préchargées.

Pour contrôler les entrées et les sorties et ce qui se passe lors de la génération de schéma le traitement d'un certain nombre de propriétés standard peut être spécifié de manière statique dans le persistence.xml ou dynamiquement dans un appel d'exécution dans Java SE à Persistence. createEntityManagerFactory () ou Persistence.generateSchema (). Propriétés passées in lors de l'exécution remplacera les propriétés définies dans le fichier persistence.xml.

Propriétés de déploiement

Les propriétés standard qui peuvent être spécifiées dans les éléments de propriété dans la persistance. Les descripteurs xml sont répertoriés dans les sections suivantes. Les vendeurs peuvent offrir des propriétés et options qui les chevauchent ou les subsument, mais les options répertoriées ici doivent être pris en charge par tous les fournisseurs conformes et doit être utilisé par les applications souhaitant rester portable. Les propriétés appartiennent toutes à l'une des deux catégories suivantes: sortie de génération et entrée de génération.

Conseil Certaines des propriétés décrites dans cette section spécifient qu'une URL soit fourni en tant que valeur pour une source de script ou un emplacement cible de script. Tandis que le la spécification ne prescrit aucun autre format, dans de nombreux cas prend en charge l'utilisation d'un chemin de fichier simple.

Sortie de génération

La présence même de l'une des deux premières propriétés de cette catégorie détermine qu'un l'opération de génération de schéma se produit, soit dans la base de données, soit dans des scripts. La propriété value dicte les opérations de schéma qui doivent se produire à l'emplacement ciblé. le les propriétés ne sont pas exclusives, ce qui signifie que plusieurs propriétés peuvent être incluses, provoquant la génération du processus de génération de schéma vers plusieurs cibles.

Épisode 697

Les deux secondes propriétés (create-target et drop-target) sont une sortie de script spécificateur et sont utilisés en combinaison avec la propriété d'action de script (javax. persistence.schema-generation.scripts.action).

javax.persistence.schema-generation.database.action

Cette propriété provoquera les actions de génération de schéma spécifiées dans la valeur de propriété se produire dans la base de données. C'est la propriété la plus courante et la plus utile, et plus les trois quarts des applications seront probablement en mesure de se contenter de spécifier cela une propriété, avec une valeur de drop-and-create, pour la génération de schéma. Le possible les valeurs sont

- aucun (par défaut): ne pas générer de schéma dans la base de données.
- create: Génère le schéma dans la base de données.
- drop: supprime le schéma de la base de données.
- drop-and-create: Supprimez le schéma de la base de données, puis générer un nouveau schéma dans la base de données.

Exemple:

```
<property name = "javax.persistence.schema-generation.database.action"
    value = "drop-and-create" />
```

javax.persistence.schema-generation.scripts.action

Cette propriété est utilisée pour que la génération de schéma génère des scripts de sortie. Cela pourrait être utilisé à la place ou en plus de la génération javax.persistence.schema. Propriété database.action. La valeur de la propriété détermine si une création le script, ou le script de suppression sera généré, ou les deux. Les valeurs possibles sont

- aucun (par défaut): ne génère aucun script.
- create: générez un script pour créer le schéma.
- drop: générez un script pour supprimer le schéma.
- drop-and-create: générez un script pour supprimer le schéma et un script pour créer le schéma.

Épisode 698

Notez qu'il doit y avoir des cibles de script correspondantes pour chaque valeur. Si le créer est spécifiée, puis javax.persistence.schema-generation.scripts.create-
La propriété cible doit également être fournie et avoir une valeur. Si l'option de suppression est spécifiée, alors la propriété javax.persistence.schema-generation.scripts.drop-target doit être fournis et avoir une valeur. Si drop-and-create est spécifié, les deux propriétés doivent être fourni avec des valeurs.

Exemple:

```
<nom de propriété = "javax.persistence.schema-generation.scripts.action"
    valeur = "créer" />
<property name = "javax.persistence.schema-generation.scripts.create-target
    value = "fichier: /// c: /scripts/create.ddl" />
```

javax.persistence.schema-generation.scripts.create-target

Cette propriété permet de spécifier l'emplacement auquel générer le script de création et est utilisé en conjonction avec javax.persistence.schema-generation.scripts.
action. La valeur est une URL de fichier et doit spécifier un chemin absolu plutôt qu'un relatif un. L'utilisation de cette propriété dans un conteneur peut être quelque peu limitée, selon le degré d'accès au système de fichiers autorisé par le conteneur.

Exemple:

```
<nom de propriété = "javax.persistence.schema-generation.scripts.action"
    valeur = "créer" />
<property name = "javax.persistence.schema-generation.scripts.create-target"
    value = "fichier: /// c: /scripts/create.ddl" />
```

javax.persistence.schema-generation.scripts.drop-target

Cette propriété est utilisée pour spécifier l'emplacement auquel générer le script de suppression et est utilisé en conjonction avec javax.persistence.schema-generation.scripts.
action. Semblable à la propriété script create-target, la valeur de cette propriété est un l'URL du fichier et doit spécifier un chemin absolu plutôt qu'un chemin relatif. Comme le script create-target, l'utilisation de cette propriété dans un conteneur peut être quelque peu limitée.

Exemple:

```
<nom de propriété = "javax.persistence.schema-generation.scripts.action"
    value = "drop" />
<property name = "javax.persistence.schema-generation.scripts.drop-target"
    value = "fichier: /// c: /scripts/drop.ddl" />
```


Épisode 699

Entrée de génération

Les deux premières propriétés de cette catégorie spécifient si la création de schéma et
la suppression doit être générée en fonction des métadonnées de mappage, des scripts ou des deux. Ils
ne sont vraiment utiles que pour le cas spécial «les deux» du mélange de métadonnées avec des scripts, bien que [1].

Si les sources sont combinées, il y a l'option supplémentaire de commander qui
se produit en premier, la création du schéma à partir des métadonnées de mappage ou des scripts. L'option
utiliser les deux sources n'est probablement pas une exigence courante, mais peut être utile
personnalisations spécifiques à un schéma principalement généré à partir des métadonnées.

La deuxième paire de propriétés, create-script-source et drop-script-source,
peut être utilisé pour spécifier les scripts exacts à utiliser comme entrées, et le dernier, sql-load-
script-source, peut être utilisé pour précharger des données dans le schéma. La valeur de chacun de ces
trois propriétés peuvent être un chemin de fichier relatif à la racine de l'unité de persistance ou une URL de fichier
qui est accessible par le fournisseur de persistance dans n'importe quel environnement dans lequel il s'exécute.

Si une propriété de sortie est spécifiée, mais aucune propriété d'entrée, les métadonnées de mappage
être utilisé comme entrée.

javax.persistence.schema-generation.create-source

Cette propriété détermine quelle entrée doit être prise en compte lors de la génération du DDL vers
créez le schéma. Les valeurs possibles sont

- métadonnées: générez un schéma à partir des métadonnées de mappage.
- script: générez un schéma à partir d'un script existant.
- metadata-then-script: générer un schéma à partir de métadonnées de mappage
  puis à partir d'un script existant.
- script-then-metadata: générer un schéma à partir d'un script existant
  puis du mappage des métadonnées.

Exemple:

```
<property name = "javax.persistence.schema-generation.create-source"
    value = "metadata-then-script" />
```

[1] Les métadonnées sont l'entrée par défaut, donc si les métadonnées sont l'entrée souhaitée, elles n'ont pas besoin d'être spécifié. Si les scripts sont les entrées souhaitées, alors la commande create-script-source ou drop-script- les propriétés source peuvent être spécifiées sans qu'il soit nécessaire de spécifier la source de création ou drop-source.

Épisode 700

javax.persistence.schema-generation.drop-source

Cette propriété détermine quelle entrée doit être prise en compte lors de la génération du DDL vers
supprimez le schéma. Les valeurs possibles sont

- métadonnées: générez DDL pour supprimer le schéma des métadonnées de mappage.
- script: utilisez un script existant pour que DDL supprime le script de schéma.
- metadata-then-script: générer du DDL à partir de métadonnées de mappage,
  puis utilisez un script existant.
- script-then-metadata: utilisez un script existant, puis générez DDL
  à partir de la cartographie des métadonnées.

Exemple:

```
<property name = "javax.persistence.schema-generation.drop-source"
    value = "metadata-then-script" />
```

javax.persistence.schema-generation.create-script-source

Cette propriété spécifie un script à utiliser lors de la création d'un schéma.
Exemple:

```
<property name = "javax.persistence.schema-generation.scripts.create-script-
la source"
value = "META-INF / createSchema.ddl" />
```

javax.persistence.schema-generation. drop-script-source

Cette propriété spécifie un script à utiliser lors de la suppression d'un schéma.
Exemple:

```
<property name = "javax.persistence.schema-generation.scripts.drop-script- source"
value = "META-INF / dropSchema.ddl" />
```


Épisode 701

javax.persistence. sql-load-script-source

Cette propriété spécifie un script à utiliser lors du préchargement d'un schéma.
Exemple:

```
<property name = "javax.persistence.sql-load-script-source"
value = "META-INF / loadData.ddl" />
```

Propriétés d'exécution

Les propriétés de déploiement décrites dans la section précédente peuvent également être transmises à l'adresse runtime, mais dans certains cas, les valeurs de propriété doivent être des objets au lieu des chaînes qui ont été abordés dans la section «Propriétés de déploiement». Table 14-3 décrit le différences lors de l'utilisation des propriétés d'exécution au lieu des propriétés de déploiement.

Tableau 14-3. *Différences de propriété de génération de schéma d'exécution*

Nom de la propriété	Différence
javax.persistence.schema-generation.create-script-source javax.persistence.schema-generation.drop-script-source javax.persistence.sql-load-script-source	java.io.Reader au lieu du fichier String chemin
javax.persistence.schema-generation.scripts.create-target javax.persistence.schema-generation.scripts.drop-target	java.io.Writer au lieu du fichier String chemin

Annotations de mappage utilisées par la génération de schéma

Lorsque nous avons mentionné la génération de schéma dans le chapitre 4, nous avons promis de passer en revue mappage des éléments d'annotation pris en compte lors de la génération de schéma. Dans

cette section, nous respectons cet engagement et expliquons quels éléments sont appliqués à la schéma généré.

Épisode 702

Cependant, quelques commentaires s'imposent avant de commencer. Premièrement le les éléments qui contiennent les propriétés dépendant du schéma sont, à quelques exceptions près, dans le annotations physiques. C'est pour essayer de les séparer du non-schéma logique- métadonnées associées. Deuxièmement, ces annotations sont ignorées, pour la plupart[2] , si le schéma n'est pas généré. C'est l'une des raisons pour lesquelles leur utilisation est un peu déplacée dans la cas, car les informations de schéma sur la base de données sont de peu d'utilité une fois que le schéma a été créé et est utilisé.

Contraintes uniques

Une contrainte unique peut être créée sur une colonne générée ou une colonne de jointure à l'aide de la élément unique dans @Column, @JoinColumn, @MapKeyColumn ou @MapKeyJoinColumn annotations. Il n'y a en fait pas beaucoup de cas où cela sera nécessaire car la plupart des fournisseurs génèrent une contrainte unique lorsque cela est approprié, comme lors de la jointure colonne de relations un-à-un. Sinon, la valeur de l'élément unique par défaut est faux. Référencement14-17 montre une entité avec une contrainte unique définie pour la colonne STR.

Annonce 14-17. Y compris des contraintes uniques

```
@Entité
Employé de classe publique {
    @Id id int privé;
    @Column (unique = vrai)
    nom de chaîne privé;
    // ...
}
```

Notez que l'élément unique n'est pas nécessaire dans la colonne d'identifiant car un La contrainte de clé primaire sera toujours générée pour la clé primaire.

Une deuxième façon d'ajouter une contrainte unique consiste à incorporer une ou plusieurs Annotations @UniqueConstraint dans un élément uniqueConstraints de la @Table, Annotations @SecondaryTable, @JoinTable, @CollectionTable ou @TableGenerator. N'importe quel nombre de contraintes uniques peut être ajouté à la définition de table, y compris

[2] L'exception à cette règle peut être l'élément facultatif des annotations de mappage, qui aboutit à une contrainte NON NULL, mais qui peut également être utilisée en mémoire pour indiquer que la valeur est ou n'est pas autorisé à être défini sur null.

Épisode 703

contraintes composées. La valeur transmise à l'annotation @UniqueConstraint est un tableau d'une ou plusieurs chaînes répertoriant les noms de colonnes constituant la contrainte. Référencement 14-18 montre comment définir une contrainte unique dans le cadre d'une table.

Annonce 14-18. Contraintes uniques spécifiées dans la définition de table

```
@Entité
@Table (nom = "EMP",
        uniqueConstraints = @ UniqueConstraint (columnNames = {"NAME"}))
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    // ...
}
```

Contraintes nulles

Les contraintes sur une colonne peuvent également être sous la forme de contraintes nulles. Une contrainte nulle indique simplement que la colonne peut être nulle ou non. Il est défini lorsque la colonne est déclaré dans le cadre du tableau.

Les contraintes nulles sont définies sur une colonne à l'aide de l'élément nullable dans la @Column, @JoinColumn, @MapKeyColumn, @MapKeyJoinColumn ou @OrderColumn annotations. Une colonne autorise les valeurs nulles par défaut, donc cet élément doit vraiment être utilisé uniquement lorsqu'une valeur pour le champ ou la propriété est requise. Référencement14-19 montre comment définir l'élément Nullable des mappages de base et de relation.

Annonce 14-19. Contraintes nulles spécifiées dans les définitions de colonne

```
@Entité
Employé de classe publique {
    @Id id int privé;
    @Column (nullable = false)
    nom de chaîne privé;
    @ManyToOne
    @JoinColumn (nullable = false)
    adresse d'adresse privée;
    // ...
}
```


Épisode 704

Index

Lorsqu'une clé primaire est créée lors de la génération de séquence, elle sera automatiquement indexé. Cependant, des index supplémentaires peuvent être générés sur une colonne ou sur une séquence de colonnes dans une table en utilisant l'élément indexes et en spécifiant un ou plusieurs @Index annotations, dont chacune spécifie un index à ajouter à la table. L'élément indexes peut être spécifié sur @Table, @SecondaryTable, @JoinTable, @CollectionTable et Annotations @TableGenerator. Référencement14-20 montre un exemple d'ajout d'un index sur la colonne EMPNAME lors de la génération du schéma de la table EMP. Nous avons nommé l'index, mais le fournisseur nous le nommerait si nous ne spécifions pas de nom. Notez que plusieurs Les noms de colonne séparés par des virgules peuvent être spécifiés dans la chaîne columnNames dans le cas d'un index multi-colonnes. Un ASC ou DESC facultatif peut également être ajouté à la colonne en utilisant la même syntaxe que dans l'annotation @OrderBy (voir Chapitre 5). Par défaut, ils seront supposé être dans l'ordre croissant. On peut même mettre une contrainte d'unicité supplémentaire sur l'index en utilisant l'élément unique.

Annonce 14-20. Ajouter un index

```
@Entité
@Table (nom = "EMP",
```

```
                indexes = {@ Index (name = "NAME_IDX", columnNames = "EMPNAME",
                unique = vrai)})
Employé de classe publique {
    @Id id int privé;
    @Column (name = "EMPNAME")
    nom de chaîne privé;
    // ...
}
```

Contraintes de clé étrangère

Lorsqu'une entité comprend une ou plusieurs colonnes de jointure, lors de la génération du schéma, les fournisseurs de persistance peuvent choisir de générer des contraintes de clé étrangère basées sur ces relations d'entité. Il n'est ni prescrit ni conseillé qu'ils génèrent les générer ou non, de sorte que différents fournisseurs peuvent faire des choses différentes. Vous pouvez contrôler si une contrainte de clé étrangère est générée pour une colonne de jointure en utilisant le

Épisode 705

Elément ForeignKey dans @JoinColumn, @PrimaryKeyJoinColumn ou Annotations @MapKeyJoinColumn. L'annotation @ForeignKey est spécifiée dans le Elément ForeignKey pour définir explicitement si la contrainte doit être générée, éventuellement donner un nom à la contrainte, et éventuellement même spécifier la définition exacte de la contrainte à être utilisé. Référencement14-21 montre une colonne de jointure avec une contrainte de clé étrangère qui inclut à la fois le mode de contrainte de CONSTRAINT et une définition de contrainte explicite. Notez que la contrainte est autoréférentielle.

Annonce 14-21. Définition de contrainte de clé étrangère sur une colonne de jointure

```
@Entity @Table (nom = "EMP")
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    @ManyToOne
    @JoinColumn (nom = "MGR",
        ForeignKey = @ ForeignKey (
            valeur = ConstraintMode.CONSTRAINT,
            ForeignKeyDefinition = "RÉFÉRENCES DE CLÉ ÉTRANGÈRE (Mgr) Emp (Id)"))
    gestionnaire d'employé privé;
    // ...
}
```

L'élément ForeignKey peut également être utilisé dans une annotation qui inclut un groupe de colonnes de jointure. Par exemple, l'élément value de @JoinColumns est un tableau de @JoinColumn, mais si l'élément ForeignKey est spécifié dans @JoinColumns, il s'appliquent à toutes les colonnes de jointure ensemble. Il ne serait pas spécifié d'inclure à la fois un Elément ForeignKey dans @JoinColumns ainsi qu'un élément ForeignKey dans un ou plusieurs des annotations @JoinColumn intégrées. Un comportement similaire s'appliquerait pour la valeur éléments des annotations @PrimaryKeyJoinColumns et @MapKeyJoinColumns, pour les JoinColumns éléments de @AssociationOverride, @CollectionTable et @JoinTable, et l'élément pkJoinColumns de @SecondaryTable.

Référencement 14-22 montre comment faire en sorte que le fournisseur ne génère pas de clé étrangère contraintes pour les colonnes de jointure de clé primaire de la table secondaire (qui dans ce cas utilisent les noms par défaut).

Épisode 706

Annonce 14-22. Désactivation des contraintes de clé étrangère ajoutées sur une table secondaire

```
@Entity @Table (nom = "EMP")
@SecondaryTable (nom = "EMP_REC",
                                    fo reignKey = @ ForeignKey (ConstraintMode.
                                        NO_CONSTRAINT))
Employé de classe publique {
    @Id id int privé;
    nom de chaîne privé;
    // ...
}
```

Notez que l'annotation @JoinTable a un élément inverseForeignKey supplémentaire qui applique une annotation @ForeignKey aux colonnes de jointure dans ses inversesJoinColumns élément.

Colonnes basées sur des chaînes

Lorsqu'aucune longueur n'est spécifiée pour une colonne générée pour stocker des valeurs de chaîne, la longueur sera par défaut de 255. Lorsqu'une colonne est générée pour un mappage de base d'un champ ou d'une propriété de type String, char [] ou Character [], sa longueur doit être explicitement répertorié dans l'élément length de l'annotation @Column si 255 n'est pas la valeur souhaitée longueur maximale. Référencement14-23 montre une entité avec des longueurs explicitement spécifiées pour les cordes.

Annonce 14-23. Spécification de la longueur des types de colonnes basés sur des caractères

```
@Entité
Employé de classe publique {
    @Id
    @Column (longueur = 40)
    nom de chaîne privé;
    @ManyToOne
    @JoinColumn (nom = "MGR")
    gestionnaire d'employé privé;
    // ...
}
```


Épisode 707

Vous pouvez voir dans l'exemple précédent qu'il n'y a pas d'élément de longueur similaire dans l'annotation @JoinColumn. Lorsque les clés primaires sont basées sur une chaîne, le fournisseur peut définir la longueur de la colonne de jointure à la même longueur que la colonne de clé primaire dans la table qui est être joint à. Cependant, il n'est pas nécessaire que cela soit pris en charge.

Il n'est pas défini pour la longueur à utiliser pour les grands objets; certaines bases de données ne le font pas exigent ou même permettent de spécifier la longueur des lobs (grands objets).

Colonnes à virgule flottante

Les colonnes contenant des types à virgule flottante ont une précision et une échelle associées à leur. La précision est juste le nombre de chiffres utilisés pour représenter la valeur, et l'échelle est le nombre de chiffres après la virgule décimale. Ces deux valeurs peuvent être spécifié comme éléments de précision et d'échelle dans l'annotation @Column lors du mappage d'un type à virgule flottante. Comme les autres éléments de génération de schéma, ils n'ont aucun effet sur le entité au moment de l'exécution. Le listing 14-24 montre comment définir ces valeurs.

Annonce 14-24. Spécification de la précision et de l'échelle de la colonne à virgule flottante

Les types

```
@Entité
classe publique PartTimeEmployee {
    // ...
    @Column (précision = 8, échelle = 2)
    flotteur privé hourlyRate;
    // ...
}
```

La précision de pointe peut être définie différemment pour différentes bases de données. dans certaines bases de données et pour certains types à virgule flottante, il s'agit du nombre de chiffres binaires, tandis que pour d'autres, c'est le nombre de chiffres décimaux.

Épisode 708

Définition de la colonne

Il peut y avoir un moment où vous êtes satisfait de toutes les colonnes générées sauf une. Le type de colonne n'est pas ce que vous souhaitez et vous ne voulez pas passer par le problème de générer manuellement le schéma pour une seule colonne. C'est un exemple lorsque l'élément columnDefinition est utile. En roulant le DDL à la main pour la colonne, nous pouvons l'inclure comme définition de colonne et laisser le fournisseur l'utiliser pour définir la colonne.

L'élément columnDefinition est disponible dans toutes les annotations orientées colonnes types, y compris @Column, @JoinColumn, @PrimaryKeyJoinColumn, @MapKeyColumn, @MapKeyJoinColumn, @OrderColumn et @DiscriminatorColumn. Chaque fois qu'une colonne doit être généré, l'élément columnDefinition peut être utilisé pour indiquer le DDL chaîne qui doit être utilisée pour générer le type (sans compter la virgule de fin). Cette donne à l'utilisateur un contrôle complet sur ce qui est généré dans le tableau pour la colonne en cours mappé. Il permet également d'utiliser un type ou un format spécifique à la base de données qui peut remplacer le type généré proposé par le fournisseur pour la base de données utilisée[1] . Annonce 14-25 affiche quelques définitions spécifiées pour deux colonnes et une colonne de jointure.

Annonce 14-25. Utilisation d'une définition de colonne pour contrôler la génération DDL

```
@Entité
Employé de classe publique {
    @Id
    @Column (columnDefinition = "NVARCHAR2 (40)")
```

```
    nom de chaîne privé;
    @Column (nom = "START_DATE",
                columnDefinition = "DATE DEFAULT SYSDATE")
    private java.sql.Date startDate;
    @ManyToOne
    @JoinColumn (nom = "MGR", columnDefinition = "NVARCHAR2 (40)")
    gestionnaire d'employé privé;
    // ...
}
```

3 La colonne résultante doit être prise en charge par le moteur d'exécution du fournisseur pour permettre la lecture depuis et écrire dans la colonne.

Épisode 709

Dans cet exemple, nous utilisons un champ de caractère Unicode pour la clé primaire, puis également pour la colonne de jointure qui fait référence à la clé primaire. Nous définissons également la date à laquelle a attribué la date actuelle par défaut au moment où l'enregistrement a été inséré (au cas où il n'aurait pas été spécifié).

La spécification de la définition de colonne est une pratique de génération de schéma assez puissante qui permet de remplacer la colonne générée par une personnalisation définie par l'application définition de colonne. Mais la puissance est également accompagnée de certains risques. Quand une colonne la définition est incluse, les autres métadonnées de génération associées à la colonne sont ignoré. Spécification de la précision, de l'échelle ou de la longueur dans la même annotation qu'une colonne une définition serait à la fois inutile et déroutante.

Non seulement l'utilisation de columnDefinition dans votre code vous lie à un schéma particulier, mais il vous lie également à une base de données particulière car le DDL a tendance à être une base de données spécifique. C'est juste un compromis flexibilité / portabilité, et vous devez décider si c'est approprié à votre application.

Résumé

C'est un exercice simple pour empaqueter et déployer des applications de persistance à l'aide de Java API de persistance. Dans la plupart des cas, il suffit d'ajouter un fichier persistence.xml très court fichier dans le JAR contenant les classes d'entité.

Dans ce chapitre, nous avons décrit comment configurer l'unité de persistance dans Java EE Environnement serveur utilisant le fichier persistence.xml et comment, dans certains cas, le nom peut être le seul paramètre requis. Nous avons ensuite expliqué quand postuler et comment spécifier le type de transaction, le fournisseur de persistance et la source de données. Nous avons montré comment utiliser et spécifier le fichier de mappage orm.xml par défaut, puis utiliser des mapper des fichiers dans la même unité de persistance. Nous avons également discuté des différentes façons dont les classes peuvent être incluses dans l'unité de persistance et comment personnaliser la persistance unité utilisant des propriétés standard et spécifiques au fournisseur.

Nous avons examiné les façons dont les unités de persistance peuvent être empaquetées et déployées sur un Java Application EE dans le cadre d'une archive EJB, d'une archive Web ou d'une archive de persistance accessible à tous les composants de l'application. Nous avons examiné comment les unités de persistance peut exister dans différentes portées d'une application Java EE déployée et quel est le nom les règles de cadrage étaient. Nous avons ensuite comparé les pratiques de configuration et de déploiement de déployer une application dans un environnement Java SE.

Enfin, nous avons montré comment un schéma peut être généré dans la base de données pour correspondre au exigences de l'unité de persistance à l'aide de scripts ou du modèle de domaine et mappage des métadonnées. Nous avons mis en garde contre l'utilisation du schéma généré pour la production systèmes, mais a montré comment il peut être utilisé pendant le développement, le prototypage et les tests pour soyez opérationnel rapidement et facilement.

Dans le chapitre suivant, nous examinons les pratiques acceptées et les meilleures pour les tests applications qui utilisent la persistance.

CHAPITRE 15

Essai

L'un des principaux arguments de vente de JPA a été la recherche d'une meilleure testabilité. le utilisation de classes Java simples et possibilité d'utiliser la persistance en dehors de l'application

serveur a rendu les applications d'entreprise beaucoup plus faciles à tester. Ce chapitre couvre l'unité tests et tests d'intégration avec des entités, avec un mélange de tests modernes et traditionnels techniques.

Test des applications d'entreprise

Les tests sont généralement considérés comme une bonne chose, mais comment devons-nous procéder exactement de le faire? Presque toutes les applications d'entreprise sont hébergées sur une sorte de serveur environnement, qu'il s'agisse d'un conteneur de servlet comme Apache Tomcat ou d'un Java EE complet serveur d'application. Une fois déployé dans un tel environnement, le développeur est plus isolé de l'application que s'il développait dans un runtime Java SE environnement. À ce stade, il ne peut être testé qu'à l'aide de l'interface publique du une application, comme un navigateur utilisant HTTP, un service Web, RMI ou une interface de messagerie.

Cela pose un problème pour les développeurs car pour faire des tests unitaires, nous voulons pouvoir se concentrer sur les composants d'une application de manière isolée. Une séquence élaborée de les opérations via un site Web peuvent être nécessaires pour accéder à une seule méthode d'un bean qui met en œuvre un service commercial particulier. Par exemple, pour afficher un enregistrement d'employé, un le client de test devra peut-être se connecter en utilisant un nom d'utilisateur et un mot de passe, parcourir plusieurs menus options, exécutez une recherche, puis accédez enfin à l'enregistrement. Ensuite, le HTML la sortie du rapport doit être vérifiée pour s'assurer que l'opération s'est déroulée comme prévu. Dans certaines applications, cette procédure peut être court-circuitée en accédant directement au URL qui récupère un enregistrement particulier. Mais avec de plus en plus d'informations en cache État de la session HTTP, les URL commencent à ressembler à des séquences aléatoires de lettres et Nombres. Obtenir un accès direct à une fonctionnalité particulière d'une application peut ne pas être facile atteindre.

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4_15

Épisode 712

Clients Java SE (appelés clients «gros») qui communiquent avec des bases de données et d'autres les ressources souffrent du même problème malgré leur capacité à exécuter le programme sans avoir besoin d'un serveur d'applications. L'interface utilisateur d'un client Java SE peut bien être une application Swing nécessitant des outils spéciaux pour la conduire afin de faire tout type de automatisation des tests. L'application n'est encore qu'une boîte noire sans moyen évident d'obtenir à l'intérieur.

De nombreuses tentatives ont été faites pour exposer les composants internes d'une application à des tests lors du déploiement sur un serveur. L'un des premiers était le Cactus[1] cadre, qui permet développeurs d'écrire des tests à l'aide de JUnit, qui sont ensuite déployés sur le serveur avec l'application et exécutée via une interface Web fournie par Cactus. Autres cadres a adopté une approche similaire en utilisant RMI au lieu d'une interface web pour contrôler les tests à distance. Actuellement, un framework appelé Arquillian[2] a commencé à gagner en popularité et utilise une approche connexe. Nous discutons brièvement d'Arquillian à la fin du chapitre.

Bien qu'efficaces, l'inconvénient de ces approches est que le serveur d'applications doit encore être opérationnel avant que nous puissions tenter tout type de test. Pour les développeurs qui utilisent le développement piloté par les tests (TDD), dans lequel les tests sont écrits avant le code et le la suite de tests unitaires complète est exécutée après chaque itération de développement (qui peut être aussi petite que un changement vers une seule méthode), tout type d'interaction avec le serveur d'application est un peu d'un problème. Même pour les développeurs qui pratiquent une méthodologie de test plus traditionnelle, l'exécution fréquente des tests est entravée par la nécessité de maintenir le serveur d'applications en marche, avec une étape de packaging et de déploiement avant chaque test.

Clairement, pour les développeurs qui souhaitent décomposer une application Java EE en son composant pièces et tester ces composants de manière isolée, il existe un besoin d'outils qui leur permettront exécuter directement des parties de l'application en dehors de l'environnement serveur dans lequel elle est normalement hébergé.

Terminologie

Tout le monde n'est pas d'accord sur ce qui constitue exactement un test unitaire ou un test d'intégration. Dans
En fait, il est fort probable que toute enquête sur un groupe de développeurs produira une grande variété de
résultats, certains de nature similaire tandis que d'autres s'aventurent dans des domaines complètement différents
essai. Par conséquent, nous pensons qu'il est important de définir notre terminologie pour les tests afin que vous
peut le traduire dans les termes avec lesquels vous êtes à l'aise.

[1] Visitez http://jakarta.apache.org/cactus/ pour plus d'informations.
[2] http://arquillian.org

Épisode 713

Nous voyons les tests tomber dans les quatre catégories suivantes:

- *Tests unitaires* : les *tests* unitaires sont écrits par les développeurs et se concentrent sur des
  composants d'une application. Selon votre approche, ce
  peut être une seule classe ou une collection de classes. La seule clé définissant
  à notre avis, le test unitaire n'est couplé à aucun
  ressources du serveur (celles-ci sont généralement supprimées dans le cadre du test
  processus) et s'exécute très rapidement. Il doit être possible d'exécuter
  une suite complète de tests unitaires à partir d'un IDE et obtenez les résultats
  dans quelques secondes. L'exécution des tests unitaires peut être automatisée et est
  souvent configuré pour se produire automatiquement dans le cadre de chaque fusion vers un
  système de gestion de la configuration.

- *Tests d'intégration* : les *tests d'* intégration sont également rédigés par les développeurs
  et se concentrer sur les cas d'utilisation au sein d'une application. Ils sont encore typiquement
  découplé du serveur d'application, mais la différence entre un
  test unitaire et un test d'intégration est que le test d'intégration rend complet
  l'utilisation de ressources externes telles qu'une base de données. En effet, une intégration
  test prend un composant d'une application et s'exécute de manière isolée comme
  s'il était toujours à l'intérieur du serveur d'applications. Exécution du test localement
  le rend beaucoup plus rapide qu'un test hébergé sur un serveur d'application,
  mais toujours plus lent qu'un test unitaire. Les tests d'intégration sont également automatisés
  et souvent exécutés au moins quotidiennement pour s'assurer qu'il n'y a pas de régressions
  introduit par les développeurs.

- *Tests fonctionnels* : les tests fonctionnels sont les tests de la boîte noire écrits
  et automatisé par des ingénieurs qualité plutôt que par des développeurs. Qualité
  les ingénieurs examinent les spécifications fonctionnelles d'un produit et ses
  interface utilisateur, et cherchez à automatiser les tests qui peuvent vérifier le produit
  comportement sans comprendre (ou se soucier) comment il est mis en œuvre.
  Les tests fonctionnels sont une partie essentielle du développement de l'application
  processus, mais il n'est pas réaliste d'exécuter ces tests dans le cadre de la journée.
  travail quotidien effectué par un développeur. Exécution automatisée de ces tests
  se déroule souvent selon un horaire différent, indépendant du programme régulier
  processus de développement.

Épisode 714

- *Tests d'acceptation* : les *tests d'* acceptation sont pilotés par le client. Ces tests, généralement effectuées manuellement, sont effectuées directement par les clients ou des représentants qui jouent le rôle du client. L'objectif de un test d'acceptation consiste à vérifier que les exigences énoncées par le client sont satisfaits dans l'interface utilisateur et le comportement du application.

Dans ce chapitre, nous nous concentrons uniquement sur les tests unitaires et les tests d'intégration. Ces tests sont écrites par les développeurs au profit des développeurs et constituent ce qu'on appelle le blanc test de boîte. Ces tests sont rédigés avec une compréhension complète de la façon dont l'application est mis en œuvre et ce qu'il faudra non seulement pour tester le chemin réussi à travers un application mais aussi pour déclencher des scénarios de panne.

Test en dehors du serveur

L'élément commun entre les tests unitaires et les tests d'intégration est qu'ils sont exécutés sans avoir besoin d'un serveur d'applications. Malheureusement pour les développeurs Java EE, cela a traditionnellement était très difficile. Les applications développées avant la version Java EE 5 sont étroitement couplé au serveur d'applications, ce qui le rend souvent difficile et contre-productif pour tenter de répliquer les services de conteneur requis dans un environnement autonome.

Pour mettre cela en perspective, regardons les Enterprise JavaBeans tels qu'ils existaient dans EJB 2.1. Sur le papier, tester une classe de bean session ne devrait être qu'un cas d'instanciation la classe bean et en invoquant la méthode métier. Pour les méthodes commerciales triviales, c'est effectivement le cas, mais les choses commencent à se détériorer rapidement une fois que les dépendances sont impliquées. Par exemple, considérons une méthode métier qui doit invoquer une autre entreprise méthode à partir d'un bean session différent.

La recherche de dépendances était la seule option dans EJB 2.1, donc si la méthode métier doit accéder à JNDI pour obtenir une référence à l'autre bean session, soit JNDI doit être travaillé autour ou la classe de bean doit être refactorisée pour que le code de recherche puisse être remplacé avec une version spécifique au test. Si le code utilise le localisateur de service[3] modèle, nous avons un problème plus important car une méthode statique singleton est utilisée pour obtenir la référence du bean. La seule solution pour tester les beans qui utilisent des localisateurs de service en dehors du conteneur est de refactoriser les classes de bean afin que la logique de localisation puisse être remplacée dans un cas de test.

[3] Alur, Deepak, John Crupi et Dan Malks. *Modèles J2EE de base: meilleures pratiques et conception Stratégies, deuxième édition* . Upper Saddle River, NJ: Prentice Hall PTR, 2003, p. 315.

Épisode 715

Ensuite, nous avons le problème du bean dépendant lui-même. La classe bean ne mettre en œuvre l'interface métier, de sorte qu'elle ne puisse pas être simplement instanciée et mise à disposition au haricot que nous essayons de tester. Au lieu de cela, il devra être sous-classé pour implémenter le l'interface métier et les stubs pour un certain nombre de méthodes EJB de bas niveau devront être fourni car l'interface métier d'EJB 2.1 étend en fait une interface qui est implémenté en interne par le serveur d'application.

Même si cela fonctionne, que se passe-t-il si nous rencontrons un conteneur géré bean entité? Non seulement nous avons les mêmes problèmes en ce qui concerne les interfaces impliquée mais la classe bean est également abstraite, avec toutes les propriétés d'état persistantes non mis en œuvre. Nous pourrions les implémenter, mais notre cadre de test démarrerait rapidement pour dépasser le code de l'application. Nous ne pouvons même pas simplement les exécuter contre la base de données car nous peut avec le code JDBC car une grande partie de la logique du bean entité, de la maintenance des relations, et les autres opérations de persistance ne sont disponibles qu'à l'intérieur d'un conteneur EJB.

Le sale secret de nombreuses applications écrites à l'aide d'anciennes versions de Java EE est

qu'il y a peu ou pas de tests du tout par les développeurs. Les développeurs écrivent, empaquetent et déploient applications; testez-les manuellement via l'interface utilisateur; et j'espère que le
Le groupe d'assurance qualité peut rédiger un test fonctionnel qui vérifie chaque fonctionnalité. C'est juste trop beaucoup de travail pour tester des composants individuels en dehors du serveur d'applications.

C'est là qu'interviennent EJB, CDI et JPA. Dans les versions ultérieures d'EJB, une classe de bean session est une classe Java simple pour les beans locaux. Aucune interface EJB spéciale n'a besoin d'être étendue ou mis en œuvre. De même avec les beans CDI. Pour tester l'unité de la logique dans une session ou un bean CDI, nous pouvons souvent simplement l'instancier et l'exécuter. Si le haricot dépend d'un autre haricot, nous peut instancier ce bean et l'injecter manuellement dans le bean testé. Si vous êtes tester le code qui utilise le gestionnaire d'entités et que vous souhaitez vérifier qu'il interagit avec le base de données comme vous vous y attendez, il suffit de démarrer le gestionnaire d'entités dans Java SE et de faire utilisation complète du gestionnaire d'entités en dehors du serveur d'applications.

Dans ce chapitre, nous montrons comment prendre un bean et du code JPA à partir d'un Java EE l'application et l'exécuter en dehors du conteneur à l'aide de tests unitaires et d'intégration approches. C'est beaucoup plus facile que par le passé.

JUnit

Le framework de test JUnit est une norme de facto pour tester les applications Java. JUnit est un cadre de test unitaire simple qui permet d'écrire les tests sous forme de classes Java. Ces Java les classes sont ensuite regroupées et exécutées dans des suites à l'aide d'un testeur qui est lui-même un

Épisode 716

classe Java simple. De cette conception simple, toute une communauté a émergé pour fournir extensions de JUnit et intégrez-le dans tous les principaux environnements de développement.

Malgré son nom, les tests unitaires ne sont que l'une des nombreuses choses que JUnit peut être utilisé pour. Il a été étendu pour prendre en charge les tests de sites Web, le stubbing automatique des interfaces pour les tests, les tests d'accès concurrentiel et les tests de performances. Beaucoup d'assurance qualité Les groupes utilisent désormais JUnit dans le cadre du mécanisme d'automatisation pour exécuter des suites entières de tests fonctionnels de bout en bout.

Pour nos besoins, nous examinons JUnit dans le contexte de ses racines de test unitaire, ainsi que stratégies qui lui permettent d'être utilisé comme cadre de test d'intégration efficace. Collectivement nous considérons ces deux approches simplement comme des tests de développement car elles sont écrites par développeurs pour aider à la qualité globale et au développement d'une application.

Nous supposons que vous êtes familiarisé avec JUnit 4 (qui utilise des annotations) à ce point. Des articles d'introduction et des didacticiels sont disponibles sur le site Web de JUnit à l'adresse www.junit.org . De nombreux livres et autres ressources en ligne traitent des tests avec JUnit dans de nombreux détails.

Test unitaire

Cela peut sembler contre-intuitif au début, mais l'un des aspects les plus intéressants entités, c'est qu'elles peuvent participer à des tests sans avoir besoin d'un serveur d'applications en cours d'exécution ou base de données en direct. Dans les sections suivantes, nous examinons le test direct des classes d'entités et l'utilisation d'entités dans le cadre de tests pour les composants Java EE. Nous discutons également de la façon de tirer parti injection de dépendances dans les tests unitaires et comment gérer la présence d'interfaces JPA.

Entités de test

Il est peu probable que les entités soient testées de manière approfondie de manière isolée. La plupart des méthodes sur les entités sont de simples getters ou setters qui se rapportent à l'état persistant de l'entité ou à son des relations. Les méthodes commerciales peuvent également apparaître sur les entités, mais sont moins courantes. Dans de nombreuses applications, les entités ne sont guère plus que des JavaBeans de base.

En règle générale, les méthodes de propriété ne nécessitent généralement pas de tests explicites. Vérifier qu'un setter attribue une valeur à un champ et le getter correspondant récupère la même valeur est ne pas tester l'application autant que le compilateur. Sauf s'il y a un effet secondaire dans un ou les deux méthodes, getters et setters sont trop simples à casser et donc trop simples pour justifier des tests.

Épisode 717

Les éléments clés à rechercher pour déterminer si une entité justifie ou non les tests individuels sont des effets secondaires d'une méthode getter ou setter (comme les données règles de transformation ou de validation) et la présence de méthodes métiers. L'entité Le listing 15-1 contient une logique non triviale qui justifie des tests spécifiques.

Annonce 15-1. Une entité qui valide et transforme les données

```
@Entité
Département de classe publique {
    @Id ID de chaîne privé;
    nom de chaîne privé;
    @OneToMany (mappedBy = "département")
    les employés de la collection privée <Employee>;

    public String getId () {identifiant de retour; }
    public void setId (ID de chaîne) {
        si (id.length ()! = 4) {
            lancer une nouvelle IllegalArgumentException (
                «Les identificateurs de service doivent comporter quatre caractères»);
        }
        this.id = id.toUpperCase ();
    }

    // ...
}
```

La méthode setId () valide à la fois le format de l'identifiant de service et transforme la chaîne en majuscules. Ce type de logique et le fait que le réglage L'identifiant peut en fait provoquer la levée d'une exception suggère que les tests seraient digne d'intérêt. Tester ce comportement est simplement une question d'instanciation de l'entité et en invoquant le setter avec des valeurs différentes. Référencement15-2 montre un ensemble de tests possible.

Annonce 15-2. Tester une méthode Setter pour les effets secondaires

```
Public class DepartmentTest {

    @Tester
    public void testValidDepartmentId () lève l'exception {
        Département dept = nouveau département ();
```


Épisode 718


```
        dept.setId ("NA65");
        Assert.assertEquals ("NA65", dept.getId ());
```

```
    }

    @Tester
    public void testDepartmentIdInvalidLength () lève l'exception {
        Département dept = nouveau département ();
        essayez {
            dept.setId ("NA6");
            Assert.fail ("Les identificateurs de service doivent être composés de quatre caractères");
        } catch (IllegalArgumentException e) {
        }
    }

    @Tester
    public void testDepartmentIdCase () lève l'exception {
        Département dept = nouveau département ();
        dept.setId ("na65");
        Assert.assertEquals ("NA65", dept.getId ());
    }
}
```

Test d'entités dans les composants

Le candidat au test le plus probable pour les entités n'est pas l'entité mais le code d'application qui utilise l'entité dans le cadre de sa logique métier. Pour la plupart des applications, cela signifie tester beans session, beans gérés, beans CDI, beans Spring ou toute autre saveur d'entreprise composant que vous utilisez. Si les entités sont obtenues ou initialisées en dehors du périmètre du composant, le test est facilité dans le sens où la classe d'entité peut simplement être instancié, rempli de données d'entité et défini dans la classe de bean pour le test. Quand utilisé comme objet de domaine dans le code d'application, une entité n'est pas différente de tout autre Java classe. Vous pouvez effectivement prétendre que ce n'est pas du tout une entité.

Bien sûr, le test unitaire d'un bean ne se résume pas à une simple instanciation d'entités à utilisé avec une méthode commerciale. Nous devons également nous préoccuper des dépendances qui le bean a pour mettre en œuvre sa logique métier. Ces dépendances sont généralement manifesté sous forme de champs sur la classe de bean qui sont remplis à l'aide d'une forme de dépendance injection (ou dans certains cas recherche de dépendances).

Épisode 719

L'injection comprend l'enrichissement de test suivant:

- @Resource (référence à n'importe quelle entrée JNDI)
- @EJB (Injecte EJB, local / distant)
- @Inject (beans CDI)
- @PersistenceContext (contexte de persistance JPA)

Lors de l'écriture de tests unitaires, le but est d'introduire l'ensemble minimum de dépendances nécessaire pour mettre en œuvre un test particulier. Si vous testez une méthode commerciale qui nécessite pour invoquer une méthode sur une interface séparée, vous ne devez vous soucier que de fournir un version stubbed de l'interface. Si le bean utilise une source de données mais n'est pas pertinent pour votre test, alors idéalement, vous voulez l'ignorer entièrement.

L'injection de dépendances est la clé d'un test unitaire efficace. En supprimant de telles choses en tant qu'API JNDI à partir du code bean et éliminant le besoin du localisateur de service pattern, vous pouvez vous assurer que la classe du bean a quelques dépendances sur le conteneur. Il vous suffit d'instancier l'instance du bean et d'injecter manuellement les ressources requises, dont la majorité sera soit d'autres beans de l'application, soit spécifiques au test implémentations d'une interface standard.

Comme nous l'avons expliqué au chapitre 3 , la forme d'injection de setter de l'injection de dépendances est le plus simple à utiliser dans les tests unitaires. Comme les méthodes setter sont presque toujours publiques, ils peuvent être appelés directement par le cas de test pour attribuer une dépendance à la classe de bean. L'injection de champ est toujours facile à gérer, tant que le champ utilise la portée du package car la convention pour les tests unitaires est d'utiliser le même nom de package que la classe en cours tested.

Lorsque la dépendance est un autre bean, vous devez choisir si toutes les dépendances de la classe de bean requise doivent être satisfaites ou si un test spécifique la version du bean doit être utilisée à la place. Si la méthode commerciale de la personne à charge bean n'affecte pas le résultat du test, cela ne vaut peut-être pas la peine d'établir la dépendance complète. À titre d'exemple, considérons le bean montré dans l'extrait 15-3. Nous avons montré une méthode unique de calcul des années de service pour un employé qui récupère un Instance Employee utilisant le bean de session EmployeeService.

Épisode 720

Annonce 15-3. Utilisation du bean EmployeeService dans une autre méthode métier

```
@Apatride
public class VacationBean {
    public statique final long MILLIS_PER_YEAR = 1000 * 60 * 60 * 24 * 365;
    @EJB EmployeeService empService;

    public int getYearsOfService (int empId) {
        Employé emp = empService.findEmployee (empId);
        long courant = System.currentTimeMillis ();
        long début = emp.getStartDate (). getTime ();
        return (int) ((courant - début) / MILLIS_PER_YEAR);
    }

    // ...
}
```

Parce que la seule chose nécessaire pour vérifier la méthode getYearsOfService () est un seule instance Employee avec une valeur de date de début, il peut ne pas être nécessaire d'utiliser le EmployeeService bean, en particulier s'il a ses propres dépendances externes qui pourraient rendent difficile l'instanciation. Une sous-classe simple de la classe EmployeeService qui renvoie une instance d'entité préconfigurée pour le test est plus que suffisante. En fait, la capacité de spécifier une valeur de retour connue de la méthode findEmployee () rend l'ensemble test beaucoup plus facile à mettre en œuvre. Le Listing 15-4 démontre l'utilisation d'une sous-classe spécifique au test implémentation d'un bean. L'implémentation est définie comme une classe interne anonyme dans la classe de test. Créer une implémentation spécifiquement pour un test s'appelle se *moquer* du class et l'instance instanciée est appelée *objet fictif* .

Annonce 15-4. Création d'une implémentation spécifique au test d'un bean

```
public class VacationBeanTest {
    @Tester
    public void testYearsOfService () lève l'exception {
        VacationBean bean = nouveau VacationBean ();
        bean.empService = new EmployeeService () {
            public Employee findEmployee (int id) {
                Employé emp = nouvel employé ();
```

Épisode 721


```
                emp.setStartDate (nouvelle heure (System.currentTimeMillis () -
                                                  VacationBean.MILLIS_PER_YEAR * 5));
                return emp;
            }

            // ...
        };
        int yearsOfService = bean.getYearsOfService (0);
        Assert.assertEquals (5, yearsOfService);
    }

    // ...
}
```

Le gestionnaire d'entités dans les tests unitaires

Les interfaces EntityManager et Query présentent un défi pour l'unité d'écriture des développeurs des tests. Le code qui interagit avec le gestionnaire d'entité peut être différent du simple (persister objet) au complexe (émettre une requête JP QL et obtenir les résultats). Il y en a deux approches de base pour gérer la présence d'interfaces standard:

- Introduisez une sous-classe qui remplace les méthodes contenant une entité les opérations de gestionnaire ou de requête avec des versions spécifiques aux tests qui ne interagir avec JPA.
- Fournir des implémentations personnalisées des interfaces standard qui peuvent être utilisé de manière prévisible pour les tests.

Avant de couvrir ces stratégies en détail, considérez l'implémentation du bean session montré dans la liste 15-5 qui fournit un service d'authentification simple. Pour une classe aussi simple, le test unitaire est étonnamment difficile. Les opérations du gestionnaire d'entités sont intégrées directement dans la méthode authenticate (), couplant l'implémentation à JPA.

Annonce 15-5. Session Bean qui effectue une authentification de base

```
@Apatride
public class UserService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;
```


Épisode 722


```
    public User authenticate (String userId, String password) {
        Utilisateur utilisateur = em.find (User.class, userId);
        if (utilisateur! = null) {
            if (password.equals (user.getPassword ())) {
                retour utilisateur;
```

```
            }
        }
        return null;
    }
}
```

La première technique que nous démontrons pour rendre cette classe testable est d'introduire un sous-classe qui élimine les appels de gestionnaire d'entités. Pour l'exemple UserServiceBean montré dans le Listing 15-5 , l'accès au gestionnaire d'entités doit d'abord être isolé à une méthode distincte avant il peut être testé. Le listing 15-6 démontre une telle refactorisation.

Annonce 15-6. Isolation des opérations Entity Manager pour les tests

```
@Apatride
public class UserService {
    @PersistenceContext (unitName = "EmployeeService")
    EntityManager em;

    public User authenticate (String userId, String password) {
        Utilisateur utilisateur = findUser (userId);
        // ...
    }

    Utilisateur findUser (String userId) {
        return em.find (User.class, userId);
    }
}
```

Une fois cette refactorisation terminée, la méthode authenticate () n'a plus dépendance vis-à-vis du gestionnaire d'entités. La classe UserService peut maintenant être sous-classée pour testing, en remplaçant la méthode findUser () par une version spécifique au test qui renvoie un résultat bien connu. Le listing 15-7 montre un cas de test complet utilisant cette technique.

Épisode 723

Annonce 15-7. Utilisation d'une sous-classe pour éliminer les dépendances d'Entity Manager

```
public class UserServiceTest {
    static final String USER_ID = "test_id";
    static final String PASSWORD = "test_password";
    static final String INVALID_USER_ID = "test_user";

    @Tester
    public void testAuthenticateValidUser () lève l'exception {
        Service MockUserService = nouveau MockUserService ();
        User user = service.authenticate (USER_ID, PASSWORD);
        Assert.assertNotNull (utilisateur);
        Assert.assertEquals (USER_ID, user.getName ());
        Assert.assertEquals (PASSWORD, user.getPassword ());
    }

    @Tester
    public void testAuthenticateInvalidUser () lève l'exception {
        Service MockUserService = nouveau MockUserService ();
        Utilisateur utilisateur = service.authenticate (INVALID_USER_ID, PASSWORD);
        Assert.assertNull (utilisateur);
    }

    class MockUserService étend UserService {
```

```
        utilisateur privé;

        public MockUserService () {
            utilisateur = nouvel utilisateur ();
            user.setName (USER_ID);
            user.setPassword (PASSWORD);
        }

        Utilisateur findUser (String userId) {
            if (userId.equals (user.getName ())) {
                retour utilisateur;
            }
            return null;
        }
    }
}
```


Épisode 724

Ce cas de test a l'avantage de laisser la méthode authenticate () d'origine implémentation intacte, ne remplaçant que la méthode findUser () pour le test. Cela marche bien pour les classes qui ont été refactorisées pour isoler les opérations de persistance, mais ces des changements ne peuvent pas toujours être apportés. L'alternative est de se moquer de l'interface EntityManager. Référencement 15-8 illustre cette approche.

Annonce 15-8. Utilisation d'un Mock Entity Manager dans un test unitaire

```
public class UserServiceTest2 {
    static final String USER_ID = "test_id";
    static final String PASSWORD = "test_password";
    static final String INVALID_USER_ID = "test_user";

    @Tester
    public void testAuthenticateValidUser () lève l'exception {
        Service UserService = nouveau UserService ();
        service.em = new TestEntityManager (USER_ID, PASSWORD);
        User user = service.authenticate (USER_ID, PASSWORD);
        Assert.assertNotNull (utilisateur);
        Assert.assertEquals (USER_ID, user.getName ());
        Assert.assertEquals (PASSWORD, user.getPassword ());
    }

    @Tester
    public void testAuthenticateInvalidUser () lève l'exception {
        Service UserService = nouveau UserService ();
        service.em = new TestEntityManager (USER_ID, PASSWORD);
        Utilisateur utilisateur = service.authenticate (INVALID_USER_ID, PASSWORD);
        Assert.assertNull (utilisateur);
    }

    La classe TestEntityManager étend MockEntityManager {
        utilisateur privé;

        public TestEntityManager (String user, String password) {
            this.user = nouvel utilisateur ();
            this.user.setName (utilisateur);
            this.user.setPassword (mot de passe);
        }
```

Épisode 725


```
        public <T> T find (Classe <T> entityClass, Object pk) {
            if (entityClass == User.class && ((String) pk) .equals (user.
            getName ())) {
                return (T) utilisateur;
            }
            return null;
        }
    }
}
```

L'avantage de cette approche par rapport au sous-classement est qu'elle laisse le haricot d'origine classe inchangée tout en lui permettant d'être testé unitaire. La classe MockEntityManager référencé dans le test est une implémentation concrète de l'interface EntityManager avec des définitions de méthode vides. Toutes les méthodes qui retournent une valeur retournent null ou un équivalent à la place. En le définissant séparément, il peut être réutilisé pour d'autres cas de test. Beaucoup les suites de tests unitaires contiennent un petit ensemble d'interfaces simulées qui peuvent être réutilisées plusieurs tests.

Astuce Vérifier www.mockobjects.com pour plus d'informations sur l'objet factice techniques et outils open source pour aider à la création d'objet simulé.

Test d'intégration

Les tests d'intégration, pour nos besoins, sont une extension des tests unitaires qui composants d'une application Java EE et les exécute en dehors d'un serveur d'applications. Contrairement aux tests unitaires, dans lesquels nous nous sommes donné beaucoup de mal pour éviter le gestionnaire d'entités, en tests d'intégration, nous l'adoptons et tirons parti du fait qu'il peut être utilisé dans Java SE.

Les sections suivantes explorent l'utilisation de JPA en dehors d'un serveur d'applications dans l'ordre pour tester la logique d'application avec une base de données en direct, mais sans démarrer le serveur d'applications. Pour mieux se rapprocher de l'environnement d'exécution, le même fournisseur doit être utilisé pour test tel qu'il est utilisé en production.

Épisode 726

Utilisation d'Entity Manager

Dans la liste 15-5 , nous avons démontré un haricot quieffectuéauthentificationbase contre un utilisateur objet récupéré de la base de données. Pour tester l'unité cette classe, un certain nombre de techniques ont été présenté pour remplacer ou simuler l'opération du gestionnaire d'entités. L'inconvénient de cette approche est que le code de test requis pour contourner les dépendances externes dans le code de l'application peut rapidement atteindre un point où il est difficile à maintenir et constitue une source potentielle de bogues.

Au lieu de se moquer du gestionnaire d'entités, une entité gérée par l'application, locale aux ressources manager peut être utilisé pour effectuer des tests sur une base de données en direct. Référencement15-9 montre

une version de test fonctionnel des cas de test UserService.

Annonce 15-9. Test d'intégration pour UserService Bean

```
public class UserServiceTest3 {
    static final String USER_ID = "test_id";
    static final String PASSWORD = "test_password";
    static final String INVALID_USER_ID = "test_user";

    private EntityManagerFactory emf;
    privé EntityManager em;

    @Avant
    public void setUp () {
        emf = Persistence.createEntityManagerFactory ("hr");
        em = emf.createEntityManager ();
        createTestData ();
    }

    @Après
    public void tearDown () {
        if (em! = null) {
            removeTestData ();
            em.close ();
        }
        if (emf! = null) {
            emf.close ();
        }
    }
```



```
    private void createTestData () {
        Utilisateur utilisateur = nouvel utilisateur ();
        user.setName (USER_ID);
        user.setPassword (PASSWORD);
        em.getTransaction (). begin ();
        em.persist (utilisateur);
        em.getTransaction (). commit ();
    }

    private void removeTestData () {
        em.getTransaction (). begin ();
        Utilisateur utilisateur = em.find (User.class, USER_ID);
        if (utilisateur! = null) {
            em.remove (utilisateur);
        }
        em.getTransaction (). commit ();
    }

    @Tester
    public void testAuthenticateValidUser () lève l'exception {
        Service UserService = nouveau UserService ();
        service.em = em;
        User user = service.authenticate (USER_ID, PASSWORD);
        Assert.assertNotNull (utilisateur);
        Assert.assertEquals (USER_ID, user.getName ());
        Assert.assertEquals (PASSWORD, user.getPassword ());
```

```
    }
    @Tester
    public void testAuthenticateInvalidUser () lève l'exception {
        Service UserService = nouveau UserService ();
        service.em = em;
        Utilisateur utilisateur = service.authenticate (INVALID_USER_ID, PASSWORD);
        Assert.assertNull (utilisateur);
    }
}
```


Épisode 728

Test du chapitre 15

Ce cas de test utilise les méthodes fixture setUp () et tearDown () pour créer
Instances EntityManagerFactory et EntityManager utilisant l'API d'amorçage Java SE
puis les ferme une fois le test terminé. Le scénario de test utilise également ces méthodes pour
amorcez la base de données avec des données de test et supprimez-la une fois le test terminé. La larme ()
Il est garanti que la méthode sera appelée même si un test échoue en raison d'une exception. Comme n'importe quel JPA
application dans l'environnement Java SE, un fichier persistence.xml devra se trouver sur le
classpath pour que la classe Persistence amorce une fabrique de gestionnaires d'entités. le
doit contenir les propriétés de connexion JDBC pour se connecter à la base de données, et si le
les classes gérées n'étaient pas déjà répertoriées, des éléments de classe devraient également être ajoutés
pour chaque classe gérée. Si le type de transaction n'a pas été spécifié, il sera par défaut
le type de transaction correct en fonction de l'environnement; sinon, il doit être défini sur
RESOURCE_LOCAL. Cet exemple illustre le modèle de base pour tous les tests d'intégration
qui utilisent un gestionnaire d'entités.

L'avantage de ce style de test par rapport à un test unitaire est qu'aucun effort n'a été nécessaire pour
maquette des interfaces de persistance. Émulation du gestionnaire d'entités et du moteur de requête dans l'ordre
tester le code qui interagit directement avec ces interfaces souffre de rendements décroissants
alors que de plus en plus d'efforts sont consacrés à la préparation d'un environnement de test au lieu d'écrire
des tests. Dans le pire des cas, des résultats de test incorrects se produisent en raison de bogues dans le test
harnais, pas dans le code de l'application. Compte tenu de la facilité avec laquelle JPA peut être utilisé à l'extérieur
le serveur d'applications, ce type d'effort peut être mieux dépensé pour établir un simple
environnement de test de base de données et rédaction de tests fonctionnels automatisés.

Cependant, malgré l'opportunité que présentent les tests en dehors du serveur d'applications,
il faut veiller à ce que ces tests apportent vraiment une valeur ajoutée. Assez souvent, les développeurs
tomber dans le piège de l'écriture de tests qui ne font guère plus que tester les fonctionnalités des fournisseurs
opposé à la vraie logique d'application. Un exemple de cette erreur est de semer une base de données,
exécuter une requête et vérifier que les résultats souhaités sont renvoyés. Cela semble valable
au début, mais tout ce qu'il teste, c'est la compréhension du développeur sur la façon d'écrire une requête.
À moins qu'il y ait un bogue dans la base de données ou le fournisseur de persistance, le test n'échouera jamais.
Une variante plus valide de ce test consiste à démarrer le scénario plus haut dans la pile d'applications
en exécutant une méthode métier sur une façade de session qui lance une requête puis
valider que les objets de transfert résultants sont correctement formés pour une présentation ultérieure en
une page JSP.

Épisode 729

Configuration du test et démontage

De nombreux tests impliquant la persistance nécessitent une sorte de données de test dans la base de données avant le test peut être exécuté. Si l'opération commerciale ne crée pas et vérifie le résultat d'une opération de persistance, la base de données doit déjà contenir des données lisibles et utilisé par le test. Parce que les tests devraient idéalement pouvoir définir et réinitialiser leurs propres données de test avant et après chaque test, nous devons avoir un moyen d'amorcer la base de données de manière appropriée.

Cela semble assez simple; utiliser JDBC pour amorcer la base de données pendant setUp () et à nouveau pendant tearDown () pour le réinitialiser. Mais il y a un danger ici. La plupart de la persistance les fournisseurs utilisent une sorte de mise en cache de données ou d'objets. Chaque fois que les données changent dans base de données sans que le fournisseur de persistance ne le sache, son cache sortira de synchroniser avec la base de données. Dans le pire des cas, cela pourrait entraîner le gestionnaire d'entités opérations pour renvoyer des entités qui ont depuis été supprimées ou qui ont des données périmées.

Il convient de rappeler que ce n'est pas un problème avec le fournisseur de persistance. Mise en cache est une bonne chose et la raison pour laquelle les solutions JPA surpassent souvent Accès JDBC dans les applications en lecture principale. L'implémentation de référence, par exemple, utilise un mécanisme sophistiqué de cache partagé qui s'étend à toute la persistance unité. Lorsque les opérations sont terminées dans un contexte de persistance particulier, les résultats sont fusionnés dans le cache partagé afin qu'ils puissent être utilisés par une autre persistance contextes. Cela se produit si le gestionnaire d'entités et le contexte de persistance sont créés dans Java SE ou Java EE. Par conséquent, vous ne pouvez pas supposer que la fermeture d'un gestionnaire d'entités efface tester les données du cache.

Il existe plusieurs approches que nous pouvons utiliser pour garder le cache cohérent avec notre base de données de test. Le premier, et le plus simple, consiste à créer et supprimer des données de test à l'aide de l'entité directeur. Toute entité conservée ou supprimée à l'aide du gestionnaire d'entités sera toujours conservée compatible avec le cache. Pour les petits ensembles de données, c'est très facile à réaliser. C'est le approche que nous avons utilisée dans l'extrait 15-9.

Pour des ensembles de données plus volumineux, cependant, il peut être fastidieux de créer et de gérer des données de test en utilisant des entités. Les extensions JUnit telles que DbUnit [4] permettent de définir les données de départ en XML puis chargés en masse dans la base de données avant le début de chaque test. Donc, étant donné que le le fournisseur de persistance ne connaîtra pas ces données, comment pouvons-nous encore les utiliser? La première La stratégie consiste à établir un ensemble de données de test en lecture seule. Tant que les données ne sont jamais changé, peu importe que l'entité existe ou non dans le cache du fournisseur. le la deuxième stratégie consiste soit à utiliser des ensembles de données spéciaux pour les opérations à modifier

[4] Visitez http://dbunit.sourceforge.net/ pour plus d'informations.

Épisode 730

tester les données sans les créer ou pour s'assurer que ces modifications ne sont jamais engagé. Si la transaction de mise à jour de la base de données est annulée, la base de données et l'état du cache restera cohérent.

Une autre option consiste à utiliser la propriété javax.persistence.sql-load-script-source décrit au chapitre 14 . Créer un script et laisser le fournisseur l'exécuter au démarrage est un moyen simple de précharger la base de données. Cependant, comme il s'agit d'un niveau d'unité de persistance propriété, cela ne se produira qu'une seule fois au démarrage de l'unité de persistance, ce qui le rend moins pratique à faire test par test.

Le dernier point à considérer est l'invalidation explicite du cache. Avant JPA 2.0, accès à le cache de deuxième niveau était propre au fournisseur. Comme discuté au chapitre 12 , nous pouvons maintenant utilisez l'interface Cache pour effacer explicitement le cache de deuxième niveau entre les tests. Le La méthode suivante montre comment invalider tout le cache de deuxième niveau donné toute instance d'EntityManagerFactory:

```
public static void clearCache (EntityManagerFactory emf) {
    emf.getCache (). evictAll ();
}
```

S'il y a des gestionnaires d'entités ouverts, l'opération clear () sur chacun doit être invoquée également. Comme nous l'avons vu précédemment, le contexte de persistance est un ensemble localisé de changements transactionnels. Il utilise les données du cache partagé mais est en fait un structure de données distincte.

Changement de configurations pour les tests

L'un des avantages de JPA est que les métadonnées spécifiées sous forme d'annotation peuvent être remplacé ou remplacé par des métadonnées spécifiées au format XML. Cela nous offre une possibilité de développer une application ciblant la plateforme de base de données de production et puis fournissez un autre ensemble de mappages (même des définitions de requête) ciblés sur un test environnement. Bien qu'il s'agisse d'une pratique courante et qu'elle présente ses avantages, il convient de noter que si vous exécutez sur une base de données de test avec d'autres mappages et définitions de requête, il y aura clairement au moins quelques différences entre l'installation de test et l'exécution dans production. Des tests de production seront toujours nécessaires, mais des tests plus tôt le cycle sur une base de données de test peut être effectué sur une base de données plus pratique ou plus accessible plate-forme et peut attraper quelques bogues plus tôt dans le cycle.

Épisode 731

Dans le contexte des tests, le mécanisme d'amorçage Java SE utilisera la persistance. xml situé dans le répertoire META-INF sur le chemin de classe. Tant que la persistance la définition d'unité à l'intérieur de ce fichier porte le même nom que celui sur lequel l'application a été écrite à, la version de test peut le recibler si nécessaire pour répondre aux besoins du test d'intégration.

Il existe deux utilisations principales de cette approche. La première consiste à spécifier les propriétés dans le fichier persistence.xml spécifique aux tests. Pour de nombreux développeurs, cela signifiera fournir des informations de connexion JDBC à une base de données locale afin que les tests ne se heurtent pas avec d'autres développeurs sur une base de données partagée.

La deuxième utilisation majeure d'un fichier persistence.xml personnalisé est de personnaliser la base de données mappages pour le déploiement sur une plate-forme de base de données complètement différente. Par exemple, si Oracle est votre base de données de production et vous ne souhaitez pas exécuter la base de données complète sur votre machine locale, vous pouvez ajuster les informations de mappage pour cibler une base de données intégrée comme Apache Derby.

> Remarque au risque de paraître quelque peu biaisé, pourrions-nous suggérer humblement Oracle Xe. il représente la puissance de la base de données Oracle adaptée à machine individuelle sans frais. De nombreux exemples de ce livre (y compris le des exemples de requêtes sQL avancées) ont été développés sur Oracle Xe.

Comme exemple de cas où cela serait nécessaire, considérons une application qui utilise le séquençage natif de la base de données Oracle. Derby n'a pas d'équivalent, donc des générateurs de table doivent être utilisés à la place. Tout d'abord, considérons un exemple d'entité qui utilise un générateur de séquence natif:

```
@Entité
Téléphone public class {
    @SequenceGenerator (nom = "Phone_Gen", sequenceName = "PHONE_SEQ")
    @Id @GeneratedValue (générateur = "Phone_Gen")
    id int privé;
```

```
    // ...
}
```

La première étape pour faire fonctionner cette entité sur Derby est de créer un mappage XML fichier qui remplace la définition du générateur Phone_Gen pour utiliser un générateur de table.

Épisode 732

Test du chapitre 15

Le fragment suivant d'un fichier de mappage montre comment remplacer la séquence générateur avec un générateur de table:

```
<entity-mappings>
    ...
    <table-generator name = "Phone_Gen", table = "ID_GEN",
                                pk-column-value = "PhoneId">
    ...
</entity-mappings>
```

C'est la même technique que nous avons appliquée au chapitre 13 lorsque nous avons discuté de la priorité générateur de séquence.

Enfin, nous devons créer un nouveau fichier persistence.xml qui référence ce mappage fichier. Si les remplacements ont été placés dans un fichier de mappage appelé derby-overrides.xml, le la configuration d'unité de persistance suivante appliquerait les remplacements de mappage:

```
<persistance>
    <persistence-unit name = "hr">
        ...
        <mapping-file> derby-overrides.xml </mapping-file>
        ...
    </persistence-unit>
</persistence>
```

Contrairement au fichier de mappage, qui définit rarement les remplacements, toutes les informations était présent dans le fichier de production persistence.xml doit être copié dans le fichier spécifique au test version. La seule exception à cela concerne les propriétés de connexion JDBC, qui doivent être personnalisés pour l'instance Derby intégrée.

Minimiser les connexions à la base de données

Les tests d'intégration s'exécutent plus lentement que les tests unitaires en raison de la nature de l'interaction avec la base de données, mais ce qui n'est peut-être pas évident d'après le cas de test présenté dans la liste 15-9 est que deux séparés les connexions sont établies à la base de données, une pour chaque testAuthenticateValidUser () et les tests testAuthenticateInvalidUser (). JUnit instancie en fait une nouvelle instance de la classe de cas de test à chaque fois qu'elle exécute une méthode de test, exécutant chacun setUp () et tearDown () le temps aussi. La raison de ce comportement est de minimiser le risque de stockage des données dans les champs d'un cas de test interférant avec l'exécution d'un autre.

Épisode 733

Bien que cela fonctionne bien pour les tests unitaires, cela peut conduire à des performances inacceptables pour tests d'intégration. Pour contourner cette limitation, les @BeforeClass et @AfterClass

Les fonctionnalités de JUnit 4 peuvent être utilisées pour créer des appareils qui ne s'exécutent qu'une seule fois pour tous les tests d'un classe. Référencement15-10 montre une classe de suite de tests qui utilise cette fonctionnalité au niveau du toute la suite de tests.

Annonce 15-10. Configuration unique de la base de données pour les tests d'intégration

```
@RunWith (Suite.class)
@ Suite.SuiteClasses ({UserServiceTest3.class})
public class DatabaseTest {
    public static EntityManagerFactory emf;

    @Avant les cours
    public static void setUpBeforeClass () lève l'exception {
        emf = Persistence.createEntityManagerFactory ("hr");
    }

    @Après les cours
    public static void tearDownAfterClass () jette une exception {
        if (emf! = null) {emf.close (); }
    }}
```

En utilisant cette suite de tests comme point de départ, tous les cas de test ajoutés à @ Suite.SuiteClasses l'annotation peut avoir accès à la statique EntityManagerFactory correctement renseignée champ de la classe DatabaseTest. La méthode setUp () de chaque cas de test n'a plus besoin que de pour référencer cette classe pour obtenir la fabrique au lieu de la créer à chaque fois. Le suivant L'exemple illustre la modification requise pour le scénario de test UnitServiceTest3:

```
@Avant
public void setUp () {
    emf = DatabaseTest.emf;
    em = emf.createEntityManager ();
    createTestData ();
}
```


Épisode 734

C'est une technique utile pour minimiser le coût d'acquisition de ressources coûteuses, mais il faut veiller à ce que les effets secondaires d'un test n'interfèrent pas accidentellement avec l'exécution d'autres tests. Étant donné que tous les tests partagent la même usine de gestionnaires d'entités, les données peut être mis en cache ou les paramètres peuvent être modifiés (pris en charge par certaines usines de gestionnaires d'entités) qui ont un impact inattendu plus tard. Tout comme il est nécessaire de conserver les tables de la base de données propre entre les tests, toute modification apportée à la fabrique du gestionnaire d'entités doit être annulée lorsque le le test se termine, que le résultat soit un succès ou un échec. C'est généralement une bonne idée pour vider le cache, comme nous l'avons montré dans la section «Configuration des tests et démontage».

Composants et persistance

Le plus souvent, les beans d'un test d'intégration ne sont pas différents des beans d'une unité tester. Vous instanciez le bean, fournissez les dépendances nécessaires et exécutez le tester. Là où nous commençons à diverger, c'est lorsque nous prenons en compte des problèmes tels que les transactions gestion et plusieurs instances de bean collaborant ensemble pour implémenter un seul cas d'utilisation. Dans les sections suivantes, nous discutons des techniques pour gérer des bean plus complexes scénarios lors des tests en dehors du conteneur.

Gestion des transactions

Les transactions sont au cœur de chaque application d'entreprise. Nous avons fait cette déclaration de retour au chapitre 3 et je l'ai ramené à la maison au chapitre6, démontrant toutes les différentes manières de quels gestionnaires d'entités et contextes de persistance peuvent croiser différentes transactions des modèles. Cela pourrait donc être une surprise d'apprendre que lorsqu'il s'agit d'écrire tests d'intégration, nous pouvons souvent contourner les exigences transactionnelles strictes du application pour développer facilement des tests en dehors du conteneur.

Les sections suivantes expliquent quand les transactions sont vraiment nécessaires et comment traduire les modèles de transaction gérés par conteneur et gérés par bean de Java EE Serveur dans votre environnement de test.

Quand utiliser les transactions

Sauf pour les gestionnaires d'entités gérés par les applications et locaux aux ressources, qui sont moins fréquemment utilisé dans l'environnement Java EE, la gestion des transactions est du ressort des beans session et autres composants qui utilisent JPA. Nous nous concentrons spécifiquement sur la session beans, mais les sujets que nous couvrons s'appliquent également aux opérations de persistance transactionnelle hébergé par tout autre composant compatible avec les transactions.

Épisode 735

La démarcation de transaction pour une méthode de bean session doit être prise en compte soigneusement lors de la rédaction des tests. Malgré l'hypothèse par défaut que les transactions sont utilisées partout dans le serveur d'applications, seul un certain nombre de méthodes nécessitent gestion des transactions à des fins de test. Parce que nous nous concentrons sur les tests persistance, la situation qui nous préoccupe est celle où le gestionnaire d'entité utilisé pour conserver, fusionner ou supprimer des instances d'entité. Nous devons également déterminer si ces entités doivent en fait être conservées dans la base de données.

Dans un environnement de test, nous utilisons une entité gérée par l'application et locale aux ressources les gestionnaires. Rappel du chapitre6 qu'un gestionnaire d'entités géré par une application peut effectuer toutes ses opérations sans transaction active. En effet, invoquer persist () met en file d'attente l'entité à conserver la prochaine fois qu'une transaction démarre et est validée. De plus, nous savons qu'une fois qu'une entité est gérée, elle peut généralement être localisée en utilisant l'opération find () sans avoir besoin d'aller dans la base de données. Compte tenu de ces faits, nous n'avons généralement besoin d'un gestionnaire d'entités traitées que si la méthode commerciale crée ou modifie les entités et exécute une requête qui doit inclure les résultats.

Bien qu'elle ne soit pas requise pour satisfaire la logique métier, une transaction peut également être requise si vous voulez que les résultats de l'opération soient persistants afin qu'ils puissent être analysés à l'aide de autre chose que le gestionnaire d'entités actif. Par exemple, les résultats de l'opération peut être lu à partir de la base de données à l'aide de JDBC et comparé à une valeur connue à l'aide d'un outil de test.

La principale chose que nous voulons souligner ici - avant de regarder comment mettre en œuvre transactions pour les tests de session bean - est-ce que le plus souvent, vous n'avez pas vraiment besoin du tout. Regardez la séquence des opérations que vous testez et déterminez si le le résultat sera affecté d'une manière ou d'une autre - d'abord si les données doivent être écrites dans le base de données, et plus tard si elle doit vraiment être validée dans le cadre du test. Compte tenu de la complexité que la gestion manuelle des transactions peut parfois exiger, n'utilisez que des transactions quand ils sont nécessaires.

Transactions gérées par conteneur

L'un des avantages les plus importants des transactions gérées par conteneurs est qu'elles sont configurés pour les méthodes bean utilisant entièrement des métadonnées. Il n'y a pas de programmation interface invoquée par le bean pour contrôler la transaction autrement que sur le contexte objets[1] , et même cela ne se produit que dans certaines circonstances. Par conséquent, une fois que nous décidons

qu'une méthode bean particulière nécessite qu'une transaction soit active, il suffit de démarrer un transaction au début du test et validez ou annulez les résultats à la fin du test.

[5] Voir la méthode setRollbackOnly () sur l'interface EJBContext.

Épisode 736

Référencement 15-11 montre une méthode bean qui nécessitera une transaction ouverte pendant un examen. La méthode assignEmployeeToDepartment () affecte un employé à un département, puis renvoie la liste des employés actuellement affectés au département en exécutant une requête. Parce que la modification des données et la requête se produisent dans le même transaction, notre scénario de test nécessitera également une transaction.

Annonce 15-11. Méthode commerciale nécessitant une transaction

```
@Apatride
Public class DepartmentServiceBean implémente DepartmentService {
    Chaîne finale statique privée QUERY =
        "SELECT e" +
        "FROM Employé e" +
        "WHERE e.department =? 1 ORDER BY e.name";

    @PersistenceContext
    EntityManager em;

    Liste publique assignEmployeeToDepartment (int deptId, int empId) {
        Département dept = em.find (Department.class, deptId);
        Employé emp = em.find (Employee.class, empId);
        dept.getEmployees (). add (emp);
        emp.setDepartment (dept);
        retourner em.createQuery (QUERY)
                .setParameter (1, dept)
                .getResultList ();
    }

    // ...
}
```

Parce que nous utilisons un gestionnaire d'entités locales aux ressources, nous simulerons transactions gérées par conteneur avec EntityTransaction transactions gérées par le cas de test. Le listing 15-12 montre le scénario de test pour assignEmployeeToDepartment () méthode. Nous avons suivi le même modèle que dans la liste15-9 , donc le setUp () et Les méthodes tearDown () ne sont pas affichées. Avant que la méthode du bean session ne soit appelée, nous créer une nouvelle transaction. Lorsque le test est terminé, nous annulons les modifications car il n'est pas nécessaire de les conserver dans la base de données.

Épisode 737

Annonce 15-12. Tester une méthode commerciale qui nécessite une transaction

```
Public class DepartmentServiceBeanTest {
    // ...
```

```
    private void createTestData () {
        Employé emp = nouvel employé (500, «Scott»);
        em.persist (emp);
        emp = nouvel employé (600, "John");
        em.persist (emp);
        Département dept = nouveau département (700, "TEST");
        dept.getEmployees (). add (emp);
        emp.setDepartment (dept);
        em.persist (dept);
    }

    @Tester
    public void testAssignEmployeeToDepartment () lève une exception {
        DepartmentServiceBean bean = nouveau DepartmentServiceBean ();
        bean.em = em;
        em.getTransaction (). begin ();
        Résultat de la liste = bean.assignEmployeeToDepartment (700, 500);
        em.getTransaction (). rollback ();
        Assert.assertEquals (2, result.size ());
        Assert.assertEquals ("John", ((Employé) result.get (0)). GetName ());
        Assert.assertEquals ("Scott", ((Employé) result.get (1)). GetName ());
    }

    // ...
}
```

Transactions gérées par Bean

Pour un bean qui utilise des transactions gérées par le bean, le problème clé que nous devons affronter avec est l'interface UserTransaction. Il peut ou non être présent dans un haricot donné méthode et peut être utilisé à diverses fins, depuis la vérification de la transaction statut pour marquer la transaction en cours pour annulation, pour validation et annulation

---

Épisode 738

transactions. Heureusement, presque toutes les méthodes UserTransaction ont une corrélation avec l'une des méthodes EntityTransaction. Parce que notre stratégie de test implique une seule instance de gestionnaire d'entités pour un test, nous devons adapter son EntityTransaction implémentation à l'interface UserTransaction.

Référencement 15-13 montre une implémentation de l'interface UserTransaction qui délègue à l'interface EntityTransaction d'une instance EntityManager. La gestion des exceptions a été ajoutée pour convertir les exceptions non vérifiées levées par les opérations EntityTransaction dans les exceptions vérifiées que les clients du L'interface UserTransaction attendra.

Annonce 15-13. Émulation de UserTransaction à l'aide d'EntityTransaction

```
public class EntityUserTransaction implémente UserTransaction {
    privé EntityManager em;

    public EntityUserTransaction (EntityManager em) {
        this.em = em;
    }

    public void begin () lève NotSupportedException {
        if (em.getTransaction (). isActive ()) {
            lancer une nouvelle NotSupportedException ();
```

```
        }
        em.getTransaction (). begin ();
    }

    public void commit () lève RollbackException {
        essayez {
            em.getTransaction (). commit ();
        } catch (javax.persistence.RollbackException e) {
            lancer une nouvelle exception RollbackException (e.getMessage ());
        }
    }

    public void rollback () lève SystemException {
        essayez {
            em.getTransaction (). rollback ();
```


Épisode 739


```
            } catch (PersistenceException e) {
                lancer une nouvelle exception SystemException (e.getMessage ());
            }
        }

        public void setRollbackOnly () {
            em.getTransaction (). setRollbackOnly ();
        }
        public int getStatus () {
            if (em.getTransaction (). isActive ()) {
                return Status.STATUS_ACTIVE;
            } autre {
                return Status.STATUS_NO_TRANSACTION;
            }
        }

        public void setTransactionTimeout (int timeout) {
            throw new UnsupportedOperationException ();
        }
}
```

Notez que nous avons implémenté setTransactionTimeout () pour lever une exception, mais cela ne doit pas nécessairement être le cas. Si le délai d'expiration de la transaction est défini simplement pour éviter que les processus ne prennent trop de temps à se terminer, il peut être prudent d'ignorer lors d'un test d'intégration.

Pour illustrer ce wrapper, considérez d'abord Listing 15-14 , ce qui démontre une variante de l'exemple de Listing 15-11 qui utilise des transactions gérées par bean au lieu de transactions gérées par des conteneurs.

Annonce 15-14. Utilisation de transactions gérées par Bean

```
@Apatride
@TransactionManagement (TransactionManagementType.BEAN)
Public class DepartmentServiceBean implémente DepartmentService {
    // ...
    @Resource UserTransaction tx;
```

Épisode 740


```
    Liste publique assignEmployeeToDepartment (int deptId, int empId) {
        essayez {
            tx.begin ();
            Département dept = em.find (Department.class, deptId);
            Employé emp = em.find (Employee.class, empId);
            dept.getEmployees (). add (emp);
            emp.setDepartment (dept);
            tx.commit ();
            retourner em.createQuery (QUERY)
                    .setParameter (1, dept)
                    .getResultList ();
        } catch (Exception e) {
            // gérer les exceptions de transaction
            // ...
        }
    }

    // ...
}
```

Utiliser le wrapper UserTransaction est simplement une question de l'injecter dans une session bean qui a déclaré une dépendance sur UserTransaction. Parce que l'emballage tient sur une instance de gestionnaire d'entités, il peut commencer et terminer les transactions EntityTransaction comme requis à partir du code d'application testé. Référencement15-15 montre le cas de test révisé de la liste 15-12 utilisation de ce wrapper pour émuler le bean géré transactions.

Annonce 15-15. Exécution d'un test avec des transactions gérées par le bean émulé

```
Public class DepartmentServiceBeanTest {
    // ...

    @Tester
    public void testAssignEmployeeToDepartment () lève une exception {
        DepartmentServiceBean bean = nouveau DepartmentServiceBean ();
        bean.em = em;
        bean.tx = nouvelle EntityUserTransaction (em);
        Résultat de la liste = bean.assignEmployeeToDepartment (700, 500);
```


Épisode 741


```
            Assert.assertEquals (2, result.size ());
            Assert.assertEquals ("John", ((Employé) result.get (0)). GetName ());
            Assert.assertEquals ("Scott", ((Employé) result.get (1)). GetName ());
        }

        // ...
    }
```

Notez que bien que l'interface UserTransaction soit utilisée, cela ne signifie pas

réellement nécessaire pour tout test particulier. Si l'état de la transaction n'affecte pas le résultat du test, envisagez d'utiliser une implémentation de l'interface UserTransaction qui ne faire n'importe quoi. Par exemple, l'implémentation de UserTransaction indiquée dans la liste15-16 convient dans tous les cas où la démarcation de transaction est déclarée mais inutile.

Annonce 15-16. Une transaction utilisateur stubbed

```
La classe publique NullUserTransaction implémente UserTransaction {
    public void begin () {}
    public void commit () {}
    public void rollback () {}
    public void setRollbackOnly () {}
    public int getStatus () {
        return Status.STATUS_NO_TRANSACTION;
    }
    public void setTransactionTimeout (int timeout) {}
}
```

Le cas de test montré dans la liste 15-12 aurait également pu tester le bean de Listing15-14 si le wrapper UserTransaction vide de la liste 15-16 a également été injecté dans le haricot exemple. Cela désactiverait les transactions gérées par le bean de la méthode commerciale réelle, permettant d'utiliser à la place les transactions du scénario de test.

Gestionnaires d'entités gérés par des conteneurs

Le type de gestionnaire d'entités par défaut pour la plupart des beans est géré par conteneur et par transaction portée, mais dans le cas des beans session avec état, ils peuvent également être étendus. Dans les deux cas cas, l'objectif du test en dehors du conteneur est de mapper l'entité gérée par l'application gestionnaire utilisé par le test pour l'un de ces types de gestionnaire d'entités géré par conteneur.

Épisode 742

La bonne nouvelle pour tester le code qui utilise le gestionnaire d'entités étendu est que le Le gestionnaire d'entités géré par l'application offre presque exactement le même ensemble de fonctionnalités. Ça peut généralement être injecté dans une instance de bean session avec état à la place d'une entité étendue manager, et la logique métier doit fonctionner sans changement dans la plupart des cas.

De même, la plupart du temps, le gestionnaire d'entités transactionnelles fonctionne très bien lorsqu'un gestionnaire d'entités géré par une application est utilisé à sa place. Le principal problème que nous dans le cas des gestionnaires d'entités à portée transactionnelle, il faut faire face au détachement. Lorsqu'une transaction se termine, toutes les entités gérées sont détachées. En termes de test, que signifie simplement que nous devons nous assurer que clear () est invoqué à la limite de la transaction pour notre gestionnaire d'entités de test.

Nous devrons peut-être également traiter la question de la propagation. À certains égards, la propagation est facile dans un environnement de test. Si vous injectez le même instance de gestionnaire d'entités en deux instances de bean, les beans partagent la même persistance contexte comme si le gestionnaire d'entités était propagé avec la transaction. En fait, c'est il est beaucoup plus probable que vous deviez injecter plusieurs gestionnaires d'entités pour simuler le absence intentionnelle de propagation (comme un bean qui invoque une méthode REQUIRES_NEW sur un autre haricot) que vous devrez faire quelque chose de spécial pour la propagation.

Regardons un exemple concret de propagation de transaction en utilisant les exemples que nous présenté au chapitre 6. Le Listing 15-17 montre l'implémentation de l'AuditService bean qui effectue la journalisation d'audit. Nous avons utilisé l'injection de setter dans cet exemple pour le comparer contre la version du chapitre 6.

Annonce 15-17. AuditService Session Bean avec injection de Setter

```
public class AuditService {
    privé EntityManager em;

    @PersistenceContext (unitName = "hr")
    public void setEntityManager (EntityManager em) {
        this.em = em;
    }

    public void logTransaction (int empNo, String action) {
        // vérifier que le numéro d'employé est valide
        if (em.find (Employee.class, empNo) == null) {
            throw new IllegalArgumentException ("Identifiant d'employé inconnu");
        }
```


Épisode 743


```
        LogRecord lr = new LogRecord (empNo, action);
        em.persist (lr);
    }
}
```

De même, le Listing 15-18 montre un fragment du bean de session EmployeeService qui utilise le bean session AuditService pour enregistrer lorsqu'une nouvelle instance Employee a été persisté. Parce que les méthodes createEmployee () et logTransaction () sont invoqués dans la même transaction sans validation entre les deux, la persistance le contexte doit être propagé de l'un à l'autre. Encore une fois, nous avons utilisé l'injection de setter au lieu de l'injection sur le terrain pour rendre le haricot plus facile à tester.

Annonce 15-18. Bean de session EmployeeService avec injection de Setter

```
@Apatride
public class EmployeeService {
    EntityManager em;
    AuditService audit;

    @PersistenceContext
    public void setEntityManager (EntityManager em) {
        this.em = em;
    }

    @EJB
    public void setAuditService (audit AuditService) {
        this.audit = audit;
    }

    public void createEmployee (Employee emp) {
        em.persist (emp);
        audit.logTransaction (emp.getId (), "employé créé");
    }

    // ...
}
```

En utilisant les deux beans session précédents comme exemple, le Listing 15-19 montre comment émuler la propagation entre deux transactionnels gérés par des conteneurs gestionnaires d'entités. La première étape pour rendre ce testable est d'instancier chaque bean session.

Épisode 744

Le bean AuditService est ensuite injecté dans le bean EmployeeService, et le test
L'instance du gestionnaire d'entités est injectée dans les deux beans session. L'injection du même
L'instance EntityManager propage efficacement toutes les modifications à partir de EmployeeService
bean au bean AuditService. Notez que nous avons également utilisé le gestionnaire d'entités dans le
test pour localiser et vérifier les résultats de la méthode commerciale.

Annonce 15-19. Simulation de la propagation de transactions gérées par des conteneurs

```
public class TestEmployeeService {
    // ...

    @Tester
    public void testCreateEmployee () lève une exception {
        EmployeeService empService = new EmployeeService ();
        AuditService auditService = nouveau AuditService ();
        empService.setEntityManager (em);
        empService.setAuditService (auditService);
        auditService.setEntityManager (em);
        Employé emp = nouvel employé ();
        emp.setId (99);
        emp.setName ("Wayne");
        empService.createEmployee (emp);
        emp = em.find (Employee.class, 99);
        Assert.assertNotNull (emp);
        Assert.assertEquals (99, emp.getId ());
        Assert.assertEquals ("Wayne", emp.getName ());
    }

    // ...
}
```

Autres services

Il y a plus dans la plupart des beans qu'une simple injection et transaction de dépendances
la gestion. Par exemple, comme nous l'avons vu au chapitre3 , les haricots peuvent également profiter de
méthodes de cycle de vie. Les autres services qui sortent du cadre de ce livre incluent la sécurité
gestion et intercepteurs.

Épisode 745

La règle générale est que dans un environnement de test, vous devez effectuer manuellement
travail qui autrement aurait été effectué automatiquement par le conteneur. Dans le cas
des méthodes de cycle de vie, par exemple, vous devez invoquer explicitement ces méthodes si elles sont
requis pour un test particulier. Compte tenu de cette exigence, il est judicieux d'utiliser package ou
méthodes de portée protégée afin qu'elles puissent être appelées manuellement par des cas de test.

Cela étant dit, soyez agressif pour déterminer le nombre réel de choses à
pour qu'un test réussisse. Juste parce que les rôles de sécurité ont été déclarés pour un
La méthode du bean session ne signifie pas qu'elle a réellement un effet sur le résultat du test.
S'il n'est pas nécessaire de l'invoquer avant le test, ne perdez pas de temps à configurer le test
l'environnement pour y arriver.

Utilisation d'un conteneur EJB intégré pour les tests d'intégration

Lorsque plusieurs beans session collaborent pour implémenter une utilisation d'application particulière cas, beaucoup de code d'échafaudage peut être nécessaire pour que les choses soient opérationnelles. Si plusieurs les cas de test partagent des graphiques similaires de beans session, une partie ou la totalité de ce code peut devoir être dupliqué sur plusieurs cas de test. Idéalement, nous voulons un cadre pour aider à résoudre les problèmes comme l'injection de dépendances dans notre environnement de test.

Heureusement, EJB prend en charge un tel conteneur. Un conteneur EJB intégré prend en charge EJB Lite, un sous-ensemble de l'ensemble des fonctionnalités EJB. EJB Lite inclut le support pour les beans session locaux, les intercepteurs, les transactions gérées par conteneur (en supposant que disponibilité d'une implémentation de gestionnaire de transactions JTA autonome) et de la sécurité, et JPA, mais n'inclut pas la prise en charge des beans session distants, des beans gérés par message, des points de terminaison de service, des minuteurs ou des beans de session asynchrones. Il offre plus qu'assez pour gérer les différents scénarios de dépendance que nous avons décrits jusqu'à présent.

Pour montrer comment utiliser un conteneur EJB intégré pour les tests d'intégration avec les beans session et le gestionnaire d'entités, nous revisitons le cas de test de propagation de la section précédente «Gestionnaires d'entités gérées par des conteneurs» et convertissez-la pour utiliser un conteneur intégré.

Les conteneurs eJB embarqués Tip ont été introduits dans eJB 3.1.

Épisode 746

Test du chapitre 15

Contrairement aux autres formes de techniques d'intégration que nous avons examinées jusqu'à présent, une le conteneur EJB intégré ne nécessite aucune simulation d'interfaces standard ni de sous-classification de beans pour remplacer le comportement spécifique au serveur. Tant qu'une application s'inscrit dans le sous-ensemble de la spécification EJB pris en charge par les conteneurs intégrés, il peut être utilisé tel quel.

L'amorçage du conteneur intégré est simple. Vous compilez et empaquetez les classes comme d'habitude dans un fichier JAR EJB et ajoutez ce fichier JAR au chemin de classe de test afin que le mécanisme d'amorçage du conteneur intégré puisse le localiser. Le statique La méthode createEJBContainer () de la classe javax.ejb.embeddable.EJBContainer peut puis être utilisé pour créer un conteneur EJB et charger le module à partir du classpath. Référencement 15-20 montre le processus d'amorçage. Options supplémentaires pour le container peut être spécifié en passant une Map of properties, mais pour les tests basiques avec un module unique, aucune configuration particulière n'est requise.

Annonce 15-20. Amorçage d'un conteneur EJB intégré dans un scénario de test

```
public class TestEmployeeService {
    conteneur EJBContainer privé;

    @Avant
    public void setUp () {
        conteneur = EJBContainer.createEJBContainer ();
    }

    @Après
    public void tearDown () {
        container.close ();
    }

    // ...
}
```

Une fois le conteneur initialisé, nous devons accéder aux beans session afin de testez-les. Le conteneur intégré expose son répertoire JNDI interne des beans session via la méthode getContext () de la classe EJBContainer. Le code de test doit alors faire une recherche JNDI globale afin d'accéder à un bean session particulier. Une recherche globale fait pas besoin de références ou d'un contexte de dénomination d'environnement. Au lieu de cela, le nom du module (dérivé du nom JAR) et les noms de session bean sont composés pour former un nom sous la racine JNDI «global». Référencement15-21 démontre cette technique en supposant les beans ont été empaquetés dans un fichier jar EJB appelé hr.jar.

Épisode 747

Annonce 15-21. Acquisition d'une référence de bean de session à partir d'un EJB intégré Récipient

```
La classe publique TestEmployeeService étend TestCase {
    conteneur EJBContainer privé;

    // ...

    private EmployeeService getServiceBean () lève une exception {
        return (EmployeeService) container.getContext ().
        lookup ("java: global / hr / EmployeeService");
    }

    private EntityManager getEntityManager () lève une exception {
        return (EntityManager) container.getContext (). lookup ("java: global /
        hr / HRService ");
    }

    // ...
}
```

Avec l'accès à un bean session en direct, nous pouvons maintenant écrire une méthode de test comme si nous étions exécuter du code directement dans le serveur d'applications. Le listing 15-22 complète cet exemple avec une nouvelle version de testCreateEmployee () qui utilise la référence de bean du conteneur intégré.

Annonce 15-22. Test d'un Session Bean acquis à partir d'un EJB intégré Récipient

```
public class TestEmployeeService {
    // ...

    @Tester
    public void testCreateEmployee () lève une exception {
        EmployeeService bean = getServiceBean ();
        Employé emp = nouvel employé ();
        emp.setId (99);
        emp.setName ("Wayne");
        bean.createEmployee (emp);
```


Épisode 748

```
        EntityManager em = getEntityManager ();
        emp = em.find (Employee.class, 99);
        Assert.assertNotNull (emp);
        Assert.assertEquals (99, emp.getId ());
        Assert.assertEquals ("Wayne", emp.getName ());
    }

    // ...
}
```

Comme indiqué précédemment dans la section «Changement de configuration pour le test», un Un fichier persistence.xml approprié à l'environnement de test peut être requis et doit être empaqueté dans le fichier jar EJB utilisé sur le chemin de classe système. De même, partager l'EJB conteneur à travers les exécutions de tests améliorera probablement les performances globales de la suite de tests encore une fois, il faut veiller à ne pas influencer accidentellement le résultat d'autres tests avec état maintenu par le conteneur EJB.

Pour deux beans session, cette approche est sans doute excessive par rapport au même test cas montré dans l'extrait 15-19 . Mais cela devrait être facile à voir même à partir de ce petit exemple comment des relations bean complexes peuvent être réalisées à l'aide d'un conteneur EJB intégré.

Cadres de test

Tant que les technologies continueront d'être créées et d'évoluer, les frameworks de test seront juste derrière, en essayant de tester davantage le code qui utilise ces technologies maniable. Depuis la sortie de JPA, quelques frameworks ont fait surface avec des niveaux variables du support JPA pour permettre les tests d'intégration. Alors que les détails de l'utilisation de ces cadres dépasse le cadre de ce livre, nous avons pensé qu'il serait peut-être utile de mentionner au moins et leur fournir un pointeur. Vous pouvez ensuite partir explorer par vous-même pour voir si l'un d'entre eux répondra à vos besoins. Si vous entendez parler d'autres personnes bénéficiant d'un soutien spécifique pour JPA mais qui ne sont pas inclus dans cette section, veuillez nous le faire savoir, et nous pouvons inclure les dans les éditions ultérieures du livre.

Unité JPA

Le cadre de l'unité JPA (https://code.google.com/p/jpa-unit) est un simple petit test framework qui prend principalement en charge le test des applications JPA non-conteneur. Ça ne prend en charge les transactions JTA, donc c'est un peu limité. Il est intégré à JUnit et teste

Épisode 749

sont ajoutés en sous-classant une classe de cas de test du framework JPA-Unit. Il fournit l'injection d'un gestionnaire d'entités et prise en charge du préchargement automatique des données d'entité sur la base d'un fichier de données XML formaté. Cela peut être suffisant pour une application avec des exigences, mais n'en offrira probablement pas suffisamment pour la majorité des applications d'entreprise. Il ne semble pas non plus être entretenu très régulièrement et n'a pas eu de mises à jour ou action sur son site Web pendant plus d'un an et demi au moment de la rédaction de cet article.

Test du printempsContexte

Le framework TestContext dans Spring n'est pas un framework JPA uniquement, mais il le fait inclure le support Spring pour l'injection d'objets JPA comme le gestionnaire d'entités et l'entité manager factory (en utilisant le standard @PersistenceContext et @PersistenceUnit annotations). Il peut s'intégrer avec JUnit, TestNG, EasyMock et d'autres tests unitaires frameworks quand vient le temps de la définition et de l'exécution des tests. Parce que tant du cadre est basé sur le test des composants Spring, cependant, adoptant son Le modèle de test n'aura pas autant de sens si votre application n'est pas basée sur Spring. Pour

applications qui utilisent Spring Data JPA, ce cadre pourrait valoir la peine d'être étudié.

Arquillian

Pour tester les applications Java EE qui utilisent des beans CDI ou des EJB pour manipuler des entités JPA, le framework Arquillian (http://arquillian.org) pourrait être votre meilleur pari pour un cadre de test d'intégration.

En général, Arquillian peut tester les éléments suivants:

- Sécurité
- CDI / EJB3
- JPA
- JAX-RS / JAX-WS / WebSockets
- JSF / JSP / Servlets

Le cadre de test Arquillian est composé de trois parties:

- Testeurs tels que JUnit ou TestNG
- Conteneurs
- Test enrichisseurs

Voici les extensions Arquillian les plus courantes et les plus utilisées:

- Persistance
- Drone
- Graphène
- Warp
- Performance

Arquillian prend en charge l'exécution de tests dans un environnement non serveur (en utilisant conteneurs CDI et EJB autonomes) ainsi que l'exécution de tests dans le conteneur. Il s'intègre avec JUnit et TestNG, et s'intègre raisonnablement bien avec ces modèles de test. Parce qu'il regroupe les classes de test et les classes d'infrastructure nécessaires dans une archive et même la déploie sur le serveur, il existe une méthode supplémentaire qui doit être fourni pour définir quels bits et pièces doivent être inclus dans cette application archiver. Le reste est assez facile à faire une fois l'environnement correctement configuré.

Arquillian a cependant quelques dépendances qui devraient être connues à l'avance. Le premier est sur ShrinkWrap, un outil JBoss qui regroupe les artefacts dans une archive. le la seconde concerne l'existence d'un adaptateur de conteneur pour le conteneur sur lequel le déploiement est effectué. Tous les conteneurs ne sont pas pris en charge, donc si vous déployez sur un serveur cible non pris en charge, vous pourriez ne pas avoir de chance. Consultez le site Web pour voir si votre conteneur est pris en charge.

Les meilleures pratiques

Une discussion complète sur les stratégies de test des développeurs dépasse le cadre de ce chapitre, mais pour faciliter le test du code d'application qui utilise des entités, envisagez d'adopter le bonnes pratiques suivantes:

- Évitez d'utiliser le gestionnaire d'entités à partir des classes d'entités. Cette crée un couplage étroit entre l'objet de domaine et le API de persistance, ce qui rend les tests difficiles. Requêtes liées à une entité, mais ne faisant pas partie de son mapping objet-relationnel, sont mieux

exécuté dans une façade de session ou un objet d'accès aux données.

• Utilisez l'injection de dépendances sur les recherches JNDI pour référencer des beans.
L'injection de dépendances est une technologie clé pour simplifier les tests.
Au lieu de se moquer des interfaces JNDI pour fournir un support d'exécution

Épisode 751

pour les tests unitaires, les valeurs requises peuvent être directement affectées au objet en utilisant une méthode setter ou un accès au champ. Notez que l'accès les champs privés d'un cas de test sont de mauvaise forme. Soit utiliser le package privé champs comme cible pour les objets injectés ou fournissent une méthode de définition.

• Isolez les opérations de persistance. Conserver EntityManager et Query les opérations séparées dans leurs propres méthodes permettent de les remplacer plus facile lors des tests unitaires.

• Refactoriser si nécessaire. N'ayez pas peur de refactoriser le code de l'application pour le rendre plus convivial pour les tests tant que la refactorisation profite au application dans son ensemble. Extraction de méthode, introduction de paramètres, et d'autres techniques de refactoring peuvent aider à décomposer la logique de l'application en morceaux testables, améliorant la lisibilité et maintenabilité de l'application dans le processus.

• Utilisez un cadre existant lorsque cela est possible. Si un framework déjà fait la plupart de ce que vous devez faire, vous pouvez clairement gagner du temps et ressources en l'utilisant et éventuellement en ajoutant simplement les bits supplémentaires dont vous avez besoin. Il peut même être approprié de proposer ces changements retour aux committers du cadre pour inclusion dans un prochain version du cadre.

Ces approches peuvent vous aider à soutenir votre style de test, quelle que soit l'approche que vous choisissez d'utiliser.

Notez qu'il y a beaucoup de jolis exemples Java ee 8 dans github à des fins de test:

https://github.com/javaee-samples/javaee8-samples

Résumé

Dans ce chapitre, nous avons commencé par une exploration des tests d'applications d'entreprise et défis auxquels sont traditionnellement confrontés les développeurs. Nous avons également examiné les différents types des tests effectués par les développeurs, les ingénieurs qualité et les clients; et nous avons raffiné notre objectif est d'examiner spécifiquement les tests des développeurs pour les applications JPA.

Épisode 752

Dans la section sur les tests unitaires, nous avons examiné comment tester les classes d'entités, puis nous avons extrait retour pour voir comment tester des beans en combinaison avec des entités dans un environnement de test unitaire. Nous avons introduit le concept d'objets fictifs et exploré comment tester le code qui dépend

sur le gestionnaire d'entités sans utiliser réellement un véritable gestionnaire d'entités.

Dans notre discussion sur les tests d'intégration, nous avons expliqué comment obtenir le gestionnaire d'entité opérationnel dans les tests JUnit dans l'environnement Java SE et les situations dans lesquelles il est logique d'utiliser cette technique. Nous avons couvert un certain nombre de problèmes liés à l'entité manager, y compris comment amorcer en toute sécurité une base de données pour les tests, comment utiliser plusieurs mappage des fichiers pour différentes configurations de base de données et comment réduire le nombre de connexions de base de données requises pour une suite de tests. Nous avons également examiné comment utiliser un conteneur EJB intégré pour les tests d'intégration.

Nous avons examiné comment utiliser les beans session dans les tests d'intégration et comment gérer avec des problèmes d'injection de dépendances et de gestion des transactions. Pour transaction gestion, nous avons examiné comment émuler la gestion des conteneurs et des bean transactions, ainsi que comment simuler la propagation du contexte de persistance dans un test environnement. Nous avons conclu par une brève enquête sur les cadres de test JPA et un résumé des bonnes pratiques à prendre en compte lors de la création d'applications Java EE à l'aide de JPA.

Épisode 753

Indice

UNE

abandonAllChanges (), 266
Fonction ABS JP QL, 387
Requête abstraite, 375
Annotation @Access, 106
accumEmployeeVacation (), 576– 577
Transaction ACID, 86
Processus d'activation, 63
addEmployee (), 195
addItems (), 217
addKeySubgraph (), 519
addNamedEntityGraph (), 521
héritage (*voir* Héritage)
plusieurs tables
type intégré, 458
variété Meet-in-the-Middle, 456
plusieurs tables secondaires, 459, 461
@PrimaryKeyJoinColumn
annotation, 457
Annotation @SecondaryTable, 456
@Table annotation, 456
structure de table, 459
entité deux tables, 457
relations (*voir* Avancé
des relations)

addNamedQuery (), 282
ORM avancé
éléments supplémentaire
facultatif, 445– 446
mappages en lecture , 14–
clés primaires composées (*voir* Clés primaires composées)
conversion de l'état de l'entité (*voir* Conversion état de l'entité)
identificateurs dérivés (*voir* Derived identifiants)
objets incorporés
bidirectionnel
des relations, 425
Informations de conta 26
Classe de téléphone, 4
remplacement de relation, 427– 4
noms de table et de colonne, 416– 417
Relations avancées
joindre des colonnes
Tableau EMPLOYÉ, 449
avec table de jonction, 451
plusieurs-à-un / un-à-un des relations, 450
auto-référencement, 449– 4
joindre des tables
mappage plusieurs-à-un, 447
relation un-à-plusieurs, 448
unidirectionnel / bidirectionnel, 447
état des relations de mappage
classe d'association, 454
entité intermédiaire, 455– 456
avec table de jonction, 454
élément orphanRemoval, 451– 453

M. Keith et al., *Pro JPA 2 dans Java EE 8* , https://doi.org/10.1007/978-1-4842-3420-4

Épisode 754

INDICE

Requêtes agrégé
AVG, 355– 3
COUNT, 35
Entité départementale, 355
Clause GROUP BY, 354, 3
Clause HAVING, 356, 358
MAX, 357
MIN, 357
SELECT, FROM et WHERE clauses, 355
SOMME, 357
syntaxe de, 354
alias(), 381
ALL JP QL expression, 386
ET Opérateur JP QL, 385
TOUTE expression JP QL, 386
Modèles de composants d'application, 54
Entité gérée par l'application
directeur, 198
Persistance gérée par l'application
contextes, 211
archivesConversations (), 299
Cadre arquillien, 737
assignEmployeeToProject (), 194
Annotation @AssociationOverride, 640
Annotation @AttributeOverride, 638
AuditService bean, 79
Fonction AVG, 356
Fonction d'agrégation AVG JP QL, 387

B

Formulaire Backus-Naur (BNF
@Annotation de base, 111, 62
Les transactic s par Be), 87, 92,
descripteur b 81
BETWEEN JP QL opérateur, 385
Relation bidire e un-à-un, 138–
Construction et ment
chemin de classe de déploiement,
options d'emballage
JAR EJB, 671
archive de persistance
archives Web, 673– 6
portée de l'unité de persist 5
Mettre à jour et supprimer les requêtes en masse, 403

C

Annotation @Cacheable, 668
Option CacheStoreMode.USE, 587
Mise en cache
interface de cache, 583
propriétés du mode cache, 587
JPA, 580
couches, 580
cache partagé, 582
cache dynamique
la gestion, 586
configuration statique, 585
Méthodes de rappel, 527
définition, 529
contextes d'entreprise, 531
écouteurs d'entité (*voir* écouteurs d'entité)
méthode cancel (), 218
Métamodèle canonique, 409
Expressions de cas, 396
Expression CASE JP QL, 386
CDI et injection contextuelle
haricots, 78
injection et résolution, 79
méthodes de production
gestionnaires d'entité
les références, 84-

Épisode 755

code producteur et qualificatif
définitions d'annotations, 83–
@ Annotation sécurisée, 82- 8
injection qualifiée, 82
portées et contextes, 80
Les haricots CDI, 78
ClassLoader.getResource (), 662
Gestion des frais de netto
clear () méthode, 234, 28
Expression COALESCE,
Expression COALESCE JP QL, 386
méthode coalesce (), 398
Cartographie de la collection
Table de jointure DEPT_EMP, 173
doublons, 185
type intégrable, 174
Entité EMPLOYÉ et TEMENT
les tables, 170,
Table de collecte EM 70
attribut d'entité, 173– 174
EntityType, 179
types énumérés, 171
clés et valeurs, 168
liste
ordre par entité / élé
attribut, 163– 164
listes ordonnées, la ce 164- 1
PrintQueue to PrintJob, 166
relation plusieurs à plusieurs, 172
valeurs nulles, 187
mappage un-à-plusieurs, 172
relations et collections d'éléments
les incorporables et les types de base, 158
Table d'entité EMPLOYEE, 160
Remplacer la table de collecte
colonnes, 160
VacationEntry intégrable
classe, 158
règles, 182, 1
ensembles, 1
Clés à cordes, 169
Association valorisée par la collection
tables de jointure, 145
mappages plusieurs à plusieurs, 143
mappages un-à-plusieurs, 140
collection unidirectionnelle
mappages, 147
@ Annotation de colonne, 111
Élément columnDefinition, 69
Mappages de colonnes, 111– 1
Annotation @ColumnResult,
commit () méthode, 251
Identificateur de composé, 439, 4
Clés primaires composées
classe d'identifiant intégrée, 43
Entité SALARI
Classe d'identité 2
Fonction CONCAT 87
Concurrence
accès entité, 555
opérations de l'entité, 555
Annotation de contrainte, 548
Classe d'implémentation de contrainte, 550
Injection constructeur, 79
Géré par conteneur, 192
Gestionnaires d'entités gérées par conteneurs, 729
Transactions eneur (CMT),
87,
EJB, 90–
intercepte saction
CDI Bean, 91- 92
attributs de transaction,
Contextes et injection de dépendances
(CDI), 51
Annotation @Convert, 421
Annotation @Converter, 423

Épisode 756

Conversion de l'état de l'entité
conversion d'attribut
collections d'éléments, 421– 42
attributs intégrés, 420– 421
limites, 422
conversion automatique, 423– 424
création de convertisseur
critères de jointure externe, 401
expressions de paramètres, 390
prédicats, 389
sous-requêtes, 390
mise à jour et suppression groupées, 404
métamodèle canonique
outils de génération, 411– 4

Interfac teConverter, 418–
Convert oléen en entier, 419
convertisseurs et requêtes, 424
COUNT DISTINCT JP QL agrégat une fonction, 387
Fonction COUNT, 357
COUNT fonction d'agrégation JP QL, 387
createCriteriaUpdate (), 403
createEmployee (), 204
createEntityGraph (), 521
createEntityMa ctory (), 36, 553, 679–
createNamedQ 79
createNativeQuery (), 4
createQuery (), 43, 275
API de critères
construction d'expressions
expressions de cas, 396
abattu, 400
dans les expressions, 394
expressions de fonction, 399
Agrégat JP QL vers CriteriaBui
cartographie des fonctions, 3
Fonction JP QL to C il
cartographie, 385,
JP QL vers le prédic aBuilder
cartographie, 385
JP QL vers CriteriaBuilder scalair
mappage d'expression, 385– 386
littéraux, 389
classe de métamodèle pour , 409
Annotation @StaticMetamodel, 410
approche fortement typée, 410
Interface CriteriaBuilder, 374
Objets CriteriaQuery et Subquery, 374
requêtes dynamiques, 366
Clause FROM
récupérer les jointures, 384
jointures intérieures et extérieures, 383
Clauses GROUP BY et HAVING, 403
API métamodèle, 405
objets et mutabilité, 374
Clause ORDER BY, 402
types paramétrés, 366
expressions de chemin, 377
définition de requête, création, 372
racines de requête, 375
Clause SELECT
sélection multiple expressions, 380
sélectionner des expressions uniques, 378
en utilisant des alias, 381
sélectionner les méthodes de clause de requête, 372
API basée sur des chaînes, 408
syntaxe et usage, 364
Clause WHERE, 384
Interface CriteriaBuilder, 364, 37
Objet CriteriaQuery, 364, 373
Objet CriteriaUpdate, 403
Fonction CURRENT_DATE JP QL, 387

Épisode 757

Fonction CURRENT_TIME JP QL, 387
CURRENT_TIMESTAMP JP Fonction QL, 387

ré

Objet d'accès aux données (DAO), 3
Langage de définition de données (DDL), 682
Mappeurs de données, 12
Interface DataSource, 86
Services de conteneurs d , 55
Supprimer les requêtes, 3
Élément identificateurs d 603
Recherche de dépendance
Recherche EJBContext (),
Dépendance EJB, 70- 71
Gestion des dépendances et Cl
déclaration de dépendances
référencement de contexte de persistance, 76
référencement d'unité de persistance, 76
référencement des ressources serveur, 77
injection de dépendance
injection de champ, 74
injection de setter, 74
attribut de rel 8
règles, 435–
mappages pa 37
identifiant entier simple, 437
attribut unique, 436
cartographie triviale, 439
Identifiants dérivés, alternative, 442
Entité détachée
opération detach (), 239
configuration de chargement avide, 250
EmployeeService bean, 246
gestionnaire d'entités par demande, 256
Page JSP, 248
attributs de chargement différé, 23
listEmployees.jsp, 247
stratégie de fusion
début de la session, 261
achèvement de la session, 262
modèle de conversation, 266
HttpSessionBindingListener rappel, 263
save () méthode, 260
façade de session, 259
bean session avec état, 260

recherche de dépendances, 72
annotations de référenc ource, 69
Entité dépendante, 81, 434
Identifiants dérivés
alternative, 443
identifiant composé, 439
entité dépendante, 434– 435
attribut dept, 441
classe d'identifiant intégrable
Classe DeptId, 442
ProjectId, classe 441
Annotation @JoinColu
entité mère, 434– 435
Classes ProjectId et De 40
cadres d'applications Web, 263
niveau Web, 263
fusionner
objet addr, 243
réglage en cascade, 245
état d'entité antérieur, 243–
Exception d'argument illég
exception, 242
instance gérée, 242
contexte de persistance, 241
méthode persist (), 242
téléphone / entité départementale, 243
Architecture MVC, 246
attribut parkingSpace, 239

Épisode 758

Entité détachée (*suite*)
contexte de persistance, 238
relation téléphonique, 240
vue des transactions, 251– 2
déclenchement du chargem seux, 249
opération detach (), 239
opérateur point (.), 321
Downcasting, 399

E

Classe contactInfo intégrable, 426
@ Annotation intégrée, 637
Annotation @EmbeddedId, 622
Mappages intégrés, 637– 638
emp-classes.jar, 673
Classe EmployeeDebugListener, 535, 540
EmployeeDebugListener.prePersist ()
méthode, 541
Table d'entité EMPLOYEE, 170
Classe EmployeeId Id, 431
Classe EmployeeService en J EE, 46- 47
unité de persis 98–
servlet, 97– 9
haricot de ses 6
emp-persistence.jar, 675
Archive des applications d'entreprise (EAR), 669
Test d'applications d'entreprise
tests d'acceptation, 702
test fonctionnel, 701
test d'intégration, 701
Clients Java SE, 700
Framework de test JUnit, 703
composants extérieurs, 702– 703
déploiement de serveur, 700
environnement serveur, 699
développement piloté par les tests, 700
test unitaire (*voir* Test unitaire)
Entreprise JavaBean (EJB)
géré par conteneur
transactions, 90
JAR, 671
modèle, 54
Requête Enterprise JavaBeans
langue (EJB QL), 270
Entité
création, 31- 33
granularité, 27
identité, 26
persistance, 26
transactionalité, 27
Graphiques d'entités
accès aux graphiques d'entités nommées, 520
addNamedEntityGraph ()
méthode, 521
annotation
graphes d'attributs, 508– 5
définition, 514
héritage, 513– 514
mapper les sous-grap é 16
@NamedEntityGrap
sous-graphiques, 50
API
addAttributeNodes (), 517
addSubgraph (), 517
création, 516
héritage, 518
carte sous-graphe clé, 519
classe d'entité racine, 518
createEntityGraph (), 521– 522
graphique de récupération par d 7
sous-ensemble de mod e, 506
chercher des graphiq 25
graphique de charge
état, 506
types, 505– 5

Épisode 759

Écouteurs d'entité, 531
  Haricots CDI, 532
  défaut, 535
  élément, 606
  écouteurs d'entités multiples, 533
Gestionnaire d'entité
  entité gérée par l'application
    directeur, 201
  gestionnaire d'entités géré par conteneur
    gestionnaire d'entité étendue, 198
    portée par transaction, 194
  createEntityManagerFactory (), 36
  createQuery (), 43
  entité détachée (*voir* entité détachée)
  requête dynamique, 42
  EmployéService, 46
  EntityManagerFactory, 36
  méthode find (), 38
  propagation flexible des transactions, 224
  javax.persistence.EntityManager, 34
  javax.persistence.
    EntityManagerFactory, 34
  cycle de vie de, 224
  requête nommée, 42
  objets et concepts, 36
  les opérations
    cascade persister, 231
    supprimer en cascade (), 233
    clear () méthode, 234
    entités employés / adresses, 230
    méthode find (), 228
    annotations de relations logi
    méthode persist (), 225– 22
    remove () méthode, 229
  contexte de persistance, 34, 3
  unités de persi , 35
  requêtes, 42, 4
  remove () métl 40
  taille et complexité, 224
  synchronisation
    paramètres de cascade, 236
    composants, 235
    objet emp, 236
    opération flush (), 235
    Exception IllegalStateException, 237
    objets de téléphone, 237
    objet ps, 237
    opération remove (), 237
    entité non gérée, 236
  gestion des transactions (*voir*
    Gestion des transactions)
  transactions, 41
  mise à jour, 40
EntityManagerFactory (), 76, 553
EntityManagerFactory ajouter
    Requête nommée (), 280
EntityManager.find (), 563
Instance EntityManager, 75
EntityManager.lock (), 563
EntityManager.persist (), 528
EntityManager.refresh (), 563
EntityManager.remove (), 528
Métadonnées d'entité
  annotations, 28- 29
  configuration, 30
  XML, 30
Entité en lecture seule, 444
Annotation @EntityResult, 491, 498
État de l'entité
  accès terrain, 104
  accès mixte, 106– 107
  accès à la propriété, 10
Interface EntityTransaction, 219
@Annotation énumérée, 423, 625
Types énumérés, 115
@Even annotation de contrainte, 549

Épisode 760

exclure-un-listed-classes, 665, 680
executeUpdate (), 285
EXISTS Opérateur JP QL, 386
Gestionnaire d'entités étendu, 197
Contextes de persistance étendus
  addEmployee (), 206
  EmployéService, 206
  changements de service de journalisation, 206
  logTransaction (), 207
  collision de contexte de persistance, 207
  héritage du contexte de persistance, 210
Méthode getEntityGraph (), 520
getEntityManagerFactory ()
    appel, 198
getIdentifier (), 590
méthode get (), 378
getOutputParameterValue (), 501
getPersistenceUnitUtil (), 590
getPersistenceUtil (), 589
getProperties (), 200
getResultList (), 285, 320
getResultStream (), 42

UserTransaction.begin (), 205

F

méthode fetch (), 384
Injection de champ, 73
Annotation @FieldResult, 4
méthode findAll (), 246, 248
méthode find (), 205
Types de colonnes à virgule flottante, 695
opération flush (), 572
Contrainte de clé étrangère, 693– 6
Clause FROM
  définition, 319
  récupérer les jointures, 384
  variables d'identification, 325
  jointures intérieure et extérieure, 382
  jointures (*voir* Jointures)
Tests fonctionnels, 701
Expressions de fonction, 399
méthode function (), 399

g

Annotation @GeneratedValue, 121
Processeur de génération, 683
appel getDepartment (), 249

748

Clause GROUP BY, 402

H

Clause HAVING, 357– 35

je

Annotation @Id, 104– 105
Annotation @IdClass, 623
Génération d'identifiant
  identifiant automatique
    génération, 122- 123
  identité de la base de donn
  séquence de base de données, 126
  @GeneratedValue
    annotation, 121
  stratégies, 121
  table
    Entités d'adresse, 125
    Élément allocationSize, 125
    colonnes, 123
    stockage des identifiants, 123
    Élément initialValue, 125
    élément de nom, 124
    SQL, 126
    élément de table, 124

Épisode 761

Clause INCREMENT BY, 127
Héritage
  hiérarchies de classes
    classes abstraites et concrètes, 466
    modèle objet java, 461
    superclasse mappée, 463– 464
    classes non-entité, 462
    classes transitoires, 465
  l'héritage mixte, 477, 479
  modèles (*voir* Modèles d'héri
Modèles d'héritage
  stratégie commune, 471–
  stratégie à table unique
    Contrat Entité employée , 470
    échantillon de données, 471
    colonne discriminante, 468
    valeur discriminante, 469
    Classe d'entité EMPLOYEE, 467
    classes de persistance, 467
    requêtes polymorphes, 467
  stratégie tab oncrète, 474, 476, 4
Les relations d'h
méthode initialize (), 550
méthode init (), 194
Opérateur IN JP QL, 386
méthode in (), 394
Test d'intégration, 701, 71
  classe de persistance, 716
  exécution de requêtes, 716
  configurations de commutation, 718
  TearDown (), 716
  configuration de test et démo
  UserService bean, 714– 715
  vs . Test de l'unité, 716
cadres
  Cadre arquillien, 737– 738
  Unité JPA, 736
  Spring TestContext, 737
IS EMPTY JP QL opérateur, 386
isLoaded (), 590
isLoaded (Objet), 589
N'EST PAS VIDE JP QL, 386
N'EST PAS NULL JP QL, 386
EST NULL JP QL, 385

J, K

élément jar-file, 666
Modèle de composants d'application Java, 55- 56
JavaBean, 54
Architecture du connecteur Java (JCA), 87
Connectivité de base de données Java (JDBC), 3
Objets de données Java (JDO), 15
Kit de développement Java (JDK), 13, 1

composants et persistance
entité gérée par conteneur
cadres, 730
conteneur EJB embarqué, 733
méthodes du cycle de vie, 733
gestion des transactions (*voir* Gestion des transactions)
gestionnaire d'entité
connexion à la base de données
minimisation, 721
Java EE 8
Composants, 52- 53
pile, 53
Les ressources par défaut de Java EE,
Java 2 Enterprise Edition (J2EE), 13
Interface de nommage
aire Java (JNDI), 70, 65
API de persistance Java
configuration, 23
EJB 3.0 / JPA 1.0, 17- 1

Épisode 762

API de persistance Java (JPA) (*suite*)
entité
création et transaction
la gestion, 34, 47
Opération CRUD, 45
granularité, 27
identité, 26
mise en œuvre, 45
manager (*voir* Entity manager)
métadonnées, 30
persistance, 26
archive de persistance, 48
unité de persistance, 48
la transaction, 27
histoire de, 17
intégration / testabilité, 23
Persistance du support Java
mappeurs nnées, 12
EJB, 13- 1
objets de es java (JDO), 15
JDBC, 13
propriétaire
solutions, 11
JPA 2.0, 19
JPA 2.1, 19
JPA 2.2 et EJB 3.2, 19
entité mobile, 22
non intrusivité, 21
requête d'objet, 22
ORM
avantages de, 4
représentation de classe, 6
définition, 3
héritage, 11
relation, 8
Projet OSS, 16
Persistance POJO, 16, 21
bases de données relationn
Java Persistence Query Languag
requêtes agrégées, 27
application, 316, 318
instruction de mise à pression en bloc
application, 303
enlèvement massif de, 302
entités A / B, 303
opération find (), 302
numéros, 302
séquences d'opérations, 303
maintien de la relation, 305
REQUIRES_NEW transaction, 301
persistance à l'échelle des transactions
contextes, 302
API Criteria, 313– 31
supprimer des requête
la description, 313
modèle de domaine, 315–
requête nommée dynamic
EntityManagerFactory
addNamedQuery (), 281
getEntityManagerFactory (), 281
définition de requête dynamique
méthode createQuery (), 275
Demande GET / POST, 277
attaque malveillante, 277
paramètres nommés, 277
Condition OU, 276
moteur de requête, 276
informations salariales, 275
menace de sécurité, 2
valeur de chaîne, 275
EJB QL, 270
Instance d'employé, 272
jointures d'entités, 272
Annotation @Entity, 315
filtration, 271
requête nommée

Épisode 763

définition, 278
Employee.findAll, 278
Interface EntityManager, 279
exécution, 279
plusieurs entités, 279
Annotations @NamedQuery, 279
paramètres de position, 280
concaténation de chaînes, 278
modèle d'objet, 313
types de paramètres
contraignant, 283
date / calendrier, 283
entité départementale, 283
paramètres de service, 284
définition de requête nommée, 283
méthode setParameter (), 282
Énumération TemporalType, 283
Clause WHERE, 284
performance et réactivité
opérations de mise à jour et de suppression en masse, 309–
requêtes, 308
fournisseur de persistance, 310
signaler des requêtes, 308
bean session sans état, 309
conseils du vendeur, 308
exécution de requête
getResultList (), 285
getSingleResult (), 286
regroupement d'expressions / multiple
les fonctions, 293
itération, 285
NonUniqueResultException
exception, 286
NoResultException
exception, 286
optimisation, 289–290
pagination, 295
délai d'expiration de la requête, 299–300
type de résultat, 288
Requête SELECT, 286
méthode setLockMode (), 286
objet unassignedQuery, 286
modifications non validées, 299
Résultat non typé, 289
conseils de requête, 307
paramètres de requête, 273
Sélectionner une requête (*voir* Sélectionner des requêtes)
type d'entité unique, 27
terminologie, 314–315
mettre à jour les requêtes, 310
Visages JavaServer (JSF), 67
Pages JavaServer (JSP), 67
Bibliothèque de balises standard de JavaServer Pages
(JSTL), 247
Persistance du support Java
mappeurs de données, 12
entreprise JavaBeans, 13-14
Objets de données Java, 15
JDBC, 13
solutions propriétaires, 11
API de transaction Java (JTA), 41, 87
javax.persistence.EntityManager, 34
paquet javax.persistence, 28
javax.persistence.query.timeout
propriété, 299
Annotation @JoinColumn, 414, 44
Stratégie jointe , 471 - 47
joinMap (), 383
join () méthode, 382
Rejoint
conditions, 3
chercher, 334
interne
JOIN opérateur et collection
champs d'association, 327–3

Épisode 764

Rejoint (*suite*)
Opérateur JOIN et valeur unique
champs d'association, 329
jointures multiples, 331
Clause WHERE, 331
carte, 332
extérieur, 333
Annotation @Join, 46
appel joinTransaction (), 211, 213
Architecture JPA ORM, 102
Cadre JPA-Unit, 736
JP QL, *voir* Requête de persistance Java
Langue (JP QL)
Transactions JTA, 202
Cadre de test JUnit, 703
Événements du cycle de vie
PostLoad, 529
PrePersist et PostPersist, 528
PreRemove et PostRemove, 528
PreUpdate et PostUpdate, 529
validation, 554
Gestion du cycle de vie, 54
COMME Opérateur JP QL, 386
Valeurs littérales, 389
loadEmployee (), 266
Annotation @Lob, 114
Fonction LOCATE JP QL, 387
Verrouillage, 559
verrouillage optimiste, 560
modes avancés, 563

L

Récupération paresseuse, 113
Relations paresseuses, 148
Fonction LENGTH JP QL, 387
Rappels de cycle de vie, 527
  méthodes de rappel
    définition, 529
    contextes d'entreprise, 531
    héritage, 536
  écouteurs d'entité, 531
    Les haricots CDI, 532
    par défaut, 535
    héritage, 537
    plusieurs écouteurs d'entités, 533
  événements du cycle de vie
    héritage, 536
    ordre d'invocation, 537
    PostLoad, 529
    PrePersist et PostPersist, 528
    PreRemove et PostRemove, 528
    PreUpdate et PostUpdate, 529
    échecs optimistes, 570
    verrouillage de lecture, 564
    système de versionnage, 561
    verrouillage d'écriture, 567
  verrouillage pessimiste, 574
    modes, 575
    échecs pessimistes, 579
    incrément forcé pessimiste
      verrouillage, 578
    portée pessimiste, 578
    timeouts pessimistes, 578
    verrouillage de lecture, 577
    verrouillage d'écriture, 575
Annotation logique, 103
logTransaction (), 203– 205
Couplage lâche, 54
Fonction LOWER JP QL, 387

M

Classes gérées et cartographie
  élément d'attribut, 617
  mappages d'objets incorporés

Épisode 765

    remplacement d'association, 640
    remplacement d'attribut, 638
    élément intégré, 637
  écouteurs d'entité, 646
    exclude-default-listeners, 648
    exclure-les-auditeurs-de-superclasse, 649
  mappages identifiant, 621
    élément embarqué-id, 622– 623
    élément id, 621–
    id-class, 623– 62
  mappages d'héritage
    remplacement d'association, 644
    remplacement d'attribut, 644
    élément discriminateur-colonne, 642
    élément de valeur discriminante, 643
    élément d'héritage, 641
  événements du cycle de vie, 645
  graphes d'entités nommées, 649
  mappages de relations et de collections
    élément-collection, 635
    mappage plusieurs-à-plusieurs, 634
    mappage plusieurs-à-un, 628
    mappage un-à-plusieurs, 630
    mappage un à un, 632
  mappages simples
    mappages de base, 624– 6
    élément transitoire, 626
    élément de version, 627
  tableaux, 619
Annotation @ManyToMany, 143, 634
Relation plusieurs-à-plusieurs, 144
Annotation @ManyToOne, 134
Fonction d'agrégation MIN JP
Héritage mixte, 477, 479
Stratégies mixtes, 478– 479
Moquerie, 708
Model-View-Controller (MVC), 246
Fonction MOD JP QL,
sélection multiple(), 37

N

Graphiques d'entités nommées, 649
Annotation @NamedNativeQuery, 488
Annotation @NamedQuery, 278
Requête de procédure stockée nommée, 614
NameValidator, classe 535
NameValidator.validateName (), 541, 542
N'EXISTE PAS l'opérateur JP QL, 386
NOT IN Opérateur JP QL, 386
PAS opérateur JP QL, 385
PAS COMME l'opérateur JP QL, 386
PAS MEMBRE DE l'opérateur JP QL, 386
Contraintes nulles, 691
Expression NULLIF JP QL, 386

O

Connectivité de la base de données d'objets (ODBC), 13
Base de données orientée objet (OODB), 15
Cartographie objet-relationnelle (ORM)
  bénéfices de, 4
  représentation de classe, 6

Fonction MAX, 357
Fonction d'agrégation MAX JP QL, 387
MEMBRE DE L'opérateur JP QL, 386
opération merge (), 241
API du métamodèle, 405
Fonction MIN, 357
définition, 3
objet incorporé
Type d'adresse intégré
définition, 151
informations d'adresse, 150
adresse partagée par deux entités, 153

Épisode 766

Cartographie objet-relationnelle (ORM) (*suite*)
Annotation @AttributeOverride, 153
description, 149
dans plusieurs entités, 154
état de l'entité
accès sur le terrain, 104
accès mixte, 106
accès propriété, 105
discordance d'impédance, 4– 5
héritage, 11
mappage de la clé primaire
génération d'identifiant (*voir* Identifiant génération)
remplacer la colonne de clé primaire, 120
types de clé primaire, 121
mappage de types simples
@Annotation de base, 111
mappage de colonnes, 111–
types énumérés, 115
gros objets, 114
paresseux, 113
liste des types persistants,
types temporels, 118– 11
état transitoire, 119– 12
mappage à la table, 108– 10
annotation de persistance, 1
des relations
cardinalité, 131– 13
associations valorisé collection (*voir* Association valorisée par la collection)
directionnalité, 130
relations paresseuses, 148
ordinalité, 132
rôles, 129
associations à valeur unique (*voir* Association à valeur unique)
méthode on (), 401
Annotation @OneToMany, 141, 630
Relation un-à-plusieurs, 142
Annotation @OneToOne, 137
Logiciel Open Source (OSS), 16
OptimisticLock on, 560-561 ,
570–
Verrouillage op
modes avancés, 563
échecs optimistes, 570
lecture de verrouillage, 564
système de versionnage, 561
verrouillage en écriture, 567
Verrouillage de lecture optimiste, 564
Verrouillage d'écriture optimiste, 567
Mappages facultatifs, 446
Clause ORDER BY, 353– 35
Opérateur OR JP QL, 385
Suppression des orphelins,

P

Contrôleur de page, 246
Expressions de paramètres, 390
Entité mère, 434
PartTimeEmployee.check
Vacances (), 536
Méthode de passivation, 63
opération persist (), 242
Annotation de persistance, 102
Archive de persistance, 674– 6
Annotation @PersistenceConte
Collision de contexte de persistance, 207
Héritage du contexte de p e, 210
Unité de persistance, 47,
@PersistenceUnit, 76, 20
Configuration de l'unité de
élément de classe, 680
persistance de la ligne de commande
propriétés, 681

Épisode 767

source de données, 6
EmployéService, 656
Prédicats, 388
Méthode PrePassivate, 63

classes gérées
classes explicitement répertoriées,
élément jar-file, 666
cours locaux, 664
fichiers de mappage, 664
fichiers de mappage, 662
fournisseur de persistance, 658
élément de propriétés, 668
élément de fournisseur, 679
mode cache partagé, 667
chemin de classe système,
Type de transaction, 657– 6
mode de validation, 668
Champ d' application de l' unité de ce, 676- 677
PersistenceUnitUtil, 590
PersistanceUtil, 589
Incrément forcé pessimiste
verrouillage, 578
Verrouillage pessimiste, 574
modes, 575
échecs pessimistes, 579
incrément forcé pessimiste
verrouillage, 578
portée pessimiste, 578
timeouts pessimistes, 578
lecture de verrouillage, 577
verrouillage en écriture, 575
Verrouillage de lecture pessimiste, 577
Verrouillage d'écriture pessimiste, 575
Annotation physique, 103
Ancien objet Java brut (POJO), 16
Méthode PostActivate, 63
@PostConstruct, 281
Rappel PostLoad, 529
Événements postpersistes, 528
Rappel PostUpdate, 529
Événement PrePersist, 528
Événements PreUpdate, 529
@PrimaryKeyJoinColumn, annotation, 457
Types de clé primaire, 121
Entité PrintJob, 165
processAllChanges (), 265– 266
Champs de producteurs, 83
Méthode du producteur, 82
public void foo () {}, 530
public void foo (Object o) {}, 532

Q

Annotation de qualificatif, 81
Requete
graphiques d'entités (*voir* Graphiques d'entités)
SQL (*voir* requêtes SQL)
Annotations @QueryHint, 307
Langage de requête, *voir* Persistance Java
Langage de requête (JP QL)
Query.setHint (), 307
Query.setLockMode (), 563

R

Déclarations de variables de plag
Mappages en lecture seule, 444–
Refresh () méthode, 306
état de l'entité, 555
Bases de données relationn
resetSyncTime (), 535, 541
Annotation @Resource, 77
Annotations de référence sur les ressources, 69
@ Politique de rétention, 549
Propriété de génération de schéma d'exécution
différences, 689

Épisode 768

S

save () méthode, 260
saveChangesToEmployee (), 266
Expressions scalaires
Expressions CASE, 351– 353
la description, 34
fonction, 348– 3
littéraux, 347– 3
fonctions de ba nées natives, 3
Génération de schéma
Élément columnDefinition, 696
définition, 682
propriétés de déploiement, 689
entrée de génération, 687
sortie de génération, 684
colonnes à virgule flottante, 695
contraintes de clé étrangère, 692– 6
processus de génération, 683
héritage et polymorphisme
abattu, 346– 347
QualityProject et Desig ,
345
discrimination de sous-cl
Clause ORDER BY, 353– 35
expressions scalaires (*voir* S
expressions)
Clause SELECT, 321
Instruction SQL, 320
Clause WHERE (*voir* clause WHERE)
Annotation @SequenceGenerator, 610
Élément générateur de séquence, 610
Interface du fournisseur de services (SPI), 658
Servlets
maintien de l'état conversationnel, 68
définition, 67
Session HTTP, 67
Beans de session

élément indexes, 692
annotations de mappage, 689
contraintes nulles, 691
colonnes basées sur des chaînes, 694
contraintes uniques, 690
Annotation @SecondaryTable, 456
méthode selectCase (), 397
Clause SELECT
expressions de constructeur, 325
définition, 319
entités et objets, 322, 324
expressions multiples, 324, 379
expressions de chemin, 321
sélection d'expressions uniques, 377
en utilisant aliase, 381
méthode select (), 377– 37
Sélectionnez les requêtes
Clause FROM, 325
getResultList (), 320
variables d'identification, 320

définition, 56
beans session singleton
définition, 66
les rappels de cycle de vie, 67
beans session avec état
caisse (), 63
définition, 61
les rappels de cycle de vie, 63
@Remove annotation, 62
mise e e du panier,
61–
beans sess ns état
interface métier, 58
définition, 58
Interface HelloService, 59
les rappels de cycle de vie,
sans interface, 59
Instance SessionContext, 77
setFirstResult (), 294

Épisode 769

setFlushMode (), 298
setLockMode (), 286
setMaxResults (), 294
setParameter (), 282, 284
Setter injection, 74
méthode setUp (), 721
Cache partagé
interface de cache, 583
gestion dynamique du cache, 586
EntityManagerFactory.getCache (), 583
mode, 667
configuration statique, 585
Stratégie de table unique, 467– 4
Haricots de session singleton, 57
Association à valeur unique
bidirectionnel un-à-un
mappages, 138
joindre des colonnes, 134
mappage plusieurs à un, 133
mappages un à un, 137
Fonction SIZE JP QL, 387
QUELQUES expressions JP QL, 386
Spring TestContext, 737
Requêtes SQL
avantage, 488
définition, 487
exécution, 488
Instructions INSERT et , 490
Requêtes JDBC, 484– 4
JPA, 489
JP QL, 483 à
liaison de para 500
mappage d'ensemble de résultats
alias de colonne, 492
clés composées, 496– 497
définition, 491
mappage de résultats multi
types non-entité, 499– 500
colonnes de résultats scalai
@SqlResultSetMapping
annotation, 491
procédures stockées
définitions, 501
JPA 2.1, 500
cartographie, 503–
types de paramètr nble de 503
types de paramètres scalaires, 501–
@SqlResultSetMapping, annotation, 491
Fonction SQRT JP QL, 387
Beans session avec état, 57, 61
Beans session sans état,
voir Session beans
API basée sur des chaînes, 407
méthode subquery (), 390
Fonction SUBSTRING JP QL, 387
Fonction SUM, 357
Fonction d'agrégation SUM JP QL, 387

T

@Table annotation, 108
Annotation @TableGenerator, 611
Stratégie table rète, 474,
476,
Clé étrangère c
@Annotation temporelle,
Types temporels, 118– 119
testAuthenticateInvalidUs
testAuthenticateValidUser (), 720
Développement piloté par les tests (TDD), 700
@ Annotation transactionnelle, 91
Intercepteurs transactionnels, 91

clés étrangères, 491
héritage, 498

Association de transaction, 202
Annotation @TransactionAttribute, 90

Épisode 770

La démarcation des transactions, 87-
Gestion des transactions, 722
Transactions ACID, 86
transactions gérées par bean
transaction utilisateur stubbed, 729
exécution de tests, 728
Interface UserTransaction, 726– 727
BMT, 92 ans
transactions gérées par conteneur
prestations, 724
méthode commerciale, 724–
Transactions EJB, 91
intercepteurs transactionnels, 92
attributs de transaction, 89
transactions d'entreprise
CMT, *voir* Géré par conteneur Transactions (CMT)
démarcation des transactions, 88
JTA
persistance gérée par l'application contexte, 214
collision de contexte de persistance, 210
contexte de persistance héritage, 211
persistance à l'échelle des transactions contexte, 205
non synchronisé, 218
transactions locales, 201
Propriétés, 86
transactions locales aux ressources
méthode begin (), 219
Interface EntityTransaction, 219– 220
méthode getRollbackOnly (), 219
méthode getTransaction (), 218
Environnement Java EE, 221
rollback / état de l'entité, 224
session beans et persistance
opérations, 723
Propagation des transactions, 202
Annulation de transaction, 221
Portée par transaction, 192
Gestionnaires d'entités à l'échelle des transactions, 193
Contextes de persistance à l'échelle de la transaction, 202
Synchronisation des transactions, 2[illegible]
@ Annotation transitoire, 107, 119
Superclasse transitoire, 465
Fonction TRIM JP QL, 387

U

Unidirectionnel un-à-plusieurs Relation, 148
Langage de modélisation unifié (UML), 6
Tests unitaires
entités
méthodes commerciales, 704
en composants, 706
validation des données et transformation, 705
méthodes de propriété, 704
méthode setId (), 705
gestionnaire d'entité
méthode authenticate (), 709– 710
Classe MockEntityManager, 712
interfaces standards, 709
Classe UserService, 710
Contextes de persistance non synchronisés, 215
Cartes non typées, 181
Requêtes de mise à jour, 3
Fonction UPPER JP QL, [illegible]
Convertisseur URL-chaîne, 423
Interface UserTransaction, 93
Classes d'utilité, 589
PersistenceUnitUtil, 590
PersistanceUtil, 589

Épisode 771

V

Validation, 535, 542
contraintes, 543
annotations de contrainte, 548
création, 548

X, Y, Z

Fichiers de mappage XML, 633
élément convertisseur, 652
désactivation de l'annotation attribut de métadonnées complètes,

groupes, 546
classes d'implémentation, 549
activation de la validation, 552
entité, 551
groupes, 546
invoquer la validation, 545
javax.persistence.validation.mode, 553
JPA, 551
groupes de validation du cycle de vie, 553
plusieurs groupes, 547
élément validation-mode, 552
Mode de validation, 668
Annotation @Version, 627
Champs de verrouillage de version, 562

W

Archive Web (WAR), 669, 673-
Clause WHERE, 384
TOUTES, TOUTES et CERTAIN[ssions, 344
forme d'expression de base, 337– 3
ENTRE les expressions, 338
expressions de collection, 342– 34
définition, 336
EXISTS expressions, 343
Dans les expressions, 341
Paramètres d'entrée, 336
COMME les express]
sous-requêtes, 339– 3
599– (
xml-ma :tadata-complete
élément, 599
mappages d'entités, 598
générateur et élément de requête, 609
query-native-nommée, 612
requête nommée, 611
requête-procédure-stockée-nommée, 614
générateur de séquence, 610
sql-result-set-mapping, 615
élément générateur de table, 611
En tête de fichier, 597
classes gérées et mappage (*voir* Classes gérées et cartographie)
élément par défaut du fichier de mappage
accès, 609
élément de catalogue, 608
élément de package, 606
élément de schéma, 607
métadonnées, 595
valeurs par défaut de l'unité de persistance
élément, 601
élément d'accès, 603
élément cascade-persist, 604
élément de catalogue, 603
identificateurs-délimités, 603
élément entity-listeners, 605
élément de schéma, 602