Red Hat JBoss Enterprise Application Platform 7.0

開発ガイド

Red Hat JBoss Enterprise Application Platform 7.0 向け

Last Updated: 2018-04-05

Red Hat JBoss Enterprise Application Platform 7.0 開発ガイド

Red Hat JBoss Enterprise Application Platform 7.0 向け

法律上の通知

概要

本書は、Red Hat JBoss Enterprise Application Platform 7.0 とそのパッチリリースを使用する Java EE の開発者向けのリファレンスや例を提供します。

目次

第1章 アプリケーションの開発 13
1.1. はじめに 13
1.1.1. Red Hat JBoss Enterprise Application Platform 7 について 13
1.2. JAVA ENTERPRISE EDITION 7 について 13
1.2.1. EE 7 プロファイルの概要 13
Java Enterprise Edition 7 の Web Profile 13
Java Enterprise Edition 7 の Full Profile 14
1.3. 開発環境のセットアップ 15
1.3.1. JBoss Developer Studio のダウンロード 15
1.3.2. JBoss Developer Studio のインストール 15
1.3.3. JBoss Developer Studio の起動 16
1.3.4. JBoss EAP サーバーを JBoss Developer Studio へ追加 16
1.4. クイックスタートサンプルの使用 20
1.4.1. Maven について 20
1.4.1.1. クイックスタートを用いた Maven の使用 21
1.4.2. クイックスタートコードサンプルのダウンロードおよび実行 21
1.4.2.1. クイックスタートのダウンロード 21
1.4.2.2. JBoss Developer Studio でのクイックスタートの実行 22
1.4.2.3. コマンドラインでのクイックスタートの実行 28
1.4.3. クイックスタートチュートリアルの確認 28
1.4.3.1. helloworld クイックスタート 28
前提条件 29
ディレクトリー構造の確認 29
コードの確認 29
1.4.3.2. numberguess クイックスタート 30
前提条件 31
設定ファイルの確認 31
1.4.3.2.1. JSF コードの確認 32
1.4.3.2.2. クラスファイルの確認 33
1.5. デフォルトの WELCOME WEB アプリケーションの設定 37
welcome-content ファイルハンドラーの変更 37
default-web-module の変更 38
デフォルトの Welcome Web アプリケーションの無効化 38

第2章 JBOSS EAP での MAVEN の使用 39
2.1. MAVEN について 39
2.1.1. Maven リポジトリー 39
2.1.2. Maven POM ファイル 39
Maven POM ファイルの最低要件 39
2.1.3. Maven 設定ファイル 40
2.1.4. Maven リポジトリーマネージャー 41
一般的に使用される Maven リポジトリーマネージャー 41
2.2. MAVEN と JBOSS EAP MAVEN リポジトリーのインストール 42
2.2.1. Maven のダウンロードとインストール 42
2.2.2. JBoss EAP の Maven リポジトリーのインストール 42
2.2.3. JBoss EAP Maven リポジトリーのローカルインストール 42
2.2.4. Apache httpd で使用する JBoss EAP Maven レポジトリーのインストール 43
2.3. MAVEN リポジトリーの使用 43
2.3.1. JBoss EAP Maven リポジトリーの設定 43
Maven 設定を使用した JBoss EAP Maven リポジトリーの設定 44
プロジェクト POM を使用した JBoss EAP Maven リポジトリーの設定 46

JBoss EAP リポジトリーの URL の確認 48
2.3.2. Red Hat JBoss Developer Studio で使用する Maven の設定 48
2.3.3. プロジェクト依存関係の管理 51
サポート対象の Maven アーティファクト 51
依存関係管理 52
JBoss EAP Java EE 仕様の BOM 53
JBoss EAP BOM とクイックスタート 53
JBoss EAP クライアント BOM 54

第3章 クラスローディングとモジュール 56
3.1. はじめに 56
3.1.1. クラスロードとモジュールの概要 56
3.1.2. モジュール 56
静的モジュール 56
動的モジュール 57
3.1.3. モジュールの依存関係 57
オプションの依存関係 58
依存関係のエクスポート 58
グローバルモジュール 58
3.1.3.1. 管理 CLI を使用したモジュールの依存関係の表示 58
3.1.4. デプロイメントでのクラスローディング 59
3.1.5. クラスローディングの優先順位 60
3.1.6. 動的モジュールの命名規則 60
3.1.7. jboss-deployment-structure.xml 61
3.2. デプロイメントへの明示的なモジュール依存関係の追加 61
前提条件 61
MANIFEST.MF への依存関係設定の追加 61
jboss-deployment-structure.xml への依存関係設定の追加 62
Jandex インデックスの作成 63
3.3. MAVEN を使用した MANIFEST.MF エントリーの生成 65
モジュール依存関係が含まれる MANIFEST.MF ファイルの生成 65
3.4. モジュールが暗黙的にロードされないようにする 66
3.5. サブシステムをデプロイメントから除外 67
3.6. デプロイメントでのプログラムを用いたクラスローダーの使用 68
3.6.1. デプロイメントでのプログラムによるクラスおよびリソースのロード 68
3.6.2. デプロイメントでのプログラムによるリソースの繰り返し 70
3.7. クラスローディングとサブデプロイメント 72
3.7.1. エンタープライズアーカイブのモジュールおよびクラスロード 72
3.7.2. サブデプロイメントクラスローダーの分離 73
3.7.3. EAR 内のサブデプロイメントクラスローダーの分離を有効にする 73
3.7.4. エンタープライズアーカイブのサブデプロイメント間で共有するセッションの設定 73
3.7.4.1. 共有セッション設定オプションのリファレンス 74
3.8. カスタムモードでのタグライブラリー記述子 (TLD) のデプロイ 77
カスタムモジュールでの TLD のデプロイ 77
3.9. 参考情報 78
3.9.1. 暗黙的なモジュール依存関係 78
3.9.2. 含まれるモジュール 87
3.9.3. JBoss デプロイメント構造のデプロイメント記述子 87

第4章 ロギング 88
4.1. ロギング 88
4.1.1. サポート対象のアプリケーションロギングフレームワーク 88
4.2. JJBOSS LOGGING FRAMEWORK を用いたロギング 88

4.2.1. JBoss Logging について 88
4.2.2. JBoss Logging を使用したアプリケーションへのロギングの追加 89
4.3. デプロイメントごとのロギング 91
4.3.1. デプロイメントごとのロギングをアプリケーションに追加 91
logging.properties の設定 91
JBoss ログマネージャーの設定オプション 91
4.4. ロギングプロファイル 93
4.4.1. アプリケーションでのロギングプロファイルの指定 94
4.5. 国際化と現地語化 95
4.5.1. はじめに 95
4.5.1.1. 国際化 95
4.5.1.2. 多言語化 95
4.5.2. JBoss Logging Tools の国際化および現地語化 95
4.5.3. 国際化されたロガー、メッセージ、例外の作成 97
4.5.3.1. 国際化されたログメッセージの作成 97
4.5.3.2. 国際化されたメッセージの作成と使用 98
4.5.3.3. 国際化された例外の作成 100
4.5.4. 国際化されたロガー、メッセージ、例外の現地語化 101
4.5.4.1. Maven での新しい翻訳プロパティーファイルの作成 101
4.5.4.2. 国際化されたロガー、例外、またはメッセージの翻訳 103
4.5.5. 国際化されたログメッセージのカスタマイズ 103
4.5.5.1. ログメッセージへのメッセージ ID とプロジェクトコードの追加 103
4.5.5.2. メッセージのログレベル設定 104
4.5.5.3. パラメーターによるログメッセージのカスタマイズ 105
4.5.5.4. 例外をログメッセージの原因として指定 105
4.5.6. 国際化された例外のカスタマイズ 106
4.5.6.1. メッセージ ID およびプロジェクトコードの例外メッセージへの追加 107
4.5.6.2. パラメーターによる例外メッセージのカスタマイズ 108
4.5.6.3. 別の例外の原因として 1 つの例外を指定 109
4.5.7. 参考資料 110
4.5.7.1. JBoss Logging Tools の Maven 設定 110
4.5.7.2. 翻訳プロパティーファイルの形式 111
4.5.7.3. JBoss Logging Tools のアノテーションに関するリファレンス 112
4.5.7.4. JBoss EAP で使用されるプロジェクトコード 113

第5章 リモート JNDI ルックアップ 117
5.1. JNDI へのオブジェクトの登録 117
5.2. リモート JNDI の設定 117

第6章 WEB アプリケーションのクラスター化 118
6.1. セッションレプリケーション 118
6.1.1. HTTP セッションレプリケーション 118
6.1.2. アプリケーションにおけるセッションレプリケーションの有効化 118
アプリケーションを配布可能にする 118
変更不能なセッション属性 119
6.2. HTTP セッションパッシベーションおよびアクティベーション 120
6.2.1. HTTP セッションパッシベーションおよびアクティベーション 120
6.2.2. アプリケーションでの HTTP セッションパッシベーションの設定 120
6.3. クラスタリングサービスのパブリック API 121
6.4. HA シングルトンサービス 122
HA シングルトン ServiceBuilder API 122
HA シングルトンサービス選択ポリシー 122
HA シングルトンサービスアプリケーションの作成 122

6.5. HA シングルトンデプロイメント 126
シングルトンデプロイメントの定義または選択 126
シングルトンデプロイメントの作成 127
設定 128
クォーラム 129
6.6. APACHE MOD_CLUSTER-MANAGER アプリケーション 129
6.6.1. mod_cluster-manager アプリケーション 129
mod_cluster-manager アプリケーションの使用 130

第7章 コンテキストおよび依存関係の挿入 (CDI) 132
7.1. CDI の概要 132
7.1.1. コンテキストと依存関係の注入 (CDI) 132
CDI の利点 132
7.1.2. Weld、Seam 2、および JavaServer Faces 間の関係 132
7.2. CDI の有効化 132
JBoss EAP での CDI の有効化 133
7.3. CDI を使用したアプリケーションの開発 133
7.3.1. デフォルトの Bean 検出モード 133
bean を定義するアノテーション 134
7.3.2. スキャンプロセスからの Bean の除外 134
7.3.3. インジェクションを使用した実装の拡張 136
7.4. あいまいな依存関係または満たされていない依存関係 136
7.4.1. 修飾子 137
'@Any' 137
7.4.2. 修飾子を使用したあいまいなインジェクションの解決 138
修飾子を使用したあいまいなインジェクションの解決 138
7.5. 管理 BEAN 139
7.5.1. Bean であるクラスのタイプ 139
@Vetoed 140
7.5.2. CDI を用いたオブジェクトの Bean へのインジェクト 140
他のオブジェクトにオブジェクトをインジェクトする 140
7.6. コンテキストおよびスコープ 141
7.7. 名前付き BEAN 142
7.7.1. 名前付き Bean の使用 142
@Named Annotation を使用した Bean 名の設定 142
7.8. BEAN ライフサイクル 143
Bean ライフサイクルの管理 143
7.8.1. プロデューサーメソッドの使用 143
7.9. 代替の BEAN 145
選択された代替の宣言 145
7.9.1. 代替を用いたインジェクションのオーバーライド 146
インジェクションのオーバーライド 146
7.10. ステレオタイプ 146
7.10.1. ステレオタイプの使用 147
ステレオタイプの定義および使用 147
7.11. オブザーバーメソッド 147
7.11.1. イベントの発生と確認 148
7.11.2. トランザクションオブザーバー 148
7.12. インターセプター 150
インターセプターの有効化 151
7.12.1. CDI とのインターセプターの使用 151
CDI とのインターセプターの使用 152
7.13. デコレーター 152

7.14. 移植可能な拡張機能 154
7.15. BEAN プロキシ 154
7.16. インジェクションでのプロキシの使用 154

第8章 JBOSS EAP MBEAN サービス 156
8.1. JBOSS MBEAN SERVICE の記述 156
8.1.1. 標準の MBean の例 156
8.2. JBOSS MBEAN サービスのデプロイ 158

第9章 コンカレンシーユーティリティー 159
9.1. コンテキストサービス 160
9.2. 管理対象スレッドファクトリー 160
9.3. 管理対象エグゼキューターサービス 161
9.4. 管理対象スケジュール済みエグゼキューターサービス 162

第10章 UNDERTOW 164
10.1. UNDERTOW ハンドラーについて 164
リクエストライフサイクル 164
交換の終了 165
10.2. デプロイメントでの既存の UNDERTOW ハンドラーの使用 165
10.3. カスタムハンドラーの作成 166

第11章 JAVA トランザクション API (JTA) 169
11.1. 概要 169
11.1.1. Java トランザクション API (JTA) の概要 169
11.2. トランザクションの概念 169
11.2.1. トランザクション 169
11.2.2. トランザクションの ACID プロパティー 169
11.2.3. トラザクションコーディネーターまたはトランザクションマネージャー 170
11.2.4. トランザクションの参加者 170
11.2.5. Java Transactions API (JTA) 170
11.2.6. Java Transaction Service (JTS) 171
11.2.7. XA リソースおよび XA トランザクション 171
11.2.8. XA リカバリー 171
11.2.9. XA リカバリープロセスの制限 172
11.2.10. 2 フェーズコミットプロトコル 173
フェーズ 1: 準備 173
フェーズ 2: コミット 173
11.2.11. トランザクションタイムアウト 173
11.2.12. 分散トランザクション 173
11.2.13. ORB 移植性 API 174
11.3. トランザクションの最適化 174
11.3.1. トランザクション最適化の概要 174
11.3.2. 1 フェーズコミット (1PC) の LRCO 最適化 175
1 フェーズコミット (1PC) 175
最終リソースコミット最適化 (LRCO: Last Resource Commit Optimization) 175
11.3.2.1. Commit Markable Resource (CMR) 175
概要 176
データベースでテーブルを作成する 176
データソースを接続可能にする 177
新しい CMR 機能を使用するために既存のリソースを更新 177
参照をトランザクションサブシステムに追加する 177
11.3.3. 推定中止 (presumed-abort) の最適化 178
11.3.4. 読み取り専用の最適化 178

11.4. トランザクションの結果 178
11.4.1. トランザクションの結果 178
11.4.2. トランザクションのコミット 179
11.4.3. トランザクションロールバック 179
11.4.4. ヒューリスティックな結果 179
ヒューリスティックロールバック 179
ヒューリスティックコミット 179
ヒューリスティック混合 179
ヒューリスティックハザード 179
11.4.5. JBoss Transactions エラーと例外 179
11.5. トランザクションライフサイクルの概要 180
11.5.1. トランザクションライフサイクル 180
11.6. トランザクションサブシステムの設定 181
11.6.1. トランザクション設定の概要 181
はじめに 181
11.6.2. トランザクションマネージャーの設定 181
管理コンソールを使用したトランザクションマネージャーの設定 181
管理 CLI を使用したトランザクションマネージャーの設定 181
11.6.3. トランザクションロギング 181
11.6.3.1. トランザクションログメッセージ 181
11.6.3.2. トランザクションサブシステムのロギング設定 182
管理コンソールを使用したトランザクションロガーの設定 182
管理 CLI を使用したトランザクションロガーの設定 183
11.6.4. トランザクションの参照と管理 183
ログストアの更新 183
準備済みトランザクションすべての表示 183
11.6.4.1. トランザクションの管理 183
トランザクションの属性の表示 183
トランザクションの参加者の表示 184
トランザクションの削除 184
トランザクションのリカバリー 184
リカバリーが必要なトランザクションの状態を更新します。 185
11.6.5. トランザクション統計情報の表示 185
11.7. 実際のトランザクションの使用 186
11.7.1. トランザクション使用の概要 186
11.7.2. トランザクションの制御 186
11.7.3. トランザクションの開始 186
11.7.4. ネストされたトランザクション 187
11.7.5. トランザクションのコミット 188
11.7.6. トランザクションのロールバック 189
11.7.7. トランザクションにおけるヒューリスティックな結果の処理方法 190
11.7.8. JTA トランザクションのエラー処理 191
11.7.8.1. トランザクションエラーの処理 191
11.8. トランザクションに関するリファレンス 192
11.8.1. JTA トランザクションの例 192
11.8.2. トランザクション API ドキュメンテーション 194

第12章 JAVA 永続 API (JPA) 195
12.1. JAVA 永続 API (JPA) について 195
12.2. HIBERNATE CORE 195
12.3. HIBERNATE ENTITYMANAGER 195
12.4. 単純な JPA アプリケーションの作成 196
12.5. CONFIGURATION (設定) 199

12.5.1. Hibernate 設定プロパティー 199
12.5.2. Hibernate JDBC と接続プロパティー 202
12.5.3. Hibernate キャッシュプロパティー 204
12.5.4. Hibernate トランザクションプロパティー 205
12.5.5. その他の Hibernate プロパティー 205
12.5.6. Hibernate SQL 方言 207
12.6. 2 次キャッシュ 209
12.6.1. 2 次キャッシュ 209
12.6.2. Hibernate 用 2 次キャッシュの設定 209
Hibernate ネイティブアプリケーションを使用した Hibernate 用 2 次キャッシュの設定 209
JPA ネイティブアプリケーションを使用した Hibernate 用 2 次キャッシュの設定 209
12.7. HIBERNATE アノテーション 210
12.8. HIBERNATE クエリー言語 216
12.8.1. Hibernate クエリー言語 217
JPQL の概要 217
HQL の概要 217
12.8.2. HQL ステートメントについて 217
12.8.3. INSERT ステートメント 218
12.8.4. FROM 節 218
12.8.5. WITH 節 219
12.8.6. HQL の順序付け 219
12.8.7. 一括更新、一括送信、および一括削除 220
12.8.8. コレクションメンバーの参照 221
12.8.9. 修飾パス式 222
12.8.10. スカラー関数 223
12.8.11. HQL の標準化された関数について 223
12.8.12. 連結演算 224
12.8.13. 動的インスタンス化 224
12.8.14. HQL 述語 225
HQL 述語 225
12.8.15. 関係比較 228
12.9. HIBERNATE サービス 229
12.9.1. Hibernate サービス 229
12.9.2. サービスコントラクト 229
12.9.3. サービス依存関係のタイプ 229
12.9.4. サービスレジストリー 230
12.9.4.1. ServiceRegistry 230
12.9.5. カスタムサービス 230
12.9.5.1. カスタムサービス 230
12.9.6. ブートストラップレジストリー 232
12.9.6.1. ブートストラップレジストリー 232
12.9.6.2. BootstrapServiceRegistryBuilder の使用 232
12.9.6.3. BootstrapRegistry サービス 232
12.9.7. SessionFactory レジストリー 233
12.9.7.1. SessionFactory レジストリー 233
12.9.7.2. SessionFactory サービス 233
12.9.8. インテグレーター 234
12.9.8.1. インテグレーター 234
12.9.8.2. インテグレーターのユースケース 234
12.10. ENVERS 235
12.10.1. Hibernate Envers 235
12.10.2. 永続クラスの監査 235
12.10.3. 監査ストラテジー 235

12.10.3.1. 監査ストラテジー 235
12.10.3.2. 監査ストラテジーの設定 236
監査ストラテジーの定義 236
12.10.4. JPA エンティティーへの監査サポートの追加 236
12.10.5. Configuration (設定) 238
12.10.5.1. Envers パラメーターの設定 238
12.10.5.2. ランタイム時に監査を有効または無効にする 238
12.10.5.3. 条件付き監査の設定 239
12.10.5.4. Envers の設定プロパティー 239
12.10.6. クエリーを介した監査情報の読み出し 241
12.11. パフォーマンスチューニング 244
12.11.1. 代替のバッチローディングアルゴリズム 244
12.11.2. 不変データのオブジェクト参照の 2 次キャッシング 246

第13章 HIBERNATE SEARCH 248
13.1. HIBERNATE SEARCH を初めて使う場合 248
13.1.1. Hibernate Search について 248
13.1.2. 概要 248
13.1.3. Directory Provider について 249
13.1.4. ワーカーについて 249
13.1.5. バックエンドセットアップおよび操作 249
13.1.5.1. バックエンド 249
13.1.5.2. Lucene 249
13.1.5.3. JMS 250
13.1.6. リーダーストラテジー 251
13.1.6.1. shared ストラテジー 251
13.1.6.2. not-shared ストラテジー 252
13.1.6.3. カスタムリーダーストラテジー 252
13.2. CONFIGURATION (設定) 252
13.2.1. 最小設定 252
13.2.2. IndexManager の設定 252
13.2.2.1. Directory-based 252
13.2.2.2. Near Real Time 252
13.2.2.3. Custom 253
13.2.3. DirectoryProvider 設定 253
ディレクトリープロバイダーおよびプロパティー 254
13.2.4. ワーカー設定 256
13.2.4.1. JMS マスター/スレーブバックエンド 259
13.2.4.2. スレーブノード 260
13.2.4.3. マスターノード 260
13.2.5. Lucene インデックス作成のチューニング 261
13.2.5.1. Lucene インデックス作成パフォーマンスのチューニング 261
13.2.5.2. Lucene IndexWriter 266
13.2.5.3. パフォーマンスオプション設定 266
13.2.5.4. インデックス作成速度のチューニング 270
13.2.5.5. セグメントサイズの制御 270
13.2.6. LockFactory 設定 270
13.2.7. インデックス形式の互換性 272
13.3. アプリケーション用 HIBERNATE SEARCH 272
13.3.1. Hibernate Seach の最初のステップ 272
13.3.2. Maven を使用した Hibernate Seach の有効化 272
13.3.3. アノテーションの追加 273
13.3.4. インデックス化 275

13.3.5. 検索 276
13.3.6. アナライザー 277
13.4. インデックス構造へのエンティティーのマッピング 278
13.4.1. エンティティーのマッピング 278
13.4.1.1. 基本的なマッピング 279
13.4.1.2. @Indexed 279
13.4.1.3. @Field 279
13.4.1.4. @NumericField 281
13.4.1.5. @Id 282
13.4.1.6. プロパティーを複数回マッピングする 283
13.4.1.7. 埋め込みオブジェクトおよび関連付けられたオブジェクト 284
13.4.1.8. 特定パスへ埋め込むオブジェクトの制限 287
13.4.2. ブースティング 289
13.4.2.1. 静的なインデックス時ブースティング 289
13.4.2.2. 動的なインデックス時ブースティング 290
13.4.3. 分析 291
13.4.3.1. デフォルトのアナライザーとクラスによるアナライザー 291
13.4.3.2. 名前付きのアナライザー 292
13.4.3.3. 使用可能なアナライザー 294
13.4.3.4. 動的アナライザーの選択 296
13.4.3.5. アナライザーの読み出し 297
13.4.4. ブリッジ 298
13.4.4.1. ビルトインブリッジ 298
13.4.4.2. カスタムブリッジ 300
13.4.4.2.1. StringBridge 300
13.4.4.2.2. パラメーター化されたブリッジ 300
13.4.4.2.3. 型対応ブリッジ 301
13.4.4.2.4. 双方向ブリッジ 301
13.4.4.2.5. FieldBridge 302
13.4.4.2.6. ClassBridge 304
13.5. クエリー 305
13.5.1. クエリーの構築 307
13.5.1.1. Lucene API を使用した Lucene クエリーの構築 307
13.5.1.2. Lucene クエリーの構築 307
13.5.1.3. キーワードクエリー 308
13.5.1.4. ファジークエリー 310
13.5.1.5. ワイルドカードクエリー 310
13.5.1.6. フレーズクエリー 311
13.5.1.7. 範囲クエリー 311
13.5.1.8. クエリーの組み合わせ 311
13.5.1.9. クエリーオプション 312
13.5.1.10. Hibernate Search クエリーの構築 313
13.5.1.10.1. 一般論 313
13.5.1.10.2. ページネーション 314
13.5.1.10.3. 並び順 314
13.5.1.10.4. フェッチングストラテジー 314
13.5.1.10.5. 射影 (Projection) 315
13.5.1.10.6. オブジェクト初期化ストラテジーのカスタマイズ 316
13.5.1.10.7. クエリーの時間制限 317
13.5.1.10.8. 時間制限での例外の発生 317
13.5.2. 結果の読み出し 318
13.5.2.1. パフォーマンスに関する注意点 318
13.5.2.2. 結果サイズ 319

13.5.2.3. ResultTransformer 319
13.5.2.4. 結果の理解 320
13.5.2.5. Filters 321
13.5.2.6. シャード化された環境におけるフィルターの使用 324
13.5.3. ファセット 326
13.5.3.1. ファセットリクエストの作成 328
13.5.3.2. ファセットリクエストの適用 329
13.5.3.3. クエリー結果の制限 330
13.5.4. クエリー処理の最適化 330
13.5.4.1. インデックス値のキャッシュ: FieldCache 331
13.6. 手動によるインデックスの変更 332
13.6.1. インデックスへのインスタンスの追加 332
13.6.2. インデックスからのインスタンスの削除 332
13.6.3. インデックスの再構築 333
13.6.3.1. flushToIndexes() の使用 333
13.6.3.2. MassIndexer の使用 334
13.7. インデックスの最適化 336
13.7.1. 自動最適化 336
13.7.2. 手動の最適化 337
13.7.3. 最適化の調整 338
13.8. 高度な機能 338
13.8.1. SearchFactory へのアクセス 338
13.8.2. IndexReader の使用 338
13.8.3. Lucene Directory へのアクセス 339
13.8.4. インデックスのシャード化 339
13.8.5. Lucene のスコア計算式のカスタマイズ 341
13.8.6. 例外処理の設定 342
13.8.7. Hibernate Search の無効化 343
13.9. モニタリング 343
JMX を介した統計へのアクセス 344
インデックス作成の監視 344

第14章 BEAN の検証 345
14.1. BEAN VALIDATION 345
14.2. バリデーション制約 345
14.2.1. バリデーション制約 345
14.2.2. Red Hat JBoss Developer Studio での制約アノテーションの作成 345
14.2.3. Hibernate Validator の制約 346
14.3. バリデーション設定 349

第15章 WEBSOCKET アプリケーションの作成 351
WebSocket アプリケーションの作成 351

第16章 JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS) 355
16.1. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS) 355
16.2. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS) のセキュリティーの設定 355

第17章 JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) 357
17.1. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) のセキュリティー 357
17.2. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) のセキュリティーの設定 357

第18章 JAVA バッチアプリケーション開発 358
18.1. 必要なバッチ依存関係 358
18.2. JOB SPECIFICATION LANGUAGE (JSL) 継承 358

例: 同じジョブ XML ファイル内の step および flow の継承 358
例: 別のジョブ XML ファイルからの step の継承 359
18.3. バッチプロパティーインジェクション 360
例: 数字を Batchlet クラスにさまざまなタイプとしてインジェクトする 362
例: 数字シーケンスを Batchlet クラスにさまざまなアレイとしてインジェクトする 362
例: クラスプロパティーを Batchlet クラスにインジェクトする 363
例: プロパティーインジェクション向けにアノテートされたフィールドにデフォルト値を割り当てる 364

付録A リファレンス資料 365
A.1. 提供された UNDERTOW ハンドラー 365
AccessControlListHandler 365
AccessLogHandler 365
AllowedMethodsHandler 367
BlockingHandler 367
ByteRangeHandler 367
CanonicalPathHandler 368
DisableCacheHandler 368
DisallowedMethodsHandler 368
EncodingHandler 368
FileErrorPageHandler 368
HttpTraceHandler 369
IPAddressAccessControlHandler 369
JDBCLogHandler 369
LearningPushHandler 370
LocalNameResolvingHandler 370
PathSeparatorHandler 371
PeerNameResolvingHandler 371
ProxyPeerAddressHandler 371
RedirectHandler 371
RequestBufferingHandler 371
RequestDumpingHandler 372
RequestLimitingHandler 372
ResourceHandler 372
ResponseRateLimitingHandler 372
SetHeaderHandler 373
SSLHeaderHandler 373
StuckThreadDetectionHandler 374
URLDecodingHandler 374

第1章 アプリケーションの開発

1.1. はじめに

1.1.1. Red Hat JBoss Enterprise Application Platform 7 について

Red Hat JBoss Enterprise Application Platform 7 (JBoss EAP) は、オープンな標準に基いて構築され、Java Enterprise Edition 7 の仕様に準拠するミドルウェアプラットフォームです。メッセージングや高可用性クラスタリングなどの技術と WildFly Application Server 10 が統合されます。

JBoss EAP には、必要な場合にだけサービスを有効にできるモジュール構造が含まれ、サービスの起動時間が短縮されます。

管理コンソールと管理コマンドラインインターフェース (CLI) では、XML 設定ファイルの編集が不要になり、タスクをスクリプト化および自動化する機能が追加されました。

JBoss EAP は、JBoss EAP インスタンスに対してスタンドアロンサーバーと管理対象ドメインの 2 つの操作モードを提供します。スタンドアロンサーバー操作モードでは、実行している JBoss EAP を 1 つのサーバーインスタンスとして表します。管理対象ドメイン操作モードでは、1 つの制御ポイントから複数の JBoss EAP インスタンスを管理できます。

また、JBoss EAP には、セキュアでスケーラブルな Java EE アプリケーションの迅速な開発を可能にする API と開発フレームワークが含まれます。

1.2. JAVA ENTERPRISE EDITION 7 について

1.2.1. EE 7 プロファイルの概要

Java Enterprise Edition 7 (EE 7) には、複数のプロファイルのサポート (つまり、API のサブセット) が含まれます。EE 7 の仕様で定義されるプロファイルは、Full Profile と Web Profile の 2 つのみです。

EE 7 の Full Profile には、EE 7 の仕様に含まれるすべての API と仕様が含まれます。EE 7 の Web Profile には、Web 開発者に役立つよう設計された特別な API のサブセットが含まれます。

JBoss EAP は、Java Enterprise Edition 7 の Full Profile および Web Profile 仕様の認定実装です。

- Java Enterprise Edition 7 の Web Profile
- Java Enterprise Edition 7 の Full Profile

Java Enterprise Edition 7 の Web Profile
Web Profile は、Java Enterprise Edition 7 仕様で定義されている 2 つのプロファイルの 1 つであり、Web アプリケーション開発向けに設計されています。Web Profile は以下の API をサポートします。

- Java EE 7 Web Profile の要件:
  - Java Platform、Enterprise Edition 7
- Java Web テクノロジー:
  - Servlet 3.1 (JSR 340)
  - JSP 2.3
  - Expression Language (EL) 3.0

    - JavaServer Faces (JSF) 2.2 (JSR 344)
    - JSP 1.2 向け Java Standard Tag Library (JSTL)
    - 他言語のデバッグサポート 1.0 (JSR 45)
- エンタープライズアプリケーションテクノロジー:
    - Contexts and Dependency Injection (CDI) 1.1 (JSR 346)
    - Dependency Injection for Java 1.0 (JSR 330)
    - Enterprise JavaBeans 3.2 Lite (JSR 345)
    - Java Persistence API 2.1 (JSR 338)
    - Java Platform 1.1 向けの共通アノテーション (JSR 250)
    - Java Transaction API (JTA) 1.2 (JSR 907)
    - Bean Validation 1.1 (JSR 349)

Java EE 7 の仕様で定義されている他のプロファイルは Full Profile であり、他の複数の API を含みます。

Java Enterprise Edition 7 の Full Profile
Java Enterprise Edition 7 (EE 7) の仕様により、プロファイルの概念が定義され、それらのプロファイルの 2 つが仕様の一部として定義されます。Full Profile は次の API と、Java Enterprise Edition 7 Web Profile でサポートされいている API をサポートします。

- EE 7 Full Profile に含まれる API:
    - Batch 1.0
    - JSON-P 1.0
    - Concurrency 1.0
    - WebSocket 1.1
    - JMS 2.0
    - JPA 2.1
    - JCA 1.7
    - JAX-RS 2.0
    - JAX-WS 2.2
    - Servlet 3.1
    - JSF 2.2
    - JSP 2.3
    - EL 3.0

- CDI 1.1
- CDI Extensions
- JTA 1.2
- Interceptors 1.2
- Common Annotations 1.1
- Managed Beans 1.0
- EJB 3.2
- Bean Validation 1.1

1.3. 開発環境のセットアップ

1.3.1. JBoss Developer Studio のダウンロード

JBoss Developer Studio は Red Hat カスタマーポータルからダウンロードできます。

1. Red Hat カスタマーポータルにログインします。
2. **ダウンロード**をクリックします。
3. **製品のダウンロード**リストで **Red Hat JBoss Developer Studio** をクリックします。
4. **Version** ドロップダウンメニューで希望のバージョンを選択します。

注記

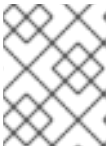
JBoss Developer Studio バージョン 9.1 以上の使用が推奨されます。

5. 表で **Red Hat JBoss Developer Studio 9.x.x Stand-alone Installer** を見つけ、**Download** をクリックします。
6. JAR ファイルを希望のディレクトリーに保存します。

1.3.2. JBoss Developer Studio のインストール

1. ターミナルを開き、ダウンロードした JAR ファイルが含まれるディレクトリーに移動します。
2. 次のコマンドを実行して GUI インストールプログラムを起動します。

```
$ java -jar jboss-devstudio-BUILD_VERSION-installer-standalone.jar
```

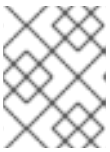


注記

または、JAR ファイルをダブルクリックしてインストールプログラムを起動することもできます。

3. **Next** をクリックしてインストールプロセスを開始します。

4. **I accept the terms of this license agreement** (ライセンス契約の内容に同意します) を選択し、**Next** をクリックします。
5. インストールパスを調整し、**Next** をクリックします。



注記

インストールパスフォルダーが存在しない場合は、メッセージが表示されます。**OK** をクリックしてフォルダーを作成します。

6. JVM を選択するか、デフォルトの JVM を選択したままにし、**Next** をクリックします。
7. プラットフォームとサーバーの選択を要求されたら、**Next** をクリックします。
8. インストールの詳細を確認し、**Next** をクリックします。
9. インストールプロセスが完了したら **Next** をクリックします。
10. JBoss Developer Studio のデスクトップショートカットを設定し、**Next** をクリックします。
11. **Done** をクリックします。

1.3.3. JBoss Developer Studio の起動

JBoss Developer Studio を起動するには、インストール中に作成されたデスクトップショートカットをダブルクリックするか、コマンドラインから起動します。コマンドラインを使用して JBoss Developer Studio を起動するには、以下の手順に従います。

1. ターミナルを開き、JBoss Developer Studio インストールディレクトリーに移動します。
2. 次のコマンドを実行して JBoss Developer Studio を起動します。

```
$ ./jbdevstudio
```

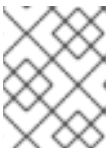
注記

Windows Server の場合は **`jbdevstudio.bat`** ファイルを使用します。

1.3.4. JBoss EAP サーバーを JBoss Developer Studio へ追加

この手順では、JBoss EAP サーバーが JBoss Developer Studio に追加されていないことを前提とします。以下の手順に従い、**Define New Server** ウィザードを使用して JBoss EAP サーバーを追加します。

1. **Servers** タブを開きます。

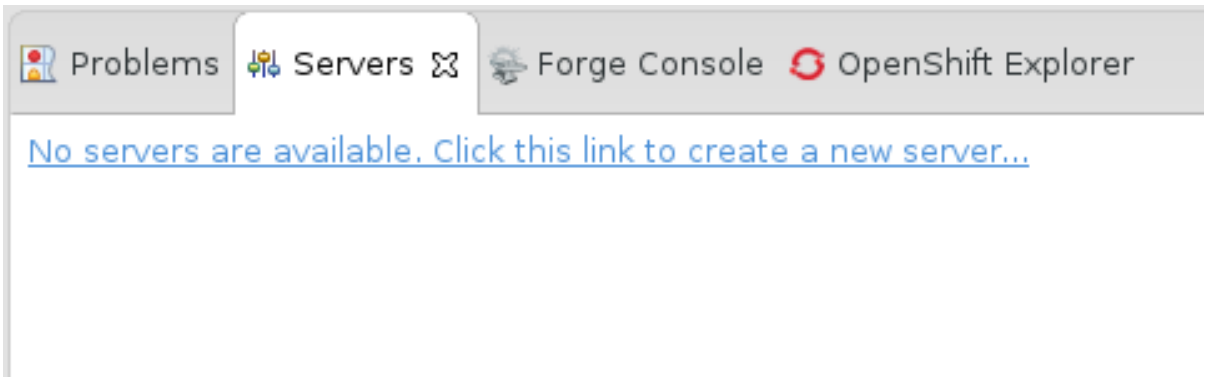


注記

Servers タブが表示されていない場合は、**Window** → **Show View** → **Servers** と選択してパネルに追加します。

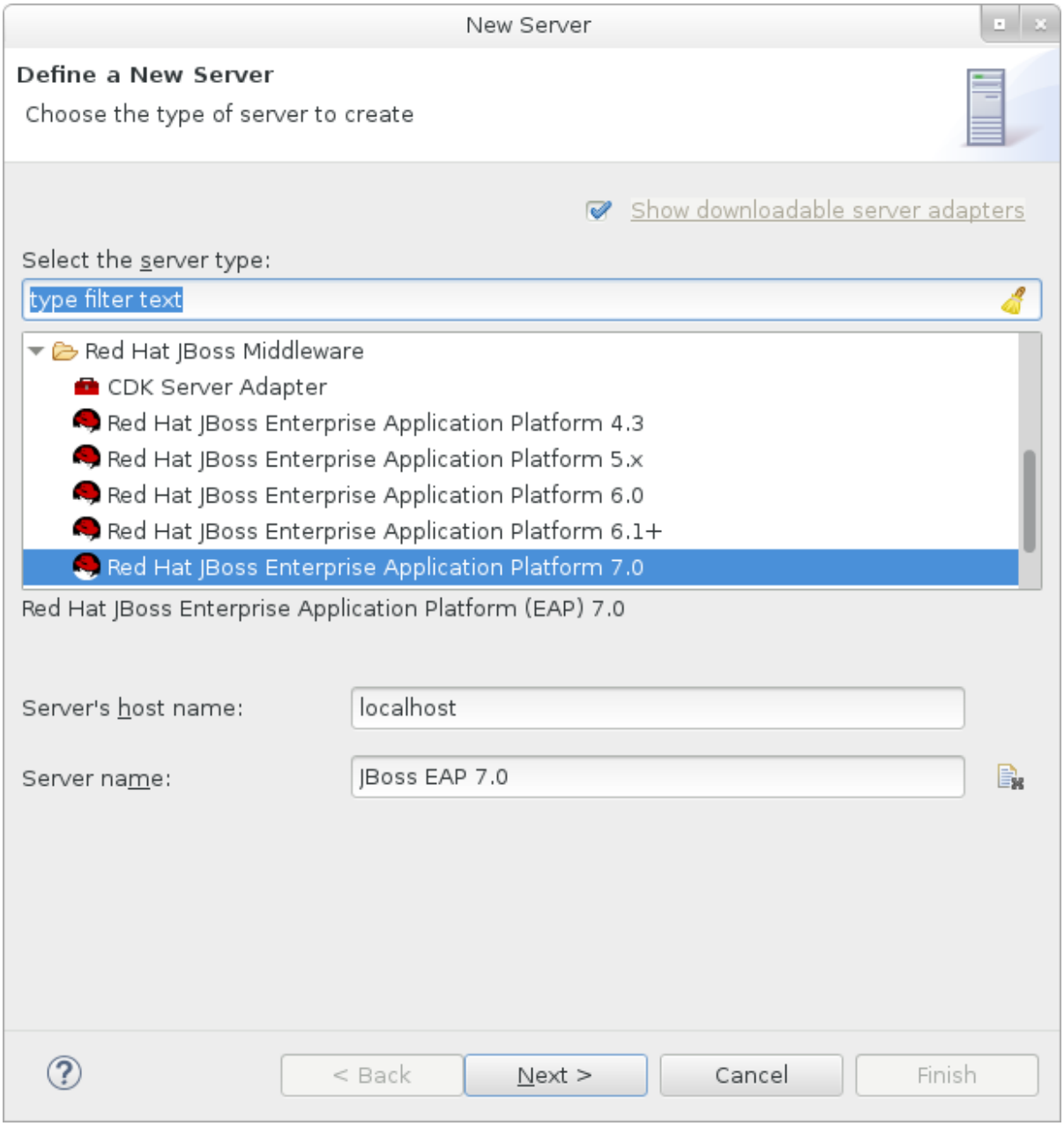
2. **No servers are available. Click this link to create a new server**(使用できるサーバーがありません。このリンクをクリックして新しいサーバーを作成してください) のリンクをクリックします。

図1.1 新しいサーバーの追加



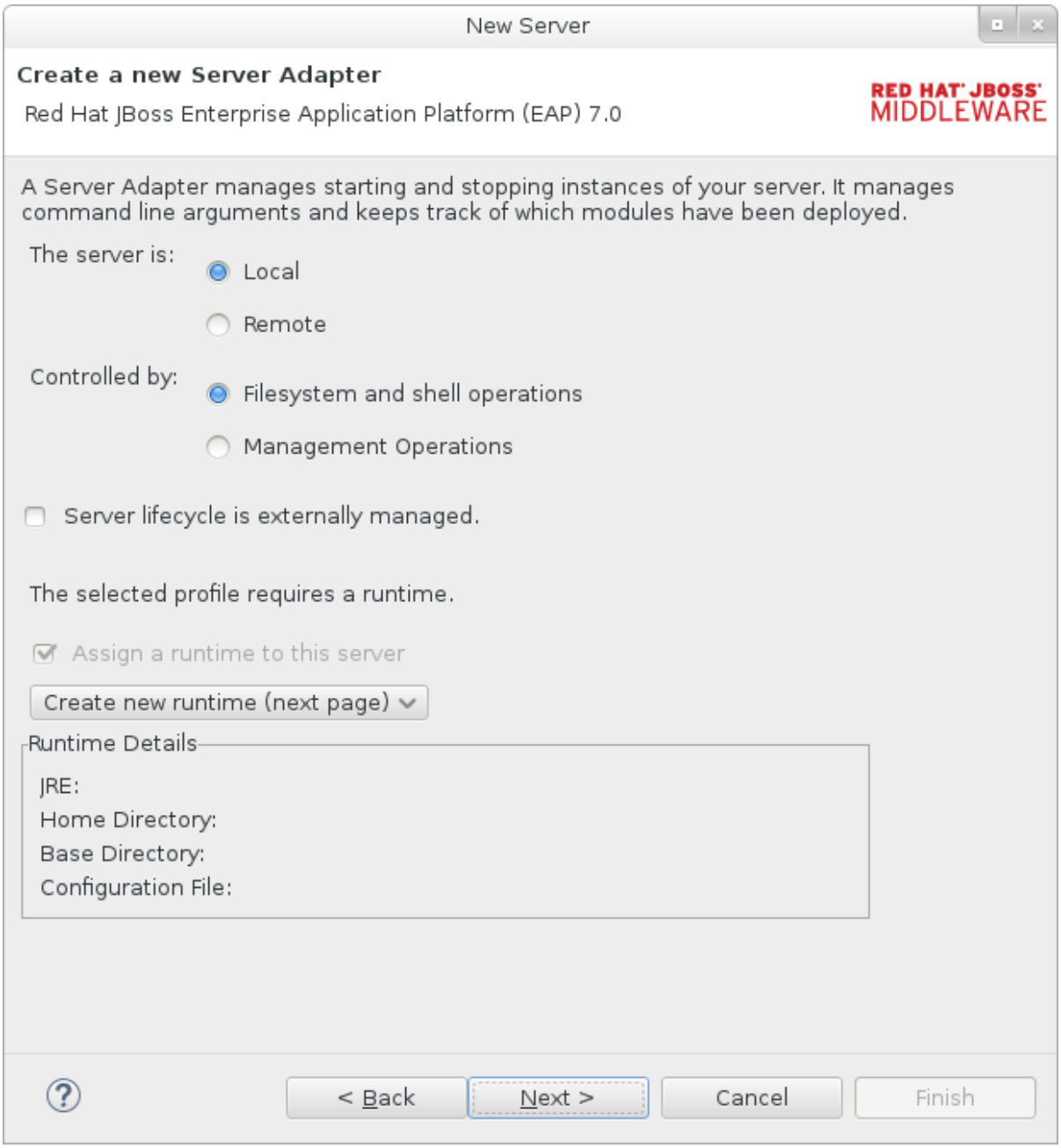
3. **Red Hat JBoss Middleware** を展開し、**JBoss Enterprise Application Platform 7.0** を選択します。サーバー名 (例: **JBoss EAP 7.0**) を入力し、**Next** をクリックします。

図1.2 新しいサーバーの定義



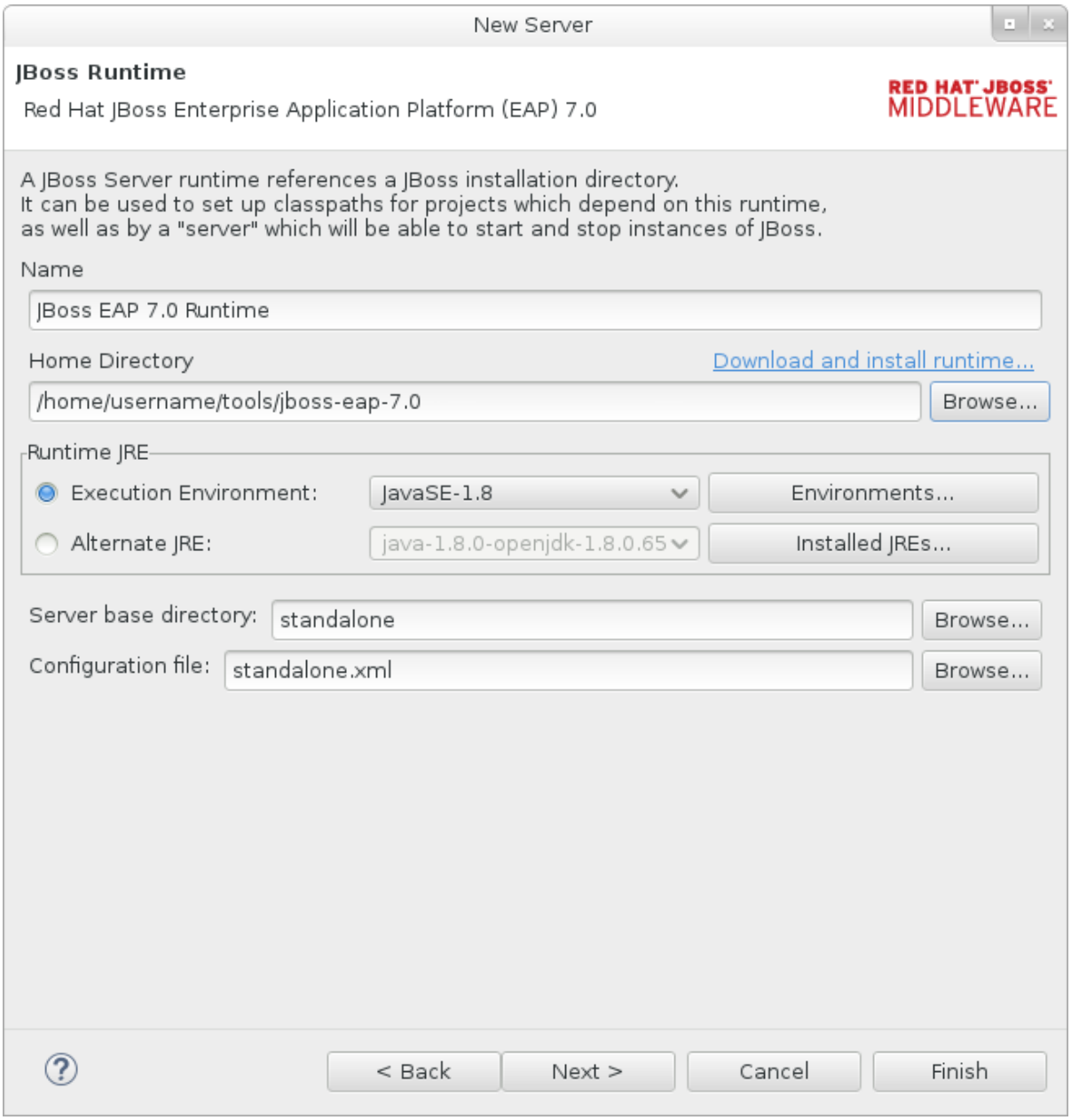
4. サーバーアダプターを作成し、サーバーの起動と停止を管理します。デフォルトの値のままにし、**Next** をクリックします。

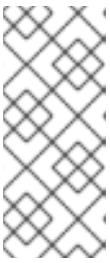
図1.3 新しいサーバーアダプターの作成



5. 名前 (例: **`JBoss EAP 7.0 Runtime`**) を入力します。**Home Directory** の横にある **Browse** をクリックし、JBoss EAP のインストールディレクトリーに移動します。次に、**Next** をクリックします。

図1.4 新しいサーバーランタイム環境の追加



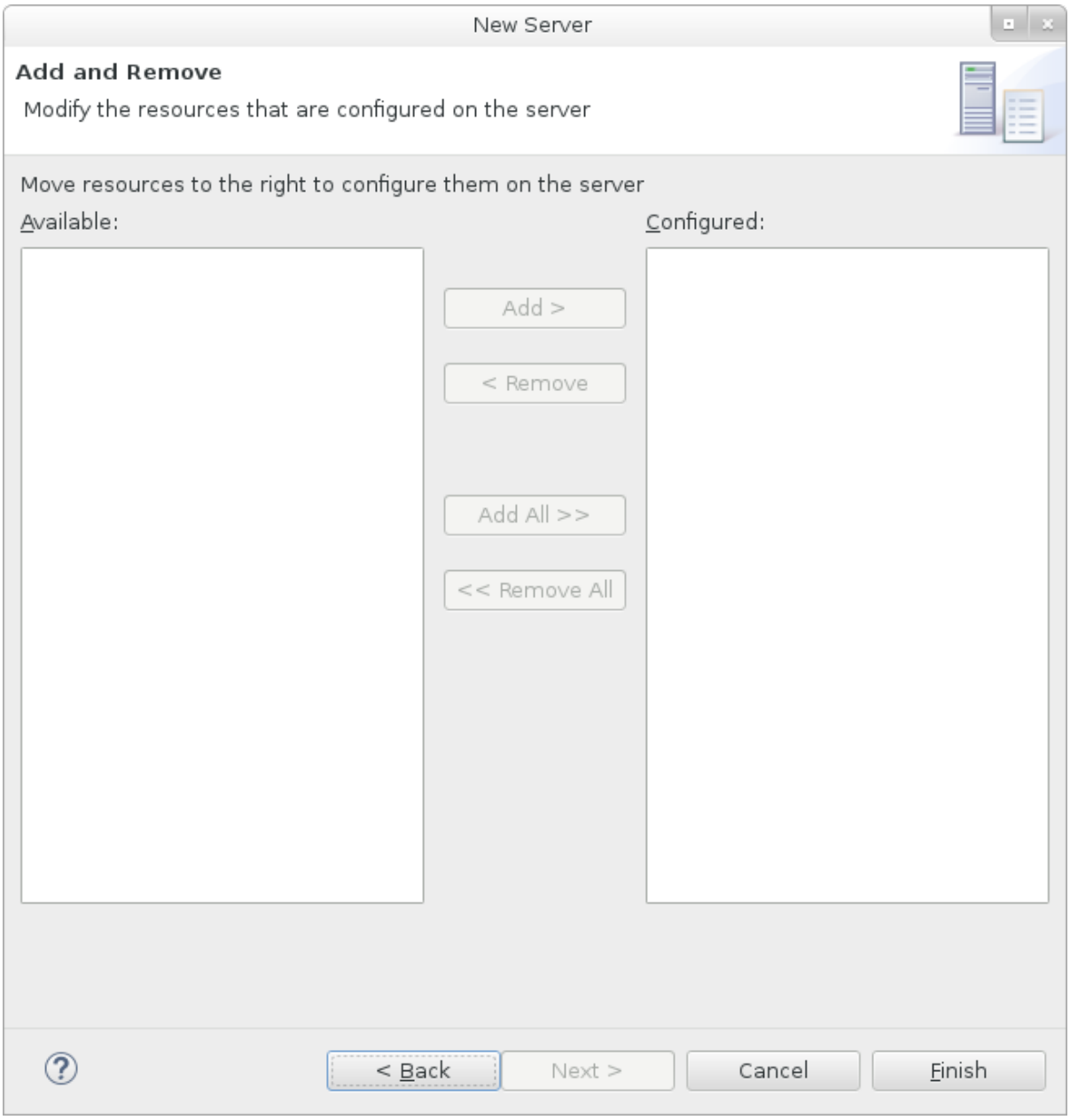


注記

一部のクイックスタートでは、異なるプロファイルまたは追加の引数を使用してサーバーを起動する必要があります。たとえば、**full** プロファイルが必要なクイックスタートをデプロイするには、新しいサーバーを定義し、**Configuration file** フィールドで **`standalone-full.xml`** を指定する必要があります。新しいサーバーにはその内容を表す名前を付けてください。

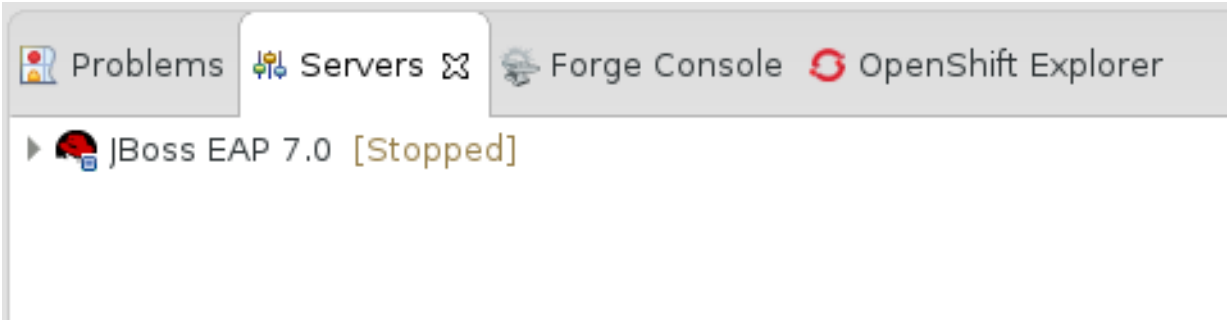
6. 新しいサーバーの既存プロジェクトを設定します。この時点ではプロジェクトは存在しないため **Finish** をクリックします。

図1.5 新しいサーバーのリソースの変更



`JBoss EAP 7.0` サーバーが **Servers** タブにリストされます。

図1.6 サーバーリスト



1.4. クイックスタートサンプルの使用

1.4.1. Maven について

Apache Maven は、ソフトウェアプロジェクトの作成、管理、および構築を行う Java アプリケーションの開発で使用される分散型ビルド自動化ツールです。Maven は Project Object Model (POM) と呼ばれる標準の設定ファイルを利用して、プロジェクトの定義や構築プロセスの管理を行います。POM はモジュールやコンポーネントの依存関係、ビルドの順番、結果となるプロジェクトパッケージングのターゲットを定義し、XML ファイルを使用して出力します。この結果、プロジェクトが適切かつ統一された状態で構築されるようになります。

Maven は、リポジトリーを使用してアーカイブを行います。Maven リポジトリーには Java ライブラリー、プラグイン、およびその他のビルドアーティファクトが格納されています。デフォルトのパブリックリポジトリーは Maven 2 Central Repository ですが、複数の開発チームの間で共通のアーティファクトを共有する目的で、社内のプライベートおよび内部リポジトリーとすることが可能です。また、サードパーティーのリポジトリーも利用できます。詳細については、Apache Maven プロジェクトおよび Introduction to Repositories ガイドを参照してください。

JBoss EAP には、Java EE 開発者が JBoss EAP 6 でアプリケーションを構築する際に一般的に使用する要件の多くを含む Maven リポジトリーが含まれます。

詳細については、Using Maven with JBoss EAP を参照してください。

1.4.1.1. クイックスタートを用いた Maven の使用

アプリケーションをビルドし、JBoss EAP 7 にデプロイするのに必要なアーティファクトと依存関係はパブリックリポジトリーでホストされます。JBoss EAP 7 のクイックスタートでは、Maven `settings.xml` ファイルを設定して、クイックスタートをビルドするときにこれらのリポジトリーを使用する必要がなくなりました。Maven リポジトリーはクイックスタートプロジェクト POM ファイルに設定されるようになりました。この設定方法は、クイックスタートを容易に使えるようにするために提供されますが、ビルドの処理が遅くなる可能性があるため、通常は本番プロジェクトでの使用は推奨されません。

Red Hat JBoss Developer Studio には Maven が含まれるため、個別にダウンロードおよびインストールする必要はありません。JBoss Developer Studio バージョン 9.1 以上を使用することが推奨されます。

Maven コマンドラインを使用してアプリケーションをビルドおよびデプロイする場合は、最初に Apache Maven プロジェクトから Maven をダウンロードし、Maven のドキュメントに記載されている手順に従ってインストールします。

1.4.2. クイックスタートコードサンプルのダウンロードおよび実行

1.4.2.1. クイックスタートのダウンロード

JBoss EAP には、さまざまな Java EE 7 の技術を使用してアプリケーションを作成するのに役立つ包括的なクイックスタートコードサンプルセットが含まれています。クイックスタートは Red Hat カスタマーポータルからダウンロードできます。

1. Red Hat カスタマーポータルにログインします。
2. **ダウンロード**をクリックします。
3. **製品のダウンロード**リストで **Red Hat JBoss Enterprise Application Platform** をクリックします。
4. **Version** ドロップダウンメニューで希望のバージョンを選択します。
5. 表で **Red Hat JBoss Enterprise Application Platform 7.0.0 Quickstarts** を見つけ、**Download** をクリックします。

6. ZIP ファイルを希望のディレクトリーに保存します。

7. Zip ファイルを展開します。

1.4.2.2. JBoss Developer Studio でのクイックスタートの実行

クイックスタートがダウンロードされたら、JBoss Developer Studio にインポートし、JBoss EAP にデプロイできます。

クイックスタートの JBoss Developer Studio へのインポート

各クイックスタートには、プロジェクトおよび設定情報が含まれる POM ファイルが同梱されています。この POM ファイルを使用すると、簡単にクイックスタートを JBoss Developer Studio にインポートできます。

重要

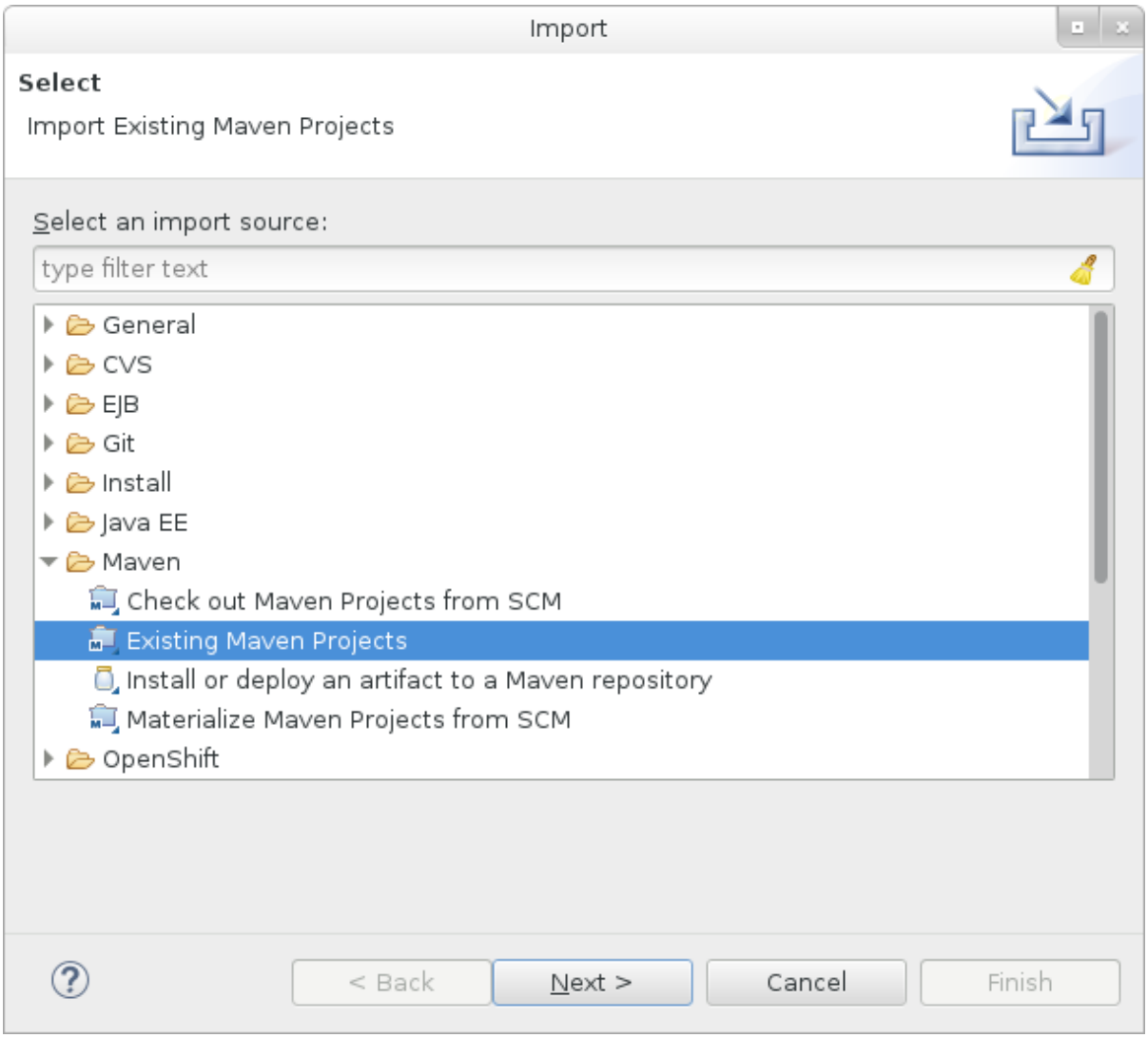
JBoss Developer Studio へのインポート時にクイックスタートプロジェクトフォルダーが IDE ワークスペース内にある場合は、IDE により無効なプロジェクト名と WAR アーカイブ名が生成されます。作業を開始する前に、クイックスタートプロジェクトフォルダーが IDE ワークスペースの外部にあることを確認してください。

1. JBoss Developer Studio を起動します。

2. **File** → **Import** と選択します。

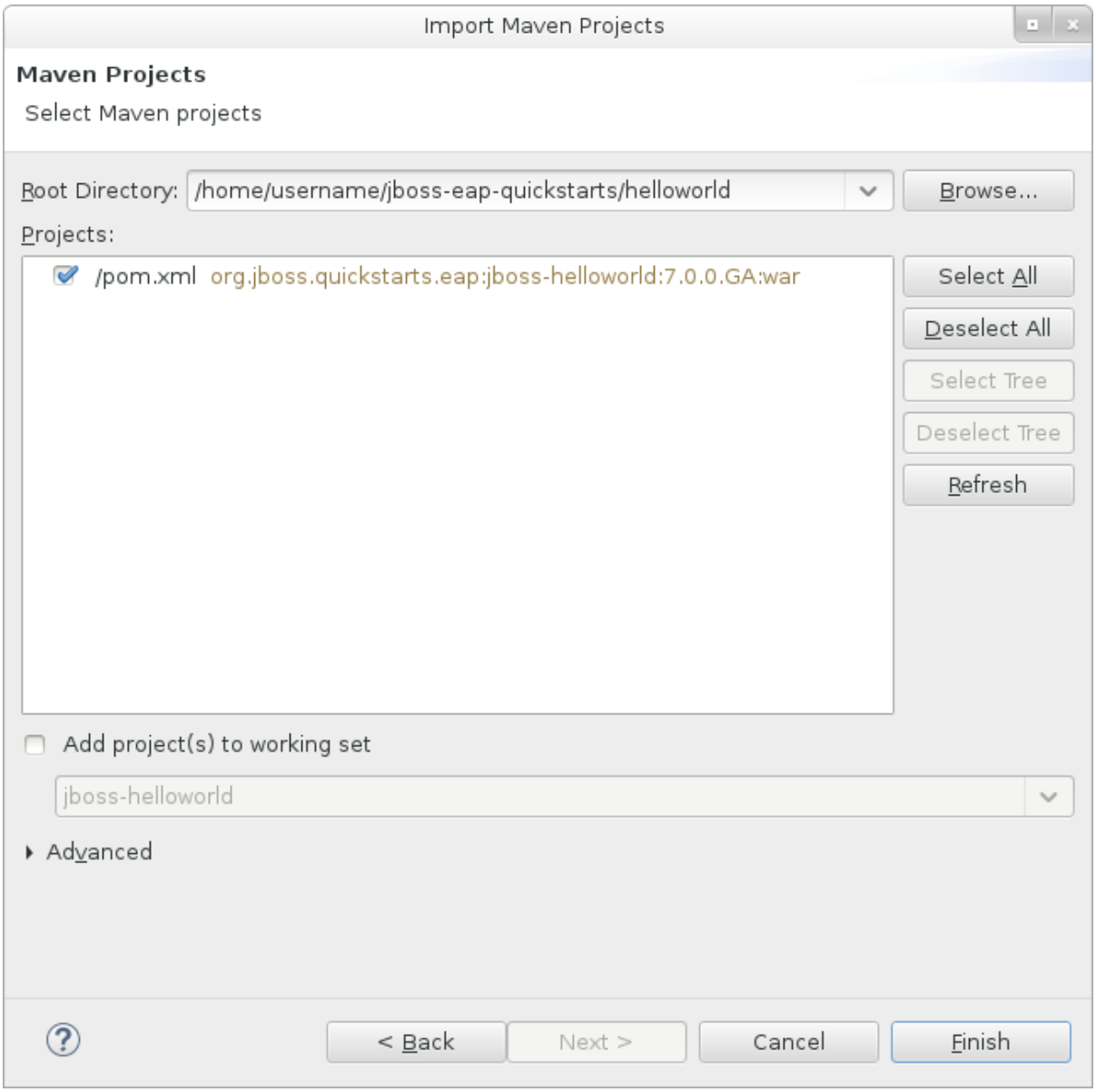
3. **Maven** → **Existing Maven Projects** と選択し、**Next** をクリックします。

図1.7 既存の Maven プロジェクトのインポート



4. 希望のクイックスタートのディレクトリー (**helloworld** など) を参照し、**OK** をクリックします。**Projects** リストボックスに、選択したクイックスタートプロジェクトの **pom.xml** ファイルが示されます。

図1.8 Maven プロジェクトの選択



5. **Finish** をクリックします。

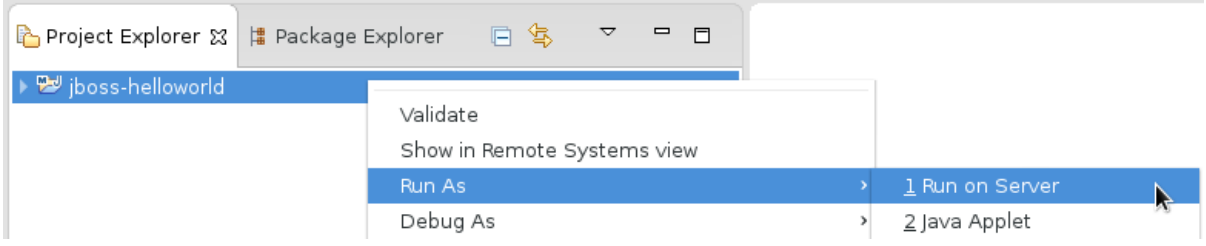
helloworld クイックスタートの実行

helloworld クイックスタートを実行すると、JBoss EAP サーバーが適切に設定および実行されたことを簡単に検証できます。

1. サーバーを定義していない場合は、JBoss EAP サーバーを JBoss Developer Studio へ追加します。

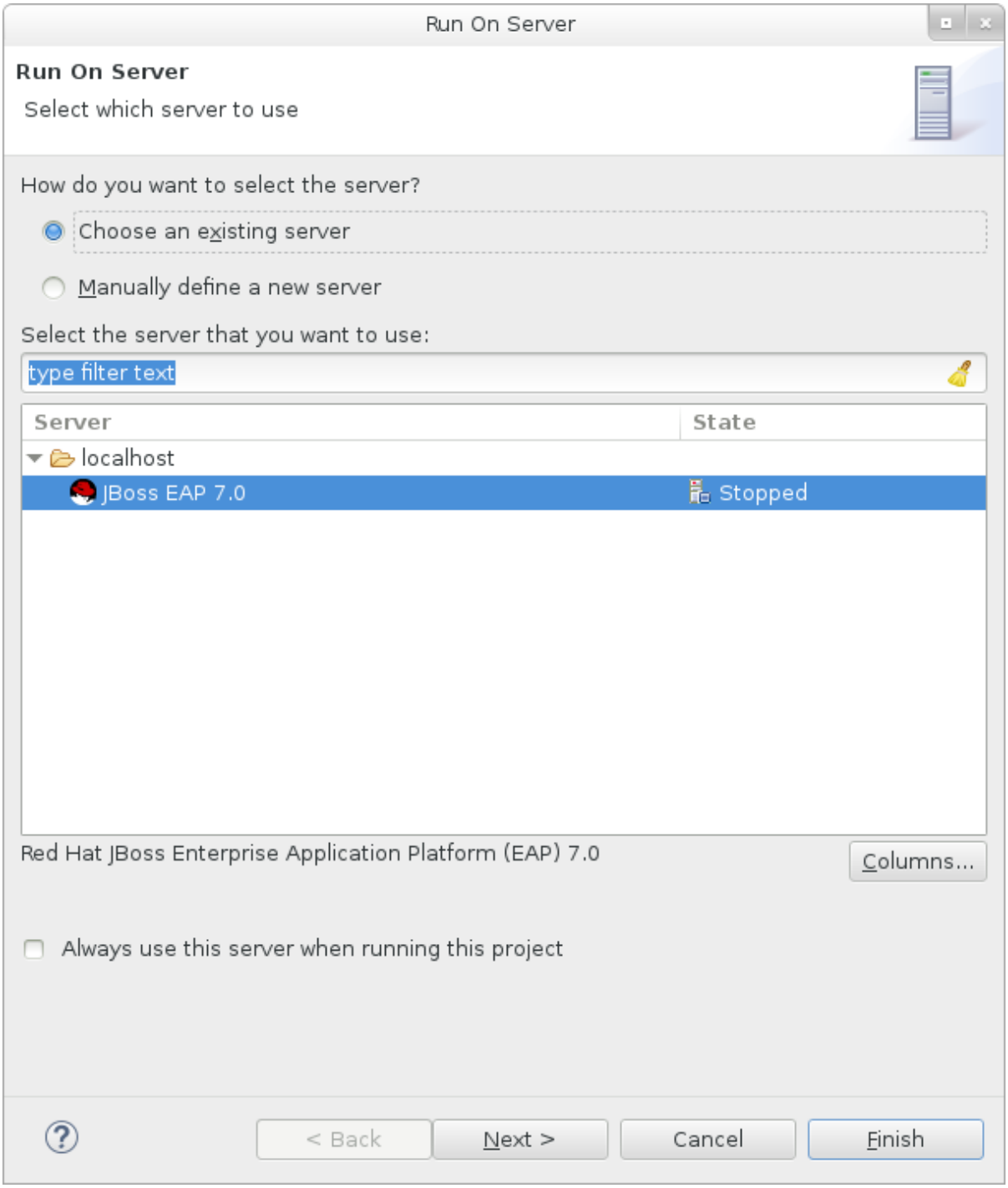
2. **Project Explorer** タブの **jboss-helloworld** プロジェクトを右クリックし、**Run As** → **Run on Server** と選択します。

図1.9 Run As - Run on Server



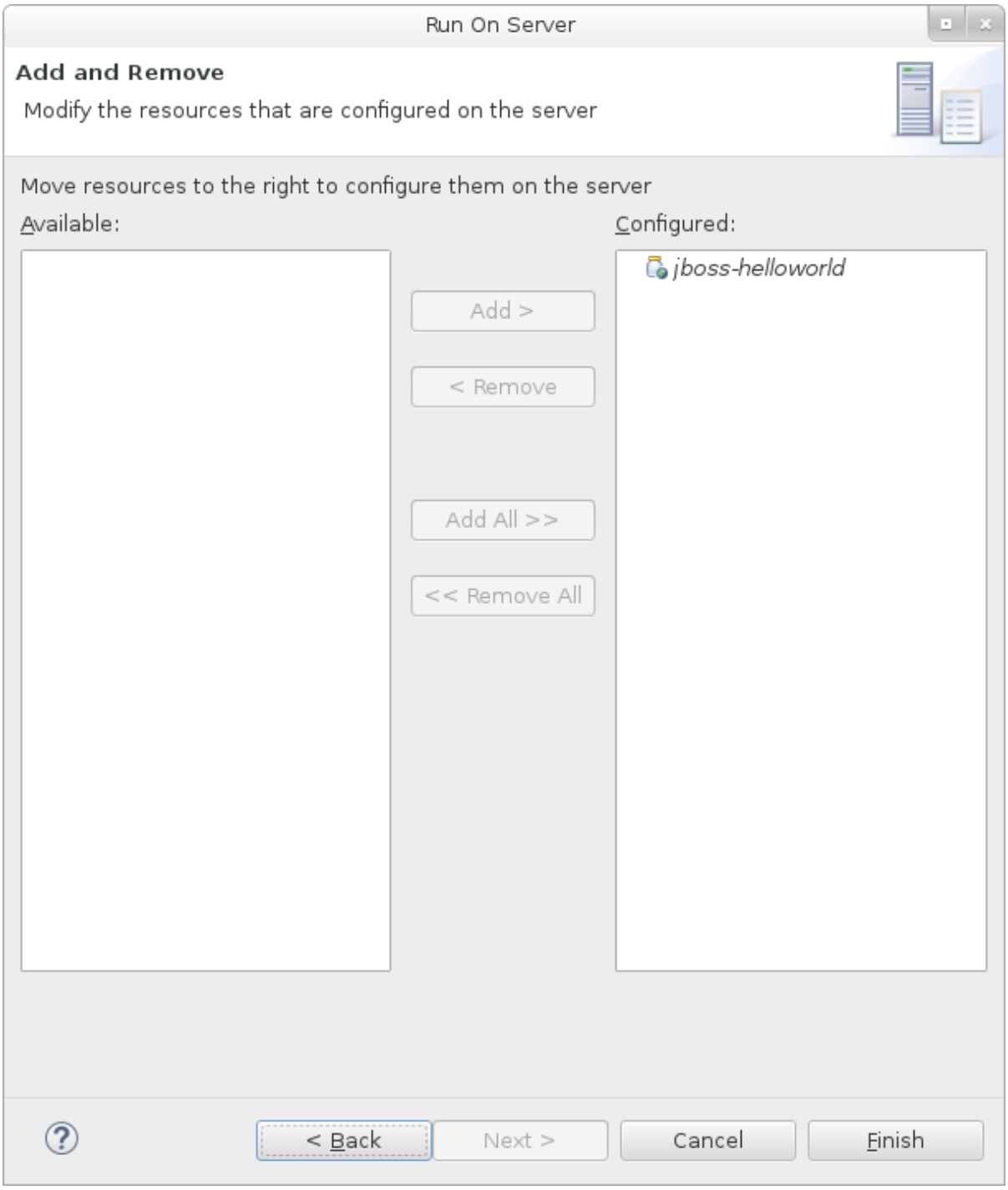
3. リストから **JBoss EAP 7.0** を選択し、**Next** をクリックします。

図1.10 Run on Server



4. **jboss-helloworld** クイックスタートはすでにリストされ、サーバー上で設定できる状態です。**Finish** をクリックしてクイックスタートをデプロイします。

図1.11 サーバーで設定されたリソースの変更



5. 結果を検証します。

- **Server** タブで、**JBoss EAP 7.0** サーバーの状態が **Started** に変わります。
- **Console** タブに、JBoss EAP サーバーの起動と **helloworld** クイックスタートのデプロイメントに関するメッセージが表示されます。

```
WFLYUT0021: Registered web context: /jboss-helloworld
WFLYSRV0010: Deployed "jboss-helloworld.war" (runtime-name :
"jboss-helloworld.war")
```

- **helloworld** アプリケーションは http://localhost:8080/jboss-helloworld で使用でき、**Hello World!** というテキストが表示されます。

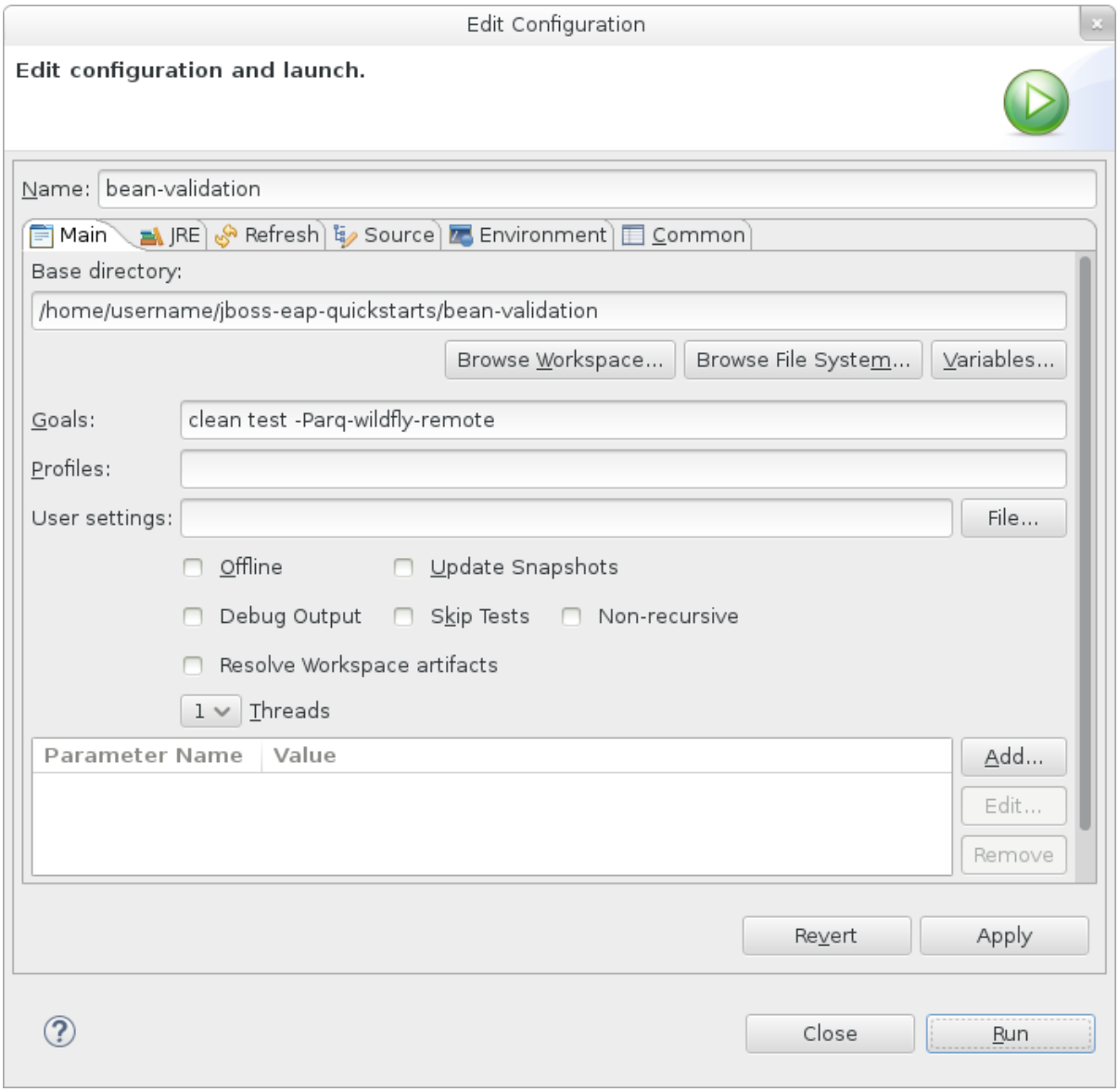
bean-validation クイックスタートの実行

bean-validation などの一部のクイックスタートは、機能のデモを行うためにユーザーインターフェースレイヤーの代わりに Arquillian テストを提供します。

1. **bean-validation** クイックスタートを JBoss Developer Studio にインポートします。
2. **Servers** サーバータブでサーバーを右クリックし、**Start** を選択して JBoss EAP サーバーを起動します。**Servers** タブが表示されない場合や、サーバーが定義されていない場合は、JBoss EAP サーバーを Red Hat JBoss Developer Studio へ追加してください。
3. **Project Explorer** タブで **jboss-bean-validation** プロジェクトを右クリックし、**Run As** → **Maven Build** と選択します。
4. 以下の内容を **Goals** 入力フィールドに入力し、**Run** を実行します。

```
clean test -Parq-wildfly-remote
```

図1.12 設定の編集



5. 結果を検証します。
   Console タブに **bean-validation** Arquillian テストの結果が表示されます。

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running
org.jboss.as.quickstarts.bean_validation.test.MemberValidationTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
2.189 sec

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------
--------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
--------------
```

1.4.2.3. コマンドラインでのクイックスタートの実行

Maven を使用すると、コマンドラインから簡単にクイックスタートをビルドおよびデプロイできます。Maven がインストールされていない場合は Apache Maven プロジェクトを参照し、ダウンロードとインストールを行ってください。

README.md ファイルは、システム要件、Maven の設定、ユーザーの追加、およびクイックスタートの実行に関する一般的な情報が含まれるクイックスタートのルートディレクトリーにあります。

各クイックスタートには、クイックスタートを実行するための特定の手順と Maven コマンドを提供する独自の **README.md** ファイルも含まれます。

コマンドラインでの helloworld クイックスタートの実行

1. **helloworld** クイックスタートのルートディレクトリーにある **README.md** ファイルを確認します。
2. JBoss EAP サーバーを起動します。

   ```
   $ EAP_HOME/bin/standalone.sh
   ```

3. **helloworld** クイックスタートディレクトリーへ移動します。
4. クイックスタートの **README.md** ファイルにある Maven コマンドを使用して、クイックスタートをビルドおよびデプロイします。

   ```
   mvn clean install wildfly:deploy
   ```

5. **helloworld** アプリケーションは http://localhost:8080/jboss-helloworld で使用でき、**Hello World!** というテキストが表示されます。

1.4.3. クイックスタートチュートリアルの確認

1.4.3.1. helloworld クイックスタート

helloworld クイックスタートは JBoss EAP に単純なサーブレットをデプロイする方法を示します。ビジネスロジックは CDI (Contexts and Dependency Injection: コンテキストと依存関係の挿入) Bean として提供されるサービスにカプセル化され、サーブレットに挿入されます。このクイックスタートに基づいて、サーバーを適切に設定および起動することができます。

コマンドラインを使用してこのクイックスタートをビルドしデプロイする手順の詳細については、**helloworld** クイックスタートディレクトリーのルートにある **README.html** ファイルを参照してください。このトピックでは、Red Hat JBoss Developer Studio を使用してクイックスタートを実行する方法を説明します (Red Hat JBoss Developer Studio がインストールされ、Maven が設定された状態で **helloworld** クイックスタートがインポートされ、正常に実行されたことを前提とします)。

前提条件

- Red Hat JBoss Developer Studio をインストールします。
- JBoss Developer Studio でのクイックスタートを実行する手順に従います。
- Web ブラウザーを開いて http://localhost:8080/jboss-helloworld にあるアプリケーションにアクセスし、**helloworld** クイックスタートが JBoss EAP に正しくデプロイされたことを確認します。

ディレクトリー構造の確認
helloworld クイックスタートのコードは **QUICKSTART_HOME/helloworld** ディレクトリーにあります。**helloworld** クイックスタートはサーブレットと CDI Bean によって構成されます。また、バージョン番号が 1.1 であり、**bean-discovery-mode** が **all** であるアプリケーションの **WEB-INF** ディレクトリーに **beans.xml** ファイルが含まれます。このマーカーファイルにより、WAR が Bean アーカイブとして識別され、JBoss EAP がこのアプリケーションで Bean を検索し、CDI をアクティベートするよう指示されます。

src/main/webapp/ ディレクトリーには、クイックスタートのファイルが含まれます。このサンプルのすべての設定ファイルは、**src/main/webapp/** 内の **WEB-INF/** ディレクトリー (**beans.xml** ファイルが含まれる) にあります。また、**src/main/webapp/** ディレクトリーには、http://localhost:8080/jboss-helloworld/HelloWorld にあるサーブレットにユーザーのブラウザーをリダイレクトするために単純なメタ更新を使用する **index.html** ファイルも含まれます。クイックスタートは **web.xml** ファイルを必要としません。

コードの確認
パッケージの宣言とインポートはこれらのリストには含まれていません。完全なリストはクイックスタートのソースコードにあります。

1. **HelloWorldServlet** コードを確認します。
   HelloWorldServlet.java ファイルは **src/main/java/org/jboss/as/quickstarts/helloworld/** ディレクトリーにあります。このサーブレットが情報をブラウザーに送ります。

   HelloWorldServlet クラスコードサンプル

```
42 @SuppressWarnings("serial")
43 @WebServlet("/HelloWorld")
44 public class HelloWorldServlet extends HttpServlet {
45
46     static String PAGE_HEADER = "<html><head>
<title>helloworld</title></head><body>";
47
48     static String PAGE_FOOTER = "</body></html>";
49
```

```
50     @Inject
51    HelloService helloService;
52
53     @Override
54     protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException {
55         resp.setContentType("text/html");
56         PrintWriter writer = resp.getWriter();
57         writer.println(PAGE_HEADER);
58         writer.println("<h1>" +
helloService.createHelloMessage("World") + "</h1>");
59         writer.println(PAGE_FOOTER);
60         writer.close();
61     }
62
63 }
```

表1.1 HelloWorldServlet の詳細

行	注記
43	必要な作業は **@WebServlet** アノテーションを追加し、サーブレットにアクセスするために使用する URL にマッピングを提供するだけです。
46～48	各 Web ページには適切な形式の HTML が必要になります。本クイックスタートは静的な文字列を使用して最低限のヘッダーとフッターの出力を書き出します。
50～51	これらの行は、実際のメッセージを生成する HelloService CDI Bean を挿入します。HelloService の API を変更しない限り、ビューレイヤーを変更せずに HelloService の実装を後で変更することが可能です。
58	この行はサービスを呼び出し、「Hello World」というメッセージを生成して HTTP 要求へ書き出します。

2. **HelloService** コードを確認します。
HelloService.java ファイルは **src/main/java/org/jboss/as/quickstarts/helloworld/** ディレクトリーにあります。このサービスは単にメッセージを返します。XML やアノテーションの登録は必要ありません。

HelloService クラスコードサンプル

```
public class HelloService {

    String createHelloMessage(String name) {
        return "Hello " + name + "!";
    }
}
```

1.4.3.2. numberguess クイックスタート

numberguess クイックスタートは単純な非永続アプリケーションを作成し、JBoss EAP にデプロイする方法を示します。情報は JSF ビューを使用して表示され、ビジネスロジックは 2 つの CDI Bean にカプセル化されます。**numberguess** クイックスタートでは 1 から 100 までの数字を当てるチャンスが 10 回与えられます。数字を選択した後、その数字が正解の数字よりも大きいかまたは小さいかが表示されます。

numberguess クイックスタートのコードは **QUICKSTART_HOME/numberguess** ディレクトリーにあります (**QUICKSTART_HOME** は JBoss EAP クイックスタートをダウンロードし、展開したディレクトリー)。**numberguess** クイックスタートは複数の Bean、設定ファイル、および Facelets (JSF) ビューによって構成され、 WAR モジュールとしてパッケージ化されます。

コマンドラインを使用してこのクイックスタートをビルドしデプロイする手順の詳細については、**numberguess** クイックスタートディレクトリーのルートにある **README.html** ファイルを参照してください。以下の例では、Red Hat JBoss Developer Studio を使用してクイックスタートを実行します。

前提条件

- Red Hat JBoss Developer Studio をインストールします。
- Red Hat JBoss Developer Studio でのクイックスタートを実行する手順に従います (手順で **helloworld** を **numberguess** に置き換えます)。
- Web ブラウザーを開いて http://localhost:8080/jboss-numberguess にあるアプリケーションにアクセスし、**numberguess** クイックスタートが JBoss EAP に正しくデプロイされたことを確認します。

設定ファイルの確認
この例のすべての設定ファイルは、クイックスタートの **QUICKSTART_HOME/numberguess/src/main/webapp/WEB-INF/** ディレクトリーにあります。

1. **faces-config.xml** ファイルを確認します。
   本クイックスタートは **faces-config.xml** ファイル名の JSF 2.2 バージョンを使用します。Facelets の標準的なバージョンが JSF 2.2 のデフォルトのビューハンドラーであるため、設定は必要ありません。このファイルはルート要素のみで構成され、JSF をアプリケーションで有効にする必要があることを示すマーカーファイルにすぎません。

```
<faces-config version="2.2"
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">

</faces-config>
```

2. **beans.xml** ファイルを確認します。
   beans.xml ファイルには、1.1 のバージョン番号と **all** の **bean-discovery-mode** が含まれます。このファイルは、WAR を Bean アーカイブとして識別し、JBoss EAP がこのアプリケーションで Bean を検索し、CDI をアクティベートするよう指示するマーカーファイルです。

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

```
        http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
    bean-discovery-mode="all">
</beans>
```

注記

このクイックスタートは **web.xml** ファイルを必要としません。

1.4.3.2.1. JSF コードの確認

JSF はソースファイルに **.xhtml** ファイル拡張子を使用しますが、レンダリングされたビューは **.jsf** 拡張子で提供されます。**home.xhtml** ファイルは **src/main/webapp/** ディレクトリーにあります。

JSF ソースコード

```
19<html xmlns="http://www.w3.org/1999/xhtml"
20 xmlns:ui="http://java.sun.com/jsf/facelets"
21 xmlns:h="http://java.sun.com/jsf/html"
22 xmlns:f="http://java.sun.com/jsf/core">
23
24 <head>
25 <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"
/>
26 <title>Numberguess</title>
27 </head>
28
29 <body>
30 <div id="content">
31  <h1>Guess a number...</h1>
32  <h:form id="numberGuess">
33
34  <!-- Feedback for the user on their guess -->
35 <div style="color: red">
36  <h:messages id="messages" globalOnly="false" />
37  <h:outputText id="Higher" value="Higher!"
38     rendered="#{game.number gt game.guess and game.guess ne 0}" />
39  <h:outputText id="Lower" value="Lower!"
40     rendered="#{game.number lt game.guess and game.guess ne 0}" />
41 </div>
42
43 <!-- Instructions for the user -->
44 <div>
45 I'm thinking of a number between <span
46 id="numberGuess:smallest">#{game.smallest}</span> and <span
47 id="numberGuess:biggest">#{game.biggest}</span>. You have
48 #{game.remainingGuesses} guesses remaining.
49 </div>
50
51 <!-- Input box for the users guess, plus a button to submit, and reset
-->
52 <!-- These are bound using EL to our CDI beans -->
53 <div>
54 Your guess:
55 <h:inputText id="inputGuess" value="#{game.guess}"
```

```
 required="true" size="3"
 disabled="#{game.number eq game.guess}"
 validator="#{game.validateNumberRange}" />
 <h:commandButton id="guessButton" value="Guess"
  action="#{game.check}"
  disabled="#{game.number eq game.guess}" />
</div>
<div>
<h:commandButton id="restartButton" value="Reset"
action="#{game.reset}" immediate="true" />
</div>
</h:form>

</div>

<br style="clear: both" />

</body>
</html>
```

以下の行番号は、JBoss Developer Studio でファイルを表示するときに示されるものに対応します。

表1.2 JSF の詳細

行	注記
36〜40	これらはユーザーに送信できるメッセージ、「Higher」(より大きい) と「Lower」(より小さい) です。
45〜48	ユーザーが数を選択するごとに数字の範囲が狭まります。有効な数の範囲が分かるようにこの文章は変更されます。
55〜58	この入力フィールドは値式を使用して Bean プロパティーにバインドされます。
58	ユーザーが誤って範囲外の数字を入力しないようにバリデーターのバインディングが使用されます。バリデーターがないと、ユーザーが範囲外の数字を使用する可能性があります。
59〜61	ユーザーの選択した数字をサーバーに送る方法がなければなりません。ここでは、Bean 上のアクションメソッドをバインドします。

1.4.3.2.2. クラスファイルの確認

numberguess クイックスタートのソースファイルはすべて **QUICKSTART_HOME/numberguess/src/main/java/org/jboss/as/quickstarts/numberguess/** ディレクトリーにあります。パッケージの宣言とインポートはリストには含まれていません。完全なリストはクイックスタートのソースコードにあります。

1. **Random.java** 修飾子コードの検証
   型に基づき挿入の対象となる 2 つの Bean を明確に区別するために修飾子が使用されます。修飾子の詳細については、修飾子を使用したあいまいな挿入の解決を参照してください。 ランダムな数字を挿入するには **@Random** 修飾子が使用されます。

   ■

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

}
```

2. **MaxNumber.java** 修飾子コードの検証
   @MaxNumber qualifier は最大許可数の挿入に使用されます。

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

3. **Generator.java** コードの検証
   Generator クラスは、producer メソッドを介して乱数を作成し、producer メソッドを介して最大可能数を公開します。このクラスはアプリケーションスコープであるため、毎回異なる乱数になることはありません。

```
@SuppressWarnings("serial")
@ApplicationScoped
public class Generator implements Serializable {

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

    private int maxNumber = 100;

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

    @Produces
    @Random
    int next() {
        // a number between 1 and 100
        return getRandom().nextInt(maxNumber - 1) + 1;
    }

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

4. **Game.java** コードの検証
   セッションスコープのクラス **Game** は、アプリケーションのプライマリーエントリーポイントであり、ゲームの設定や再設定、ユーザーが選択する数字のキャプチャーや検証、**FacesMessage** によるユーザーへのフィードバック提供を行います。コンストラクト後の

lifecycle メソッドを使用し、**@Random Instance**`<Integer>` Bean から乱数を取得することによりゲームを初期化します。

このクラスの **@Named** アノテーションを見てください。このアノテーションは式言語 (EL) を使用して Bean が JSF ビューにアクセスできるようにしたい場合のみ必要です。この場合 **#{game}** が EL になります。

```
@SuppressWarnings("serial")
@Named
@SessionScoped
public class Game implements Serializable {

    /**
     * The number that the user needs to guess
     */
    private int number;

    /**
     * The users latest guess
     */
    private int guess;

    /**
     * The smallest number guessed so far (so we can track the valid
guess range).
     */
    private int smallest;

    /**
     * The largest number guessed so far
     */
    private int biggest;

    /**
     * The number of guesses remaining
     */
    private int remainingGuesses;

    /**
     * The maximum number we should ask them to guess
     */
    @Inject
    @MaxNumber
    private int maxNumber;

    /**
     * The random number to guess
     */
    @Inject
    @Random
    Instance<Integer> randomNumber;

    public Game() {
    }

    public int getNumber() {
```

```
        return number;
    }

    public int getGuess() {
        return guess;
    }

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

    public int getSmallest() {
        return smallest;
    }

    public int getBiggest() {
        return biggest;
    }

    public int getRemainingGuesses() {
        return remainingGuesses;
    }

    /**
     * Check whether the current guess is correct, and update the
biggest/smallest guesses as needed. Give feedback to the user
     * if they are correct.
     */
    public void check() {
        if (guess > number) {
            biggest = guess - 1;
        } else if (guess < number) {
            smallest = guess + 1;
        } else if (guess == number) {
            FacesContext.getCurrentInstance().addMessage(null, new
FacesMessage("Correct!"));
        }
        remainingGuesses--;
    }

    /**
     * Reset the game, by putting all values back to their defaults,
and getting a new random number. We also call this method
     * when the user starts playing for the first time using
{@linkplain PostConstruct @PostConstruct} to set the initial
     * values.
     */
    @PostConstruct
    public void reset() {
        this.smallest = 0;
        this.guess = 0;
        this.remainingGuesses = 10;
        this.biggest = maxNumber;
        this.number = randomNumber.get();
    }
```

```
    /**
     * A JSF validation method which checks whether the guess is
valid. It might not be valid because there are no guesses left,
     * or because the guess is not in range.
     *
     */
    public void validateNumberRange(FacesContext context,
UIComponent toValidate, Object value) {
        if (remainingGuesses <= 0) {
            FacesMessage message = new FacesMessage("No guesses
left!");
            context.addMessage(toValidate.getClientId(context),
message);
            ((UIInput) toValidate).setValid(false);
            return;
        }
        int input = (Integer) value;

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

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

1.5. デフォルトの WELCOME WEB アプリケーションの設定

JBoss EAP には、デフォルトでポート 8080 のルートコンテキストで表示される **Welcome** アプリケーションが含まれます。

このデフォルトの **Welcome** アプリケーションは、独自の Web アプリケーションで置き換えることができます。これは、以下の 2 つのいずれかの方法で設定できます。

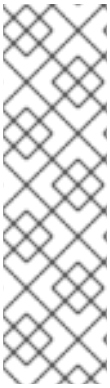
- **welcome-content** ファイルハンドラーを変更する
- **default-web-module** を変更する

Welcome コンテンツを無効にすることもできます。

welcome-content ファイルハンドラーの変更

新しいデプロイメントを参照する、既存の **welcome-content** ファイルハンドラーのパスを変更します。

```
/subsystem=undertow/configuration=handler/file=welcome-content:write-
attribute(name=path,value="/path/to/content")
```



注記

または、サーバーのルートにより使用される異なるファイルハンドラーを作成することもできます。

```
/subsystem=undertow/configuration=handler/file=NEW_FILE_HANDLER
:add(path="/path/to/content")
/subsystem=undertow/server=default-server/host=default-
host/location=\/:write-
attribute(name=handler,value=NEW_FILE_HANDLER)
```

変更を反映するためにサーバーをリロードします。

```
reload
```

default-web-module の変更
デプロイされた Web アプリケーションをサーバーのルートにマップします。

```
/subsystem=undertow/server=default-server/host=default-host:write-
attribute(name=default-web-module,value=hello.war)
```

変更を反映するためにサーバーをリロードします。

```
reload
```

デフォルトの Welcome Web アプリケーションの無効化
`default-host` の **`location`** エントリー (`/`) を削除して welcome アプリケーションを無効にします。

```
/subsystem=undertow/server=default-server/host=default-
host/location=\/:remove
```

変更を反映するためにサーバーをリロードします。

```
reload
```

第2章 JBOSS EAP での MAVEN の使用

2.1. MAVEN について

2.1.1. Maven リポジトリー

Apache Maven は、ソフトウェアプロジェクトの作成、管理、および構築を行う Java アプリケーションの開発で使用される分散型ビルド自動化ツールです。Maven は Project Object Model (POM) と呼ばれる標準の設定ファイルを利用して、プロジェクトの定義や構築プロセスの管理を行います。POM はモジュールやコンポーネントの依存関係、ビルドの順番、結果となるプロジェクトパッケージングのターゲットを定義し、XML ファイルを使用して出力します。この結果、プロジェクトが適切かつ統一された状態で構築されるようになります。

Maven は、リポジトリーを使用してアーカイブを行います。Maven リポジトリーには Java ライブラリー、プラグイン、その他のビルドアーティファクトが格納されています。デフォルトのパブリックリポジトリーは Maven 2 Central Repository ですが、複数の開発チームの間で共通のアーティファクトを共有する目的で、社内のプライベートおよび内部リポジトリーとすることが可能です。また、サードパーティーのリポジトリーもあります。JBoss EAP には、Java EE 開発者が JBoss EAP 6 でアプリケーションを構築する際に利用する要件の多くが含まれる Maven リポジトリーが含まれます。このリポジトリーを使用するようプロジェクトを設定するには、JBoss EAP Maven リポジトリーの設定を参照してください。

Maven の詳細については、Welcome to Apache Maven を参照してください。

Maven リポジトリーの詳細については、Apache Maven Project - Introduction to Repositories を参照してください。

2.1.2. Maven POM ファイル

プロジェクトオブジェクトモデル (POM) ファイルはプロジェクトをビルドするために Maven で使用する設定ファイルです。POM ファイルは XML のファイルであり、プロジェクトの情報やビルド方法を含みます。これには、ソース、テスト、およびターゲットのディレクトリーの場所、プロジェクトの依存関係、プラグインリポジトリー、実行できるゴールが含まれます。また、バージョン、説明、開発者、メーリングリスト、ライセンスなどのプロジェクトに関する追加情報も含まれます。**pom.xml** ファイルでは一部の設定オプションを設定する必要があり、他のすべてのオプションはデフォルト値に設定されます。

pom.xml ファイルのスキーマは http://maven.apache.org/maven-v4_0_0.xsd にあります。

POM ファイルの詳細については、Apache Maven Project POM Reference を参照してください。

Maven POM ファイルの最低要件
pom.xml ファイルの最低要件は次のとおりです。

- プロジェクトルート
- modelVersion
- groupId - プロジェクトのグループの ID
- artifactId - アーティファクト (プロジェクト) の ID
- version - 指定したグループ下のアーティファクトのバージョン

例: サンプル pom.xml ファイル

基本的な **pom.xml** ファイルは次のようになります。

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jboss.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

2.1.3. Maven 設定ファイル

Maven の **settings.xml** ファイルには Maven に関するユーザー固有の設定情報が含まれています。開発者の ID、プロキシ情報、ローカルリポジトリーの場所など、 **pom.xml** ファイルで配布されてはならないユーザー固有の情報が含まれています。

settings.xml が存在する場所は 2 つあります。

- **Maven インストール:** 設定ファイルは **$M2_HOME/conf/** ディレクトリーにあります。これらの設定は **global** 設定と呼ばれます。デフォルトの Maven 設定ファイルはコピー可能なテンプレートであり、これを基にユーザー設定ファイルを設定することが可能です。
- **ユーザーのインストール:** 設定ファイルは **${user.home}/.m2/** ディレクトリーにあります。Maven とユーザーの **settings.xml** ファイルが存在する場合、内容はマージされます。重複する内容がある場合は、ユーザーの **settings.xml** ファイルが優先されます。

例: Maven 設定ファイル

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <profiles>
    <!-- Configure the JBoss EAP Maven repository -->
    <profile>
      <id>jboss-eap-maven-repository</id>
      <repositories>
        <repository>
          <id>jboss-eap</id>
          <url>file:///path/to/repo/jboss-eap-7.0.0.GA-maven-
repository/maven-repository</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-eap-maven-plugin-repository</id>
          <url>file:///path/to/repo/jboss-eap-7.0.0.GA-maven-
repository/maven-repository</url>
          <releases>
```

```
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <activeProfiles>
    <!-- Optionally, make the repository active by default -->
    <activeProfile>jboss-eap-maven-repository</activeProfile>
  </activeProfiles>
</settings>
```

settings.xml ファイルのスキーマは http://maven.apache.org/xsd/settings-1.0.0.xsd にあります。

2.1.4. Maven リポジトリーマネージャー

リポジトリーマネージャーは、 Maven リポジトリーを容易に管理できるようにするツールです。リポジトリーマネージャーには、次のような利点があります。

- ユーザーの組織のリポジトリーとリモート Maven リポジトリーとの間のプロキシを設定する機能を提供します。これには、デプロイメントの高速化や効率化、Maven によるダウンロード対象を制御するレベルの向上など、さまざまな利点があります。
- 独自に生成したアーティファクトのデプロイ先を提供し、組織内の異なる開発チーム間におけるコラボレーションを可能にします。

Maven リポジトリーマネージャーの詳細については、Best Practice - Using a Repository Manager を参照してください。

一般的に使用される Maven リポジトリーマネージャー

Sonatype Nexus

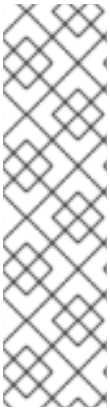
Nexus の詳細については、Sonatype Nexus documentation を参照してください。

Artifactory

Artifactory の詳細については、JFrog Artifactory ドキュメンテーションを参照してください。

Apache Archiva

Apache Archiva の詳細については、Apache Archiva: The Build Artifact Repository Manager を参照してください。



注記

通常リポジトリーマネージャーが使用されるエンタープライズ環境では、Maven は、このマネージャーを使用してすべてのプロジェクトに対してすべてのアーティファクトを問い合わせる必要があります。Maven は、宣言されたすべてのリポジトリーを使用して不明なアーティファクトを見つけるため、探しているものが見つからない場合に、**central**リポジトリー (組み込みの親 POM で定義されます) で検索を試行します。この **central** の場所をオーバーライドするには、**central** で定義を追加してデフォルトの **central** リポジトリーがリポジトリーマネージャーになるようにします。これは、確立されたプロジェクトには適切ですが、クリーンな、または「新しい」プロジェクトの場合は、**周期的**な依存関係が作成されるため、問題が発生します。

2.2. MAVEN と JBOSS EAP MAVEN リポジトリーのインストール

2.2.1. Maven のダウンロードとインストール

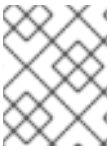
Maven コマンドラインを使用してアプリケーションをビルドし、JBoss EAP にデプロイする場合は、Maven をダウンロードし、インストールする必要があります。Red Hat JBoss Developer Studio を使用してアプリケーションをビルドおよびデプロイする場合、Maven は Red Hat JBoss Developer Studio で配布されるため、この手順を省略できます。

1. Apache Maven Project - Download Maven にアクセスし、ご使用のオペレーティングシステムに対応する最新のディストリビューションをダウンロードします。
2. ご使用のオペレーシングシステムに Apache Maven をダウンロードおよびインストールする方法については、Maven のドキュメントを参照してください。

2.2.2. JBoss EAP の Maven リポジトリーのインストール

JBoss EAP Maven リポジトリーをインストールする方法は 3 つあります。

- JBoss EAP Maven レポジトリーはローカルファイルシステムにインストールできます。詳細な手順については、JBoss EAP Maven リポジトリーのローカルインストールを参照してください。
- JBoss EAP Maven レポジトリーは Apache Web Server にインストールできます。詳細については、Apache httpd で使用する JBoss EAP Maven リポジトリーのインストールを参照してください。
- JBoss EAP Maven リポジトリーは Nexus Maven リポジトリーマネージャーを使用してインストールできます。詳細については、Repository Management Using Nexus Maven Repository Manager を参照してください。



注記

JBoss EAP Maven リポジトリーはオンラインで利用したり、示された 3 つのいずれかの方法でダウンロードし、ローカルにインストールしたりできます。

2.2.3. JBoss EAP Maven リポジトリーのローカルインストール

この例では、ローカルのファイルシステムへ JBoss EAP Maven リポジトリーをダウンロードする手順を取り上げます。このオプションは簡単に設定できます。このオプションを使用すると、ローカルマシンですぐに使用を開始できます。開発で Maven の使用方法を理解するのに役に立ちますが、チームによる実稼働環境での使用には推奨されません。

JBoss EAP Maven リポジトリーをダウンロードし、ローカルファイルシステムにインストールするには、以下の手順に従ってください。

1. Web ブラウザーを開き、URL https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform にアクセスします。
2. リストに **Red Hat JBoss Enterprise Application Platform 7.0 Maven Repository** があることを確認します。
3. ダウンロード ボタンをクリックし、リポジトリーが含まれる **`.zip`** ファイルをダウンロードします。

4. ローカルファイルシステム上の Zip 形式のファイルを希望のディレクトリーで展開します。これにより、**maven-repository/** という名前のサブディレクトリーに Maven レポジトリーが含まれる新しい **jboss-eap-7.0.0.GA-maven-repository/** ディレクトリーが作成されます。

重要

古いローカルリポジトリーを引き続き使用する場合は、そのリポジトリーを Maven **settings.xml** 設定ファイルで個別に設定する必要があります。各ローカルリポジトリーは、独自の **<repository>** タグ内で設定する必要があります。

重要

新しい Maven リポジトリーをダウンロードする場合は、使用する前に、**.m2/** ディレクトリーにあるキャッシュされた **repository/** サブディレクトリーを削除してください。

2.2.4. Apache httpd で使用する JBoss EAP Maven レポジトリーのインストール

この例では、Apache httpd で使用する JBoss EAP Maven リポジトリーをダウンロードする手順を示します。Web サーバーにアクセスできる開発者は Maven リポジトリーにもアクセスできるため、このオプションはマルチユーザーの開発環境や複数のチームにまたがる開発環境に適しています。

注記

最初に Apache httpd を設定する必要があります。手順については、Apache HTTP Server Project ドキュメンテーションを参照してください。

1. Web ブラウザーを開き、URL https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=appplatform にアクセスします。
2. リストに **Red Hat JBoss Enterprise Application Platform 7.0 Maven Repository** があることを確認します。
3. ダウンロード ボタンをクリックし、リポジトリーが含まれる **.zip** ファイルをダウンロードします。
4. Apache サーバー上で Web にアクセス可能なディレクトリーに Zip 形式のファイルを展開します。
5. 作成されたディレクトリーで読み取りアクセスとディレクトリーの閲覧を許可するよう Apache を設定します。
この設定により、マルチユーザー環境が Apache httpd 上で Maven リポジトリーにアクセスできるようになります。

2.3. MAVEN リポジトリーの使用

2.3.1. JBoss EAP Maven リポジトリーの設定

概要

プロジェクトで JBoss EAP Maven リポジトリーを使用するよう Maven に指示する方法は 2 つあります。

- リポジトリーを Maven グローバルまたはユーザー設定で設定します。
- リポジトリーをプロジェクトの POM ファイルで設定します。

Maven 設定を使用した JBoss EAP Maven リポジトリーの設定
これは推奨される方法です。リポジトリーマネージャーや共有サーバー上のリポジトリーを使用して Maven を設定すると、プロジェクトの制御および管理を行いやすくなります。また、代替のミラーを使用してプロジェクトファイルを変更せずにリポジトリーマネージャーに特定のリポジトリーのルックアップ要求をすべてリダイレクトすることも可能になります。ミラーの詳細については、http://maven.apache.org/guides/mini/guide-mirror-settings.html を参照してください。

プロジェクトの POM ファイルにリポジトリー設定が含まれていない場合、この設定方法はすべての Maven プロジェクトに対して適用されます。

この項では、Maven の設定方法について説明します。Maven インストールグローバル設定またはユーザーのインストール設定を指定できます。

Maven 設定ファイルの指定

1. 使用しているオペレーションシステムの Maven **settings.xml** ファイルを見つけます。通常、このファイルは **${user.home}/.m2/** ディレクトリーにあります。
   - Linux または Mac の場合、これは **~/.m2/** になります。
   - Windows の場合、これは **\Documents and Settings\.m2** または **\Users\.m2** になります。
2. **settings.xml** ファイルが見つからない場合は、**${user.home}/.m2/conf/** ディレクトリーの **settings.xml** ファイルを **${user.home}/.m2/** ディレクトリーへコピーします。
3. 以下の XML を **<profiles>** element of the **settings.xml** ファイルにコピーします。JBoss EAP リポジトリーの URL を調べ、**JBOSS_EAP_REPOSITORY_URL** をその URL に置き換えます。

```
<!-- Configure the JBoss Enterprise Maven repository -->
<profile>
  <id>jboss-enterprise-maven-repository</id>
  <repositories>
    <repository>
      <id>jboss-enterprise-maven-repository</id>
      <url>JBOSS_EAP_REPOSITORY_URL</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-enterprise-maven-repository</id>
```

```
      <url>JBOSS_EAP_REPOSITORY_URL</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
```

以下に、オンラインの JBoss EAP Maven リポジトリーにアクセスする設定例を示します。

```
<!-- Configure the JBoss Enterprise Maven repository -->
<profile>
  <id>jboss-enterprise-maven-repository</id>
  <repositories>
    <repository>
      <id>jboss-enterprise-maven-repository</id>
      <url>https://maven.repository.redhat.com/ga/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-enterprise-maven-repository</id>
      <url>https://maven.repository.redhat.com/ga/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
```

4. 次の XML を **settings.xml** ファイルの **<activeProfiles>** 要素へコピーします。

```
<activeProfile>jboss-enterprise-maven-repository</activeProfile>
```

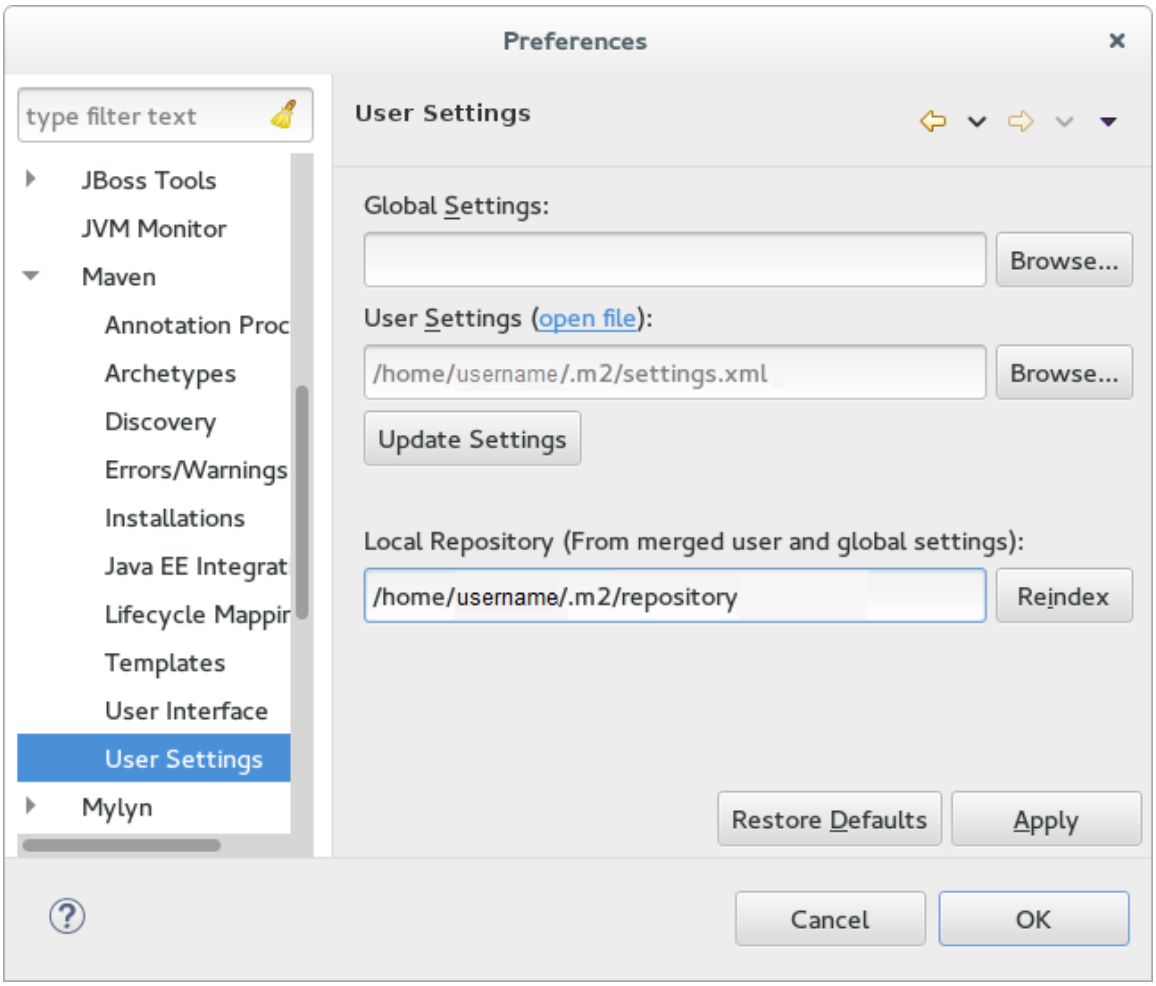
5. Red Hat JBoss Developer Studio の実行中に **settings.xml** ファイルを変更する場合は、ユーザー設定を更新する必要があります。

   a. メニューで **Window** → **Preferences** と選択します。

   b. **Preferences** ウィンドウで **Maven** を展開し、**User Settings** を設定します。

   c. **Update Settings** ボタンをクリックし、Red Hat JBoss Developer Studio で Maven のユーザー設定を更新します。

Maven ユーザー設定の更新 のスクリーンショット



重要

Maven リポジトリーに古いアーティファクトが含まれる場合は、プロジェクトをビルドまたはデプロイしたときに以下のいずれかの Maven エラーメッセージが表示されることがあります。

- Missing artifact ARTIFACT_NAME
- [ERROR] Failed to execute goal on project PROJECT_NAME; Could not resolve dependencies for PROJECT_NAME

この問題を解決するには、最新の Maven アーティファクトをダウンロードするためにローカルリポジトリーのキャッシュバージョンを削除します。キャッシュバージョンは `${user.home}/.m2/repository/` に存在します。

プロジェクト POM を使用した JBoss EAP Maven リポジトリーの設定

警告

この設定方法は、設定されたプロジェクトのグローバルおよびユーザー Maven 設定を上書きするため、回避する必要があります。

プロジェクト POM ファイルを使用してリポジトリーを設定する場合は、慎重に計画する必要があります。このような設定では、推移的に含まれた POM が問題になります。Maven は、外部リポジトリーで不明なアーティファクトを問い合わせ、これによりビルド処理に時間がかかるようになるためです。また、アーティファクトの抽出元を制御できなくなることもあります。

注記

リポジトリーの URL はリポジトリーの場所 (ファイルシステムまたは Web サーバー) によって異なります。リポジトリーのインストール方法については、JBoss EAP の Maven リポジトリーのインストールを参照してください。各インストールオプションの例は次のとおりです。

ファイルシステム

file:///path/to/repo/jboss-eap-maven-repository

Apache Web Server

http://intranet.acme.com/jboss-eap-maven-repository/

Nexus リポジトリーマネージャー

https://intranet.acme.com/nexus/content/repositories/jboss-eap-maven-repository

プロジェクトの POM ファイルの設定

1. テキストエディターでプロジェクトの **pom.xml** ファイルを開きます。
2. 次のリポジトリー設定を追加します。すでにファイルに **<repositories>** 設定が存在する場合は **<repository>** 要素を追加します。必ず **<url>** を実際のリポジトリーの場所に変更するようにしてください。

```
<repositories>
   <repository>
      <id>jboss-eap-repository-group</id>
      <name>JBoss EAP Maven Repository</name>
      <url>JBOSS_EAP_REPOSITORY_URL</url>
      <layout>default</layout>
      <releases>
         <enabled>true</enabled>
         <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
         <enabled>true</enabled>
         <updatePolicy>never</updatePolicy>
      </snapshots>
   </repository>
</repositories>
```

3. 次のプラグインリポジトリー設定を追加します。すでにファイルに **<pluginRepositories>** 設定が存在する場合は **<pluginRepository>** 要素を追加します。

```
<pluginRepositories>
   <pluginRepository>
      <id>jboss-eap-repository-group</id>
      <name>JBoss EAP Maven Repository</name>
      <url>JBOSS_EAP_REPOSITORY_URL</url>
```

```
      <releases>
         <enabled>true</enabled>
      </releases>
      <snapshots>
         <enabled>true</enabled>
      </snapshots>
   </pluginRepository>
</pluginRepositories>
```

JBoss EAP リポジトリーの URL の確認
リポジトリーの URL は、リポジトリーが存在する場所によって異なります。以下のいずれかのリポジトリーの場所を使用するよう Maven を設定できます。

- オンラインの JBoss EAP Maven リポジトリーを使用するには、URL https://maven.repository.redhat.com/ga/ を指定します。
- ローカルファイルシステムにインストールされた JBoss EAP Maven リポジトリーを使用するには、リポジトリーをダウンロードし、URL のローカルファイルパスを使用する必要があります (例: **file:///path/to/repo/jboss-eap-7.0-maven-repository/maven-repository/**)。
- Apache Web Server にリポジトリーをインストールする場合、リポジトリーの URL は **http://intranet.acme.com/jboss-eap-7.0-maven-repository/maven-repository/** のようになります。
- Nexus リポジトリーマネージャーを使用して JBoss EAP Maven リポジトリーをインストールする場合、URL は **https://intranet.acme.com/nexus/content/repositories/jboss-eap-7.0-maven-repository/maven-repository/** のようになります。

注記

リモートリポジトリーへのアクセスには、HTTP サーバーのリポジトリー用の **`http://`** やファイルサーバーのリポジトリー用の **`file://`** などの一般的なプロトコルが使用されます。

2.3.2. Red Hat JBoss Developer Studio で使用する Maven の設定

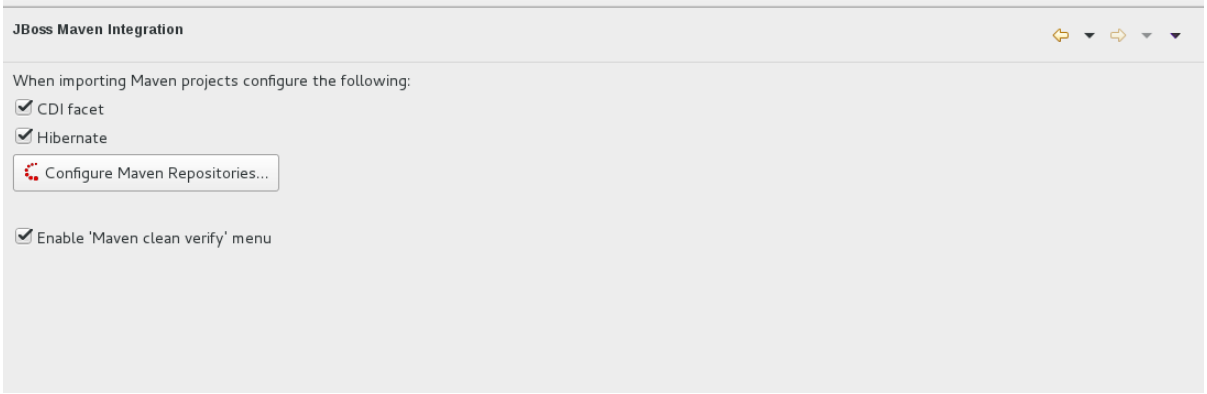
アプリケーションをビルドし、Red Hat JBoss Enterprise Application Platform にデプロイするのに必要なアーティファクトと依存関係は、パブリックリポジトリーでホストされます。アプリケーションをビルドするときにこのリポジトリーを使用するよう Maven を設定する必要があります。このトピックでは、Red Hat JBoss Developer Studio を使用してアプリケーションをビルドおよびデプロイする場合に Maven を設定する手順について説明します。

Maven は Red Hat JBoss Developer Studio で配布されるため、個別にインストールする必要がありません。ただし、JBoss EAP へのデプロイメントのために Java EE Web Project ウィザードで使用する Maven を設定する必要があります。以下の手順は、Red Hat JBoss Developer Studio 内から Maven 設定ファイルを編集して JBoss EAP で使用する Maven を設定する方法を示しています。

Red Hat JBoss Developer Studio での Maven の設定

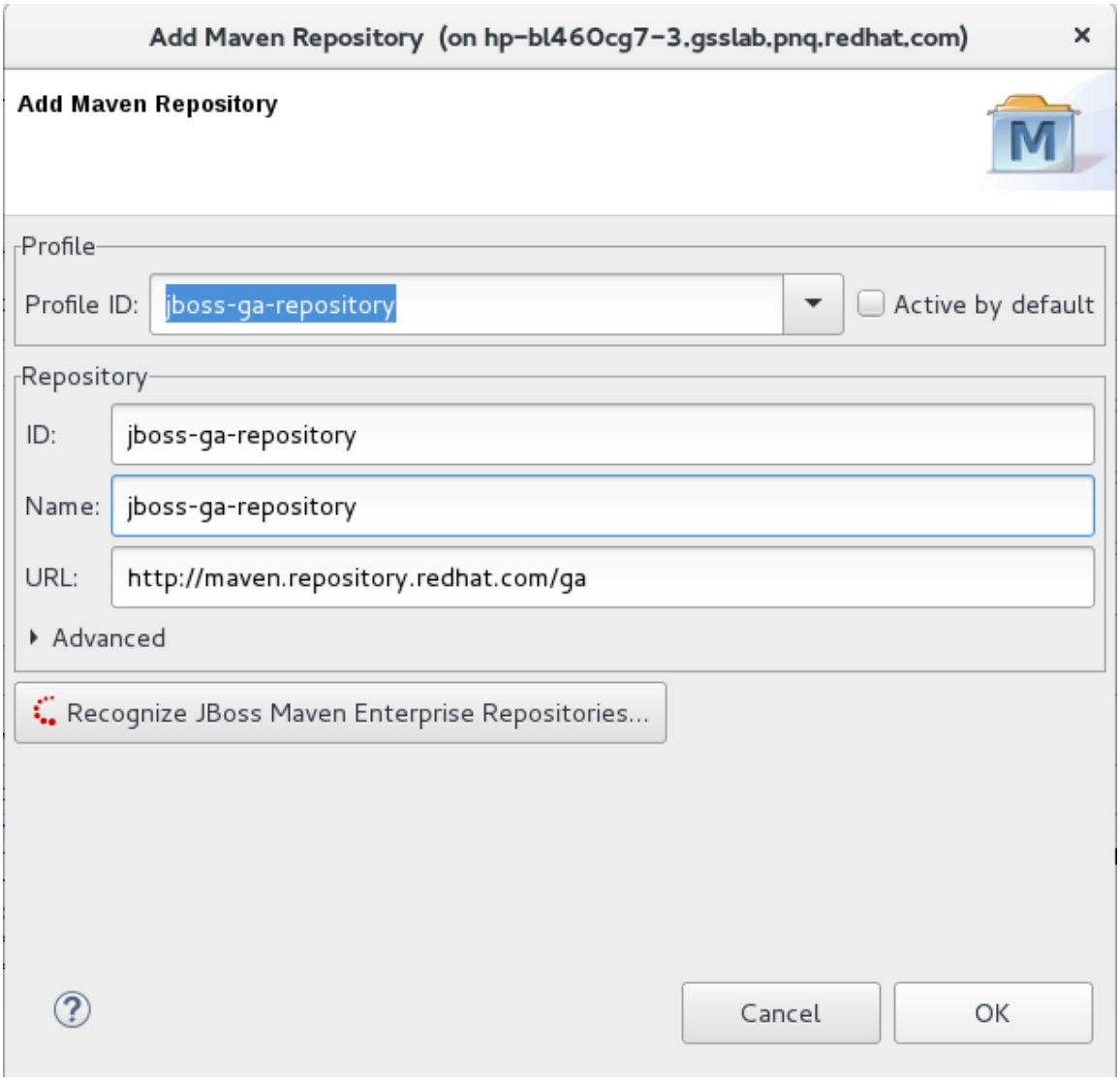
1. **`Window`** → **`Preferences`** をクリックし、**JBoss Tools** を展開して、**JBoss Maven Integration** を選択します。

   Preferences ウィンドウの JBoss Maven 統合ペイン



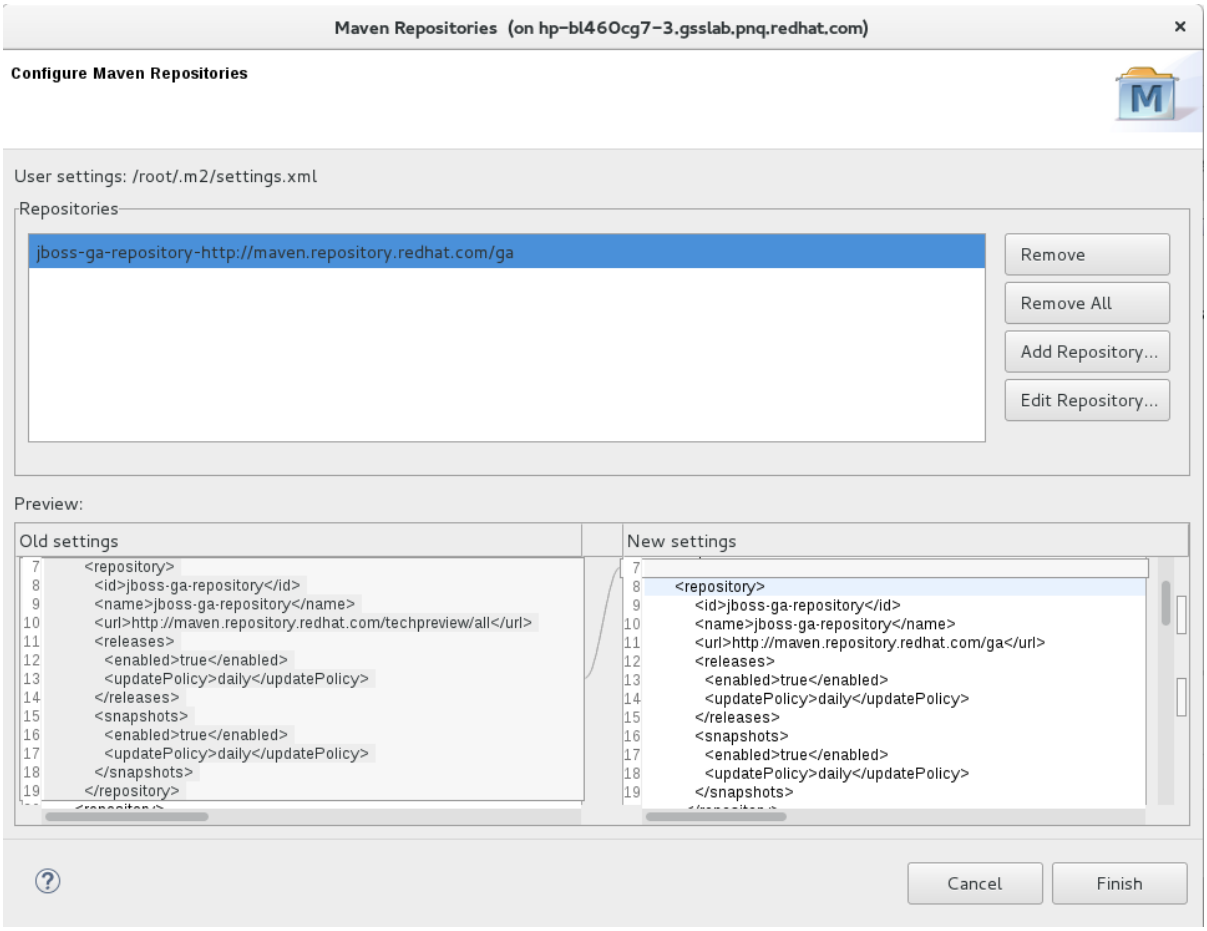
2. **Configure Maven Repositories** をクリックします。
3. **Add Repository** をクリックして JBoss Enterprise Maven リポジトリーを設定します。**Add Maven Repository** ダイアログで以下の手順を実行します。
   a. **Profile ID**、**Repository ID**、および **Repository Name** の値を **jboss-ga-repository** に設定します。
   b. **Repository URL** の値を **http://maven.repository.redhat.com/ga** に設定します。
   c. **Active by default** チェックボックスをクリックして Maven リポジトリーを有効にします。
   d. **OK** をクリックします。

Maven リポジトリーの追加



4. リポジトリーを確認して、**終了** をクリックします。

Maven リポジトリーの確認



5. **"Are you sure you want to update the file `MAVEN_HOME/settings.xml`?** というメッセージが表示されます。**Yes** をクリックして設定を更新します。**OK** をクリックしてダイアログを閉じます。

JBoss EAP Maven リポジトリーが Red Hat JBoss Developer Studio での使用向けに設定されます。

2.3.3. プロジェクト依存関係の管理

このトピックでは、Red Hat JBoss Enterprise Application Platform 向けの BOM (Bill of Materials) POM の使用方法について説明します。

BOM は、指定モジュールに対するすべてのランタイム依存関係のバージョンを指定する Maven **`pom.xml`** (POM) ファイルです。バージョン依存関係は、ファイルの依存関係管理セクションにリストされています。

プロジェクトは、**`groupId:artifactId:version`** (GAV) をプロジェクト **`pom.xml`** ファイルの依存関係管理セクションに追加し、**`<scope>import</scope>`** および **`<type>pom</type>`** 要素の値を指定して、BOM を使用します。

注記

多くの場合、プロジェクト POM ファイルの依存関係によって **`provided`** スコープが使用されます。これは、これらのクラスが実行時にアプリケーションサーバーによって提供され、ユーザーアプリケーションとともにパッケージ化する必要がないためです。

サポート対象の Maven アーティファクト
製品のビルドプロセスの一部として、JBoss EAP のすべてのランタイムコンポーネントは制御された環境でソースからビルドされます。これにより、バイナリーアーティファクトに悪意のあるコードが含ま

れないようにし、製品のライフサイクルが終了するまでサポートを提供できるようにします。これらのアーティファクトは、**1.0.0-redhat-1** のように使用される **-redhat** バージョン修飾子によって簡単に識別可能です。

サポートされるアーティファクトをビルド設定 **pom.xml** ファイルに追加すると、ローカルビルドおよびテスト向けの適切なバイナリーアーティファクトがビルドで使用されるようになります。**-redhat** バージョンのアーティファクトは、サポートされるパブリック API の一部とは限らず、今後の改訂で変更されることがあります。サポートされるパブリック API の詳細については、本リリースに同梱されている JavaDoc ドキュメントを参照してください。

たとえば、サポートされているバージョンの Hibernate を使用するには、ビルド設定に以下のようなコードを追加します。

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.0.1.Final-redhat-1</version>
  <scope>provided</scope>
</dependency>
```

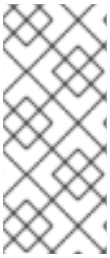
上記の例には、**<version/>** の値が含まれていることに注意してください。ただし、依存関係バージョンの設定には、Maven の依存関係管理を使用することが推奨されます。

依存関係管理
Maven には、ビルド全体で直接的および推移的な依存関係のバージョンを管理するメカニズムが含まれています。依存関係管理の使用に関する一般的な情報については、Apache Maven Project の Introduction to the Dependency Mechanism を参照してください。

サポートされる Red Hat の依存関係を 1 つ以上ビルドに直接使用しても、ビルドの推移的な依存関係がすべて Red Hat アーティファクトによって完全にサポートされるとは限りません。Maven のビルドでは、Maven の中央リポジトリーおよびその他の Maven リポジトリーから複数のアーティファクトソースの組み合わせが使用することが一般的です。

JBoss EAP Maven リポジトリーには、サポートされるすべての JBoss EAP バイナリーアーティファクトを指定する依存関係管理 BOM が含まれています。この BOM は、ビルドの直接的および推移的依存関係に対して、サポートされる JBoss EAP 依存関係の優先順位を決定するためにビルドで使用できます。つまり、推移的な依存関係が、サポートされる正しい依存関係バージョン (該当する場合) に対して管理されます。この BOM のバージョンは、JBoss EAP リリースのバージョンと一致します。

```
<dependencyManagement>
  <dependencies>
    ...
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>eap-runtime-artifacts</artifactId>
      <version>7.0.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```



注記

JBoss EAP 7 では、この BOM の名前が **eap6-supported-artifacts** から **eap-runtime-artifacts** に変更されました。この変更の目的は、この POM のアーティファクトが JBoss EAP ランタイムの一部であるが、必ずしもサポートされるパブリック API の一部ではないことを明確にすることです。一部の jar には、リリースごとに異なる場合がある内部 API と機能が含まれます。

JBoss EAP Java EE 仕様の BOM
`jboss-javaee-7.0` BOM には、JBoss EAP によって使用される Java EE 仕様の API JAR が含まれています。

この BOM をプロジェクトで使用するには、JSP のバージョンが含まれる GAV に対する依存関係と、アプリケーションのビルドおよびデプロイに必要なサーブレット API JAR を追加します。

以下の例では、**`1.0.3.Final-redhat-1`** バージョンの **`jboss-javaee-7.0`** BOM が使用されています。

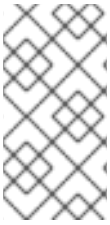
```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-javaee-7.0</artifactId>
      <version>1.0.3.Final-redhat-1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.jboss.spec.javax.servlet</groupId>
    <artifactId>jboss-servlet-api_3.1_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jboss.spec.javax.servlet.jsp</groupId>
    <artifactId>jboss-jsp-api_2.3_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>
```

JBoss EAP BOM とクイックスタート
クイックスタートは、Maven リポジトリーの主要なユースケース例を提供します。下表に、クイックスタートによって使用される Maven BOM を示します。

表2.1 クイックスタートによって使用される JBoss BOM

BOM アーティファクト ID	ユースケース
jboss-eap-javaee7	サポートされる JBoss EAP JavaEE 7 API と追加の JBoss EAP API jar

BOM アーティファクト ID	ユースケース
jboss-eap-javaee7-with-spring3	jboss-eap-javaee7 および推奨される Spring 3 バージョン
jboss-eap-javaee7-with-spring4	jboss-eap-javaee7 および推奨される Spring 4 バージョン
jjboss-eap-javaee7-with-tools	jboss-eap-javaee7 および Arquillian などの開発ツール



注記

ほとんどのユースケースに対して使用方法を単純にするために、JBoss EAP 6 のこれらの BOM は少ない数の BOM に統合されました。Hibernate、ロギング、トランザクション、メッセージング、および他のパブリック API jar は **jboss-javaee7-eap** に含まれるようになり、各ユースケースで個別の BOM が必要なくなりました。

以下の例では、**7.0.0.GA** バージョンの **jboss-eap-javaee7** BOM が使用されています。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-javaee7</artifactId>
      <version>7.0.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>
```

JBoss EAP クライアント BOM
クライアント BOM は、依存関係管理セクションを作成したり、依存関係を定義したりしません。クライアント BOM は他の BOM の集合体であり、リモートクライアントのユースケースに必要な依存関係のセットをパッケージ化するために使用されます。

wildfly-ejb-client-bom および **wildfly-jms-client-bom** の BOM は **jboss-eap-javaee7** BOM により管理されるため、プロジェクト依存関係でバージョンを管理する必要はありません。

以下に **wildfly-ejb-client-bom** および **wildfly-jms-client-bom** クライアント BOM 依存関係をプロジェクトに追加する方法の例を示します。

```
<dependencyManagement>
  <dependencies>
```

```
    <!-- jboss-eap-javaee7: JBoss stack of the Java EE APIs and related
components.  -->
    <dependency>
      <groupId>org.jboss.bom</groupId>
      <artifactId>jboss-eap-javaee7</artifactId>
      <version>7.0.0.GA</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
  ...
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-ejb-client-bom</artifactId>
    <type>pom</type>
  </dependency>
  <dependency>
    <groupId>org.jboss.eap</groupId>
    <artifactId>wildfly-jms-client-bom</artifactId>
    <type>pom</type>
  </dependency>
  ...
</dependencies>
```

Maven 依存関係および BOM POM ファイルの詳細については、Apache Maven Project - Introduction to the Dependency Mechanism を参照してください。

第3章 クラスローディングとモジュール

3.1. はじめに

3.1.1. クラスロードとモジュールの概要

JBoss EAP は、デプロイされたアプリケーションのクラスパスを制御するためにモジュール形式のクラスロードシステムを使用します。このシステムは、階層クラスローダーの従来のシステムよりも、柔軟性があり、より詳細に制御できます。開発者は、アプリケーションで利用可能なクラスに対して粒度の細かい制御を行い、アプリケーションサーバーで提供されるクラスを無視して独自のクラスを使用するようデプロイメントを設定できます。

モジュール形式のクラスローダーにより、すべての Java クラスはモジュールと呼ばれる論理グループに分けられます。各モジュールは、独自のクラスパスに追加されたモジュールからクラスを取得するために、他のモジュールの依存関係を定義できます。デプロイされた各 JAR および WAR ファイルはモジュールとして扱われるため、開発者はモジュール設定をアプリケーションに追加してアプリケーションのクラスパスの内容を制御できます。

3.1.2. モジュール

モジュールは、クラスローディングおよび依存関係管理に使用されるクラスの論理グループです。JBoss EAP は、**静的**モジュールと**動的**モジュールの 2 つの種類のモジュールを識別します。この 2 つの種類のモジュールの主な違いは、パッケージ化方法です。

静的モジュール
静的モジュールは、アプリケーションサーバーの **`EAP_HOME/modules/`** ディレクトリーで定義されます。各モジュールは **`EAP_HOME/modules/com/mysql/`** のようにサブディレクトリーとして存在します。各モジュールには、**`module.xml`** 設定ファイルとすべての必要な JAR ファイルが含まれるスロットサブディレクトリー (デフォルトでは **`main`**) が含まれます。アプリケーションサーバーにより提供される API は、Java EE API と他の API を含む静的モジュールとして提供されます。

MySQL JDBC ドライバー `module.xml` ファイルの例

```
<?xml version="1.0" ?>
<module xmlns="urn:jboss:module:1.1" name="com.mysql">
  <resources>
    <resource-root path="mysql-connector-java-5.1.36-bin.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

モジュール名 (**`com.mysql`**) は、モジュールのディレクトリー構造 (スロット名 (**`main`**) を除く) に一致する必要があります。

カスタム静的モジュールの作成は、同じサードパーティーライブラリーを使用する同じサーバー上に多くのアプリケーションがデプロイされる場合に役立ちます。これらのライブラリーを各アプリケーションとバンドルする代わりに、管理者はこれらのライブラリーが含まれるモジュールを作成およびインストールできます。アプリケーションは、カスタム静的モジュールで明示的な依存関係を宣言できます。

JBoss EAP ディストリビューションで提供されるモジュールは、**`EAP_HOME/modules`** ディレクトリー

内の **system** ディレクトリーにあります。このため、サードパーティーによって提供されるモジュールから分離されます。また、JBoss EAP 上で使用する、Red Hat により提供されるすべての製品によって、**system** ディレクトリー内にモジュールがインストールされます。

各モジュールに 1 つのディレクトリーを使用して、カスタムモジュールが **EAP_HOME/modules** ディレクトリーにインストールされるようにする必要があります。こうすると、同梱されたバージョンではなく、**system** ディレクトリーに存在するカスタムバージョンのモジュールがロードされるようになります。これにより、ユーザー提供のモジュールがシステムモジュールよりも優先されます。

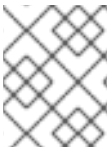
JBOSS_MODULEPATH 環境変数を使用して JBoss EAP がモジュールを検索する場所を変更する場合は、指定された場所の 1 つで **system** サブディレクトリー構造を探します。**system** 構造は、**JBOSS_MODULEPATH** で指定された場所のどこかに存在する必要があります。

動的モジュール
動的モジュールは、各 JAR または WAR デプロイメント (または、EAR 内のサブデプロイメント) に対してアプリケーションサーバーによって作成およびロードされます。動的モジュールの名前は、デプロイされたアーカイブの名前から派生されます。デプロイメントはモジュールとしてロードされるため、依存関係を設定し、他のデプロイメントで依存関係として使用することが可能です。

モジュールは必要な場合にのみロードされます。通常、モジュールは、明示的または暗黙的な依存関係があるアプリケーションがデプロイされる場合にのみロードされます。

3.1.3. モジュールの依存関係

モジュール依存関係は、あるモジュールに他の 1 つまたは複数のモジュールのクラスが必要になるという宣言です。JBoss EAP がモジュールをロードするときに、モジュール形式のクラスローダーがモジュールの依存関係を解析し、各依存関係のクラスをクラスパスに追加します。指定の依存関係が見つからない場合、モジュールはロードできません。



注記

モジュールとモジュール形式のクラスロードシステムに関する完全な詳細については、節モジュールを参照してください。

デプロイされたアプリケーション (JAR や WAR など) は動的モジュールとしてロードされ、依存関係を利用して JBoss EAP によって提供される API にアクセスします。

依存関係には**明示的**と**暗黙的**の 2 つのタイプがあります。

明示的な依存関係

明示的な依存関係は開発者が設定ファイルで宣言します。静的モジュールでは、依存関係を **module.xml** ファイルに宣言できます。動的モジュールでは、デプロイメントの **MANIFEST.MF** または **jboss-deployment-structure.xml** デプロイメント記述子に依存関係を宣言できます。

暗黙的な依存関係

暗黙的な依存関係は、デプロイメントで特定の状態やメタデータが見つかったときに自動的に追加されます。JBoss EAP に同梱される Java EE 7 API は、デプロイメントで暗黙的な依存関係が検出されたときに追加されるモジュールの例になります。

jboss-deployment-structure.xml デプロイメント記述子ファイルを使用して、特定の暗黙的な依存関係を除外するようデプロイメントを設定することも可能です。これは、JBoss EAP が暗黙的な依存関係として追加しようとする特定バージョンのライブラリーをアプリケーションがバンドルする場合に役に立つことがあります。

jboss-deployment-structure.xml デプロイメント記述子の使用の詳細については、節デプロイメントへの明示的なモジュール依存関係の追加を参照してください。

オプションの依存関係
明示的な依存関係は、オプションとして指定できます。オプションの依存関係をロードできなくても、モジュールのロードは失敗しません。ただし、依存関係は後で使用できるようになっても、モジュールのクラスパスには**追加されません**。依存関係はモジュールがロードされるときに利用可能である必要があります。

依存関係のエクスポート
モジュールのクラスパスには独自のクラスとその直接の依存関係のクラスのみが含まれます。モジュールは 1 つの依存関係の依存関係クラスにはアクセスできませんが、暗黙的な依存関係のエクスポートを指定できます。エクスポートされた依存関係は、エクスポートするモジュールに依存するモジュールへ提供されます。

たとえば、モジュール **A** はモジュール **B** に依存し、モジュール **B** はモジュール **C** に依存します。モジュール **A** はモジュール **B** のクラスにアクセスでき、モジュール **B** はモジュール **C** のクラスにアクセスできます。モジュール **A**は以下のいずれかの条件を満たさない限り、モジュール **C** のクラスにアクセスできません。

- モジュール **A** が、モジュール **C** に対する明示的な依存関係を宣言する
- モジュール **B** がモジュール **C** の依存関係をエクスポートする

グローバルモジュール
グローバルモジュールは、JBoss EAP が各アプリケーションへの依存関係として提供するモジュールです。このモジュールをグローバルモジュールの JBoss EAP のリストへ追加すると、モジュールをグローバルモジュールにすることができます。モジュールへの変更は必要ありません。

詳細については、JBoss EAP **Configuration Guide** の節 **Define Global Modules** を参照してください。

3.1.3.1. 管理 CLI を使用したモジュールの依存関係の表示

以下の管理操作を使用すると、特定のモジュールとその依存関係に関する情報を表示できます。

```
/core-service=module-loading:module-info(name=$MODULE_NAME)
```

module-info 出力の例

```
[standalone@localhost:9990 /] /core-service=module-loading:module-
info(name=org.jboss.logmanager
{
    "outcome" => "success",
    "result" => {
        "name" => "org.jboss.logmanager:main",
        "main-class" => undefined,
        "fallback-loader" => undefined,
        "dependencies" => [
            {
                "dependency-name" => "ModuleDependency",
                "module-name" => "javax.api:main",
                "export-filter" => "Reject",
                "import-filter" => "multi-path filter {exclude children of
\"META-INF/\", exclude equals \"META-INF\", default accept}",
```

```
                "optional" => false
            },
            {
                "dependency-name" => "ModuleDependency",
                "module-name" => "org.jboss.modules:main",
                "export-filter" => "Reject",
                "import-filter" => "multi-path filter {exclude children of
\"META-INF/\", exclude equals \"META-INF\", default accept}",
                "optional" => false
            }
        ],
        "local-loader-class" => undefined,
        "resource-loaders" => [
            {
                "type" => "org.jboss.modules.JarFileResourceLoader",
                "paths" => [
                    "",
                    "org/jboss/logmanager",
                    "META-INF/services",
                    "org",
                    "META-INF/maven/org.jboss.logmanager/jboss-
logmanager",
                    "org/jboss",
                    "org/jboss/logmanager/errormanager",
                    "org/jboss/logmanager/formatters",
                    "META-INF",
                    "org/jboss/logmanager/filters",
                    "org/jboss/logmanager/config",
                    "META-INF/maven",
                    "org/jboss/logmanager/handlers",
                    "META-INF/maven/org.jboss.logmanager"
                ]
            },
            {
                "type" =>
"org.jboss.modules.NativeLibraryResourceLoader",
                "paths" => undefined
            }
        ]
    }
}
```

3.1.4. デプロイメントでのクラスローディング

JBoss EAP では、クラスローディングを行うために、デプロイメントはすべてモジュールとして処理されます。このようなデプロイメントは動的モジュールと呼ばれます。クラスローディングの動作はデプロイメントの種類によって異なります。

WAR デプロイメント

WAR デプロイメントは 1 つのモジュールとして考慮されます。**WEB-INF/lib** ディレクトリーのクラスは **WEB-INF/classes** ディレクトリーにあるクラスと同じように処理されます。WAR にパッケージされているクラスはすべて、同じクラスローダーでロードされます。

EAR デプロイメント

EAR デプロイメントは複数のモジュールで構成され、以下のルールに従って定義されます。

1. EAR の **lib/** ディレクトリーは親モジュールと呼ばれる 1 つのモジュールです。
2. また、EAR 内の各 WAR デプロイメントは 1 つのモジュールです。
3. 同様に、EAR 内の EJB JAR デプロイメントも 1 つのモジュールです。

サブデプロイメントモジュール (EAR 内の WAR および JAR デプロイメント) は、自動的に親モジュールに依存しますが、サブデプロイメントモジュール同士が自動的に依存するわけではありません。これは、サブデプロイメントの分離 (subdeployment isolation) と呼ばれ、デプロイメントごとまたはアプリケーションサーバー全体で無効にすることができます。

サブデプロイメントモジュール間の明示的な依存関係は、他のモジュールと同じ方法で追加することが可能です。

3.1.5. クラスローディングの優先順位

JBoss EAP のモジュール形式クラスローダーは、優先順位を決定してクラスローディングの競合が発生しないようにします。

デプロイメントに、パッケージとクラスの完全なリストがデプロイメントごとおよび依存関係ごとに作成されます。このリストは、クラスローディングの優先順位のルールに従って順序付けされます。実行時にクラスをロードすると、クラスローダーはこのリストを検索し、最初に一致したものをロードします。こうすることで、デプロイメントクラスパス内の同じクラスやパッケージの複数のコピーが競合しないようになります。

クラスローダーは上から順にクラスをロードします。

1. **暗黙的な依存関係:** これらの依存関係 (JAVA EE API など) は JBoss EAP によって自動的に追加されます。これらの依存関係には一般的な機能や JBoss EAP によって提供される API が含まれるため、これらの依存関係のクラスローダー優先順位は最も高くなります。
   暗黙的な各依存関係の完全な詳細については、暗黙的なモジュール依存関係 を参照してください。
2. **明示的な依存関係:** これらの依存関係は、アプリケーションの **MANIFEST.MF** ファイルや新しいオプションの JBoss デプロイメント記述子 **jboss-deployment-structure.xml** ファイルを使用してアプリケーション設定に手動で追加されます。
   明示的な依存関係の追加方法については、デプロイメントへの明示的なモジュール依存関係の追加を参照してください。
3. **ローカルリソース:** これらはデプロイメント内にパッケージ化されるクラスファイル (例: WAR ファイルの **WEB-INF/classes** または **WEB-INF/lib** ディレクトリー内) です。
4. **デプロイメント間の依存関係:** これらは EAR デプロイメントにある他のデプロイメントに対する依存関係です。これには、EAR の **lib** ディレクトリーにあるクラスや他の EJB jar で定義されたクラスが含まれることがあります。

3.1.6. 動的モジュールの命名規則

JBoss EAP では、すべてのデプロイメントが、以下の規則に従って名前が付けられたモジュールとしてロードされます。

- WAR および JAR ファイルのデプロイメントは次の形式で名前が付けられます。

```
deployment.DEPLOYMENT_NAME
```

たとえば、**inventory.war** と **store.jar** のモジュール名はそれぞれ **deployment.inventory.war** と **deployment.store.jar** になります。

- エンタープライズアーカイブ (EAR) 内のサブデプロイメントは次の形式で名前が付けられます。

```
deployment.EAR_NAME.SUBDEPLOYMENT_NAME
```

たとえば、エンタープライズアーカイブ **accounts.ear** 内にある **reports.war** のサブデプロイメントのモジュール名は **deployment.accounts.ear.reports.war** になります。

3.1.7. jboss-deployment-structure.xml

jboss-deployment-structure.xml は JBoss EAP のオプションのデプロイメント記述子です。このデプロイメント記述子を使用すると、デプロイメントでクラスローディングを制御できます。

このデプロイメント記述子の XML スキーマは、**/docs/schema/jboss-deployment-structure-1_2.xsd** にあります。

3.2. デプロイメントへの明示的なモジュール依存関係の追加

明示的なモジュール依存関係をアプリケーションに追加すると、これらのモジュールのクラスをデプロイメント時にアプリケーションのクラスパスに追加することができます。

注記

JBoss EAP では、依存関係がデプロイメントに自動的に追加されます。詳細については、暗黙的なモジュール依存関係を参照してください。

前提条件

1. モジュールの依存関係を追加するソフトウェアプロジェクト。
2. 依存関係として追加するモジュールの名前を知っている必要があります。JBoss EAP に含まれる静的モジュールのリストについては、含まれるモジュールを参照してください。モジュールが他のデプロイメントである場合は、動的モジュールの名前付けを参照してモジュール名を判断してください。

依存関係を設定するには、以下の 2 つの方法があります。

- デプロイメントの **MANIFEST.MF** ファイルにエントリーを追加します。
- **jboss-deployment-structure.xml** デプロイメント記述子にエントリーを追加します。

MANIFEST.MF への依存関係設定の追加

MANIFEST.MF ファイルの必要な依存関係エントリーを作成するよう Maven プロジェクトを設定できます。

1. **MANIFEST.MF** という名前のファイルを作成します (プロジェクトにない場合)。Web アプリケーション (WAR) では、このファイルを **META-INF** ディレクトリーに追加します。EJB アーカイブ (JAR) では、このファイルを **META-INF** ディレクトリーに追加します。
2. 依存関係モジュール名をコンマで区切り、依存関係エントリーを **MANIFEST.MF** ファイルへ追加します。

```
Dependencies: org.javassist, org.apache.velocity, org.antlr
```

- 依存関係をオプションにするには、依存関係エントリーのモジュール名に **optional** を付けます。

```
Dependencies: org.javassist optional, org.apache.velocity
```

- 依存関係エントリーのモジュール名に **export** を付けると、依存関係をエクスポートすることができます。

```
Dependencies: org.javassist, org.apache.velocity export
```

- **annotations** フラグは、EJB インターセプターを宣言するときなど、アノテーションのスキャン中に処理する必要があるアノテーションがモジュールの依存関係に含まれる場合に必要になります。この設定を行わないと、モジュールに宣言された EJB インターセプターをデプロイメントで使用できません。アノテーションのスキャンが関係するその他の状況でも、この設定が必要になる場合があります。

```
Dependencies: org.javassist, test.module annotations
```

- デフォルトでは、依存関係の **META-INF** 内のアイテムにはアクセスできません。**services** 依存関係により **META-INF/services** のアイテムにアクセスできるようになり、モジュール内の **services** をロードできるようになります。

```
Dependencies: org.javassist, org.hibernate services
```

- **beans.xml** ファイルをスキャンし、生成される Bean をアプリケーションが利用できるようにするために、**meta-inf** 依存関係を使用できます。

```
Dependencies: org.javassist, test.module meta-inf
```

jboss-deployment-structure.xml への依存関係設定の追加

1. **jboss-deployment-structure.xml** という名前の新しいファイルを作成し (アプリケーションにない場合)、プロジェクトに追加します。このファイルは **<jboss-deployment-structure>** がルート要素の XML ファイルです。

```
<jboss-deployment-structure>

</jboss-deployment-structure>
```

   Web アプリケーション (WAR) では、このファイルを **WEB-INF** ディレクトリーに追加します。EJB アーカイブ (JAR) では、このファイルを **META-INF** ディレクトリーに追加します。

2. **<deployment>** 要素をドキュメントルート内に作成し、その中に **<dependencies>** 要素を作成します。

3. **<dependencies>** ノード内に各モジュールの依存関係に対するモジュール要素を追加します。**name** 属性をモジュールの名前に設定します。

```
<module name="org.javassist" />
```

- 値が **true** のモジュールエントリーに **optional** 属性を追加することにより依存関係をオプションにすることができます。この属性のデフォルト値は **false** です。

```
<module name="org.javassist" optional="true" />
```

- 値が **true** のモジュールエントリーに **export** 属性を追加することにより依存関係をエクスポートできます。この属性のデフォルト値は **false** です。

```
<module name="org.javassist" export="true" />
```

- アノテーションのスキャン中に処理する必要があるアノテーションがモジュール依存関係に含まれる場合は、**annotations** フラグが使用されます。

```
<module name="test.module" annotations="true" />
```

- **Services** 依存関係は、この依存関係にある **services** を使用するかどうか、およびどのように使用するかを指定します。デフォルト値は **none** です。この属性に **import** の値を指定することは、依存関係モジュールの **META-INF/services** パスを含むインポートフィルターリストの最後にフィルターを追加することと同じです この属性に **export** の値を設定することは、エクスポートフィルターリストに対して同じアクションを実行することと同じです。

```
<module name="org.hibernate" services="import" />
```

- **META-INF** 依存関係は、この依存関係の **META-INF** エントリーを使用するかどうか、およびどのように使用するかを指定します。デフォルト値は **none** です。この属性に **import** の値を指定することは、依存関係モジュールの **META-INF/**** パスを含むインポートフィルターリストの最後にフィルターを追加することと同じです この属性に **export** の値を設定することは、エクスポートフィルターリストに対して同じアクションを実行することと同じです。

```
<module name="test.module" meta-inf="import" />
```

例: 2 つの依存関係がある jboss-deployment-structure.xml

```
<jboss-deployment-structure>
   <deployment>
      <dependencies>
         <module name="org.javassist" />
         <module name="org.apache.velocity" export="true" />
      </dependencies>
   </deployment>
</jboss-deployment-structure>
```

JBoss EAP では、デプロイ時に、指定されたモジュールからアプリケーションのクラスパスにクラスが追加されます。

Jandex インデックスの作成

annotations フラグを使用するには、モジュールに Jandex インデックスが含まれる必要があります。JBoss EAP 7.0 では、これは自動的に生成されます。ただし、このインデックスを手動で追加する場合は、後方互換性を確保するために、モジュールに追加する新しい「index JAR」を作成します。Jandex JAR を使用してインデックスをビルドした後、新しい JAR ファイルに挿入します。

Jandex インデックスの作成:

1. インデックスを作成します。

```
java -jar modules/system/layers/base/org/jboss/jandex/main/jandex-
jandex-2.0.0.Final-redhat-1.jar $JAR_FILE
```

2. 一時作業領域を作成します。

```
mkdir /tmp/META-INF
```

3. インデックスファイルをワーキングディレクトリーへ移動します。

```
mv $JAR_FILE.ifx /tmp/META-INF/jandex.idx
```

   a. オプション 1: インデックスを新しい JAR ファイルに含めます。

```
jar cf index.jar -C /tmp META-INF/jandex.idx
```

   JAR をモジュールディレクトリーに置き、 **module.xml** を編集してリソースルートへ追加します。

   b. オプション 2: インデックスを既存の JAR に追加します。

```
java -jar /modules/org/jboss/jandex/main/jandex-1.0.3.Final-
redhat-1.jar -m $JAR_FILE
```

4. アノテーションインデックスを使用するようモジュールインポートに指示し、アノテーションのスキャンでアノテーションを見つけられるようにします。

   a. オプション 1: MANIFEST.MF を使用してモジュールの依存関係を追加する場合は、**annotations** をモジュール名の後に追加します。たとえば、

```
Dependencies: test.module, other.module
```

   を以下のように変更します。

```
Dependencies: test.module annotations, other.module
```

   b. オプション 2: **jboss-deployment-structure.xml** を使用してモジュールの依存関係を追加する場合は、モジュールの依存関係に **annotations="true"** を追加します。

注記

静的モジュール内のクラスで定義されたアノテーション付き Java EE コンポーネントをアプリケーションで使用する場合は、アノテーションインデックスが必要です。JBoss EAP 7.0 では、静的モジュールのアノテーションインデックスは自動的に生成されるため、作成する必要がありません。ただし、**MANIFEST.MF** または **jboss-deployment-structure.xml** ファイルのいずれかに依存関係を追加して、アノテーションを使用するようモジュールインポートに指示する必要があります。

3.3. MAVEN を使用した MANIFEST.MF エントリーの生成

Maven JAR、EJB、または WAR パッケージングプラグインを使用する Maven プロジェクトでは、**Dependencies** エントリーを持つ **MANIFEST.MF** ファイルを生成することができます。この場合、依存関係の一覧は自動的に生成されず、**pom.xml** に指定された詳細が含まれる **MANIFEST.MF** ファイルのみが作成されます。

Maven を使用して **MANIFEST.MF** エントリーを生成する前に、以下のものが必要になります。

- JAR、EJB、または WAR プラグイン (**maven-jar-plugin**、**maven-ejb-plugin**、または **maven-war-plugin**) のいずれかを使用している Maven プロジェクト。
- プロジェクトのモジュール依存関係の名前を知っている必要があります。JBoss EAP に含まれる静的モジュールのリストについては、含まれるモジュールを参照してください。モジュールが他のデプロイメントである場合は、動的モジュールの名前付けを参照してモジュール名を判断してください。

モジュール依存関係が含まれる MANIFEST.MF ファイルの生成

1. プロジェクトの **pom.xml** ファイルにあるパッケージングプラグイン設定に次の設定を追加します。

```
<configuration>
   <archive>
      <manifestEntries>
         <Dependencies></Dependencies>
      </manifestEntries>
   </archive>
</configuration>
```

2. モジュール依存関係のリストを **<Dependencies>** 要素に追加します。**MANIFEST.MF** ファイルに依存関係を追加するときと同じ形式を使用します。

```
<Dependencies>org.javassist, org.apache.velocity</Dependencies>
```

ここでは、**optional** 属性と **export** 属性を使用することもできます。

```
<Dependencies>org.javassist optional, org.apache.velocity
export</Dependencies>
```

3. Maven アセンブリーゴールを使用してプロジェクトをビルドします。

```
[Localhost ]$ mvn assembly:single
```

アセンブリーゴールを使用してプロジェクトをビルドすると、指定のモジュール依存関係を持つ **MANIFEST.MF** ファイルが最終アーカイブに含まれます。

例: pom.xml で設定されたモジュール依存関係

注記

この例は WAR プラグインの例になりますが、JAR や EJB プラグイン (maven-jar-plugin や maven-ejb-plugin) でも動作します。

```
<plugins>
   <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
         <archive>
            <manifestEntries>
               <Dependencies>org.javassist,
org.apache.velocity</Dependencies>
            </manifestEntries>
         </archive>
      </configuration>
   </plugin>
</plugins>
```

3.4. モジュールが暗黙的にロードされないようにする

暗黙的な依存関係がロードされないようデプロイ可能なアプリケーションを設定できます。これは、アプリケーションサーバーにより提供される暗黙的な依存関係とは異なるバージョンのライブラリーやフレームワークがアプリケーションに含まれる場合に役に立つことがあります。

前提条件

- 暗黙的な依存関係を除外するソフトウェアプロジェクト。
- 除外するモジュール名を知っている必要があります。暗黙的な依存関係のリストや状態については暗黙的なモジュール依存関係を参照してください。

jboss-deployment-structure.xml への依存関係除外設定の追加

1. **jboss-deployment-structure.xml** という名前の新しいファイルを作成し (アプリケーションにない場合)、プロジェクトに追加します。このファイルは **<jboss-deployment-structure>** がルート要素の XML ファイルです。

   ```
   <jboss-deployment-structure>

   </jboss-deployment-structure>
   ```

   Web アプリケーション (WAR) では、このファイルを **WEB-INF** ディレクトリーに追加します。EJB アーカイブ (JAR) では、このファイルを **META-INF** ディレクトリーに追加します。

2. **<deployment>** 要素をドキュメントルート内に作成し、その中に **<exclusions>** 要素を作成します。

   ```
   <deployment>
      <exclusions>

      </exclusions>
   </deployment>
   ```

3. exclusions 要素内で、除外する各モジュールに対して **<module>** 要素を追加します。 **name** 属性をモジュールの名前に設定します。

```
<module name="org.javassist" />
```

例: 2 つのモジュールの除外

```
<jboss-deployment-structure>
   <deployment>
      <exclusions>
         <module name="org.javassist" />
         <module name="org.dom4j" />
      </exclusions>
   </deployment>
</jboss-deployment-structure>
```

3.5. サブシステムをデプロイメントから除外

サブシステムの除外は、サブシステムの削除と同じ効果がありますが、単一のデプロイメントにのみ適用されます。**jboss-deployment-structure.xml** 設定ファイルを編集することにより、デプロイメントからサブシステムを除外できます。

サブシステムの除外

1. **jboss-deployment-structure.xml** ファイルを編集します。
2. **<deployment>** タグ内に以下の XML を追加します。

```
<exclude-subsystems>
  <subsystem name="SUBSYSTEM_NAME" />
</exclude-subsystems>
```

3. **jboss-deployment-structure.xml** ファイルを保存します。

サブシステムのデプロイメントユニットプロセッサーがデプロイメント上で実行されなくなります。

例: サンプル jboss-deployment-structure.xml ファイル

```
<jboss-deployment-structure xmlns="urn:jboss:deployment-structure:1.2">
  <ear-subdeployments-isolated>true</ear-subdeployments-isolated>
  <deployment>
    <exclude-subsystems>
      <subsystem name="jaxrs" />
    </exclude-subsystems>
    <exclusions>
      <module name="org.javassist" />
    </exclusions>
    <dependencies>
      <module name="deployment.javassist.proxy" />
      <module name="deployment.myjavassist" />
      <module name="myservicemodule" services="import"/>
    </dependencies>
    <resources>
      <resource-root path="my-library.jar" />
    </resources>
  </deployment>
  <sub-deployment name="myapp.war">
```

```
    <dependencies>
      <module name="deployment.myear.ear.myejbjar.jar" />
    </dependencies>
    <local-last value="true" />
  </sub-deployment>
  <module name="deployment.myjavassist" >
    <resources>
     <resource-root path="javassist.jar" >
       <filter>
         <exclude path="javassist/util/proxy" />
       </filter>
     </resource-root>
    </resources>
  </module>
  <module name="deployment.javassist.proxy" >
    <dependencies>
      <module name="org.javassist" >
        <imports>
          <include path="javassist/util/proxy" />
          <exclude path="/**" />
        </imports>
      </module>
    </dependencies>
  </module>
</jboss-deployment-structure>
```

3.6. デプロイメントでのプログラムを用いたクラスローダーの使用

3.6.1. デプロイメントでのプログラムによるクラスおよびリソースのロード

プログラムを用いて、アプリケーションコードでクラスやリソースを検索またはロードできます。

Class.forName() メソッドを使用したクラスのロード

`Class.forName()` メソッドを使用すると、プログラムでクラスをロードおよび初期化できます。このメソッドには 2 つのシグネチャーがあります。

- **Class.forName(String className):** このシグネチャーは、1 つのパラメーター (ロードする必要があるクラスの名前) のみを取ります。このメソッドシグネチャーを使用すると、現在のクラスのクラスローダーによってクラスがロードされ、デフォルトで新たにロードされたクラスが初期化されます。

- **Class.forName(String className, boolean initialize, ClassLoader loader):** このシグネチャーは、クラス名、クラスを初期化するかどうかを指定するブール値、およびクラスをロードする ClassLoader の 3 つのパラメーターを想定します。

プログラムでクラスをロードする場合は、3 つの引数のシグネチャーを用いる方法が推奨されます。このシグネチャーを使用すると、ロード時に目的のクラスを初期化するかどうかを制御できます。また、JVM はコールスタックをチェックして、使用するクラスローダーを判断する必要がないため、クラスローダーの取得および提供がより効率的になります。コードが含まれるクラスの名前が **`CurrentClass`** である場合は、**`CurrentClass.class.getClassLoader()`** メソッドを使用してクラスのクラスローダーを取得できます。

例: ロードするクラスローダーを提供し、TargetClass を初期化する

以下は、ロードするクラスローダーを提供し、**`TargetClass`** クラスを初期化する例になります。

```
Class<?> targetClass = Class.forName("com.myorg.util.TargetClass", true,
CurrentClass.class.getClassLoader());
```

名前ですべてのリソースを検索

リソースの名前とパスがわかり、直接そのリソースをロードする場合は、標準的な Java 開発キットクラスまたは ClassLoader API を使用するのが最良の方法です。

- **単一リソースをロードする:** ご使用のクラスと同じディレクトリーまたはデプロイメントの他のクラスと同じディレクトリーにある単一のリソースをロードする場合は、**`Class.getResourceAsStream()`** メソッドを使用できます。

  ```
  InputStream inputStream =
  CurrentClass.class.getResourceAsStream("targetResourceName");
  ```

- **単一リソースのすべてのインスタンスをロードする:** デプロイメントのクラスローダーが見える単一リソースのすべてのインスタンスをロードするには、**`Class.getClassLoader().getResources(String resourceName)`** メソッドを使用します。ここで、**`resourceName`** はリソースの完全修飾パスに置き換えます。このメソッドは、指定の名前でクラスローダーがアクセスできるリソースに対し、すべての **`URL`** オブジェクトの列挙を返します。その後、URL の配列で繰り返し処理し、**`openStream()`** メソッドを使用して各ストリームを開くことができます。
  例: リソースのすべてのインスタンスをロードし、結果で繰り返し処理を行う

  ```
  Enumeration<URL> urls =
  CurrentClass.class.getClassLoader().getResources("full/path/to/resou
  rce");
  while (urls.hasMoreElements()) {
      URL url = urls.nextElement();
      InputStream inputStream = null;
      try {
          inputStream = url.openStream();
          // Process the inputStream
          ...
      } catch(IOException ioException) {
          // Handle the error
      } finally {
          if (inputStream != null) {
              try {
                  inputStream.close();
              } catch (Exception e) {
                  // ignore
              }
          }
      }
  }
  ```

URL インスタンスはローカルストレージからロードされるため、**`openConnection()`** や他の関連するメソッドを使用する必要はありません。ストリームは非常に簡単に使用でき、ストリームを使用することにより、コードの複雑さが最小限に抑えられます。

- **クラスローダーよりクラスファイルをロードする:** クラスがすでにロードされている場合は、以下の構文を使用して、そのクラスに対応するクラスファイルをロードできます。
  例: ロードされたクラスのクラスファイルをロードする

  -

```
InputStream inputStream =
CurrentClass.class.getResourceAsStream(TargetClass.class.getSimpleNa
me() + ".class");
```

クラスがロードされていない場合は、クラスローダーを使用し、パスを変換する必要があります。

例: ロードされていないクラスのクラスファイルをロードする

```
String className = "com.myorg.util.TargetClass"
InputStream inputStream =
CurrentClass.class.getClassLoader().getResourceAsStream(className.re
place('.', '/') + ".class");
```

3.6.2. デプロイメントでのプログラムによるリソースの繰り返し

JBoss Modules ライブラリーは、すべてのデプロイメントリソースを繰り返し処理するために複数の API を提供します。JBoss Modules API の JavaDoc は http://docs.jboss.org/jbossmodules/1.3.0.Final/api/ にあります。これらの API を使用するには、以下の依存関係を **MANIFEST.MF** に追加する必要があります。

```
依存関係: org.jboss.modules
```

これらの API により柔軟性が向上しますが、直接のパスルックアップよりも動作がかなり遅くなることに注意してください。

このトピックでは、アプリケーションコードでプログラムを用いてリソースを繰り返す方法を説明します。

- **デプロイメント内およびすべてのインポート内のリソースをリストする:** 場合によっては、正確なパスでリソースをルックアップできないことがあります。たとえば、正確なパスがわからなかったり、指定のパスで複数のファイルをチェックしたりする必要がある場合などです。このような場合、JBoss Modules ライブラリーはすべてのデプロイメントを繰り返し処理するための API を複数提供します。2 つのメソッドのいずれかを使用すると、デプロイメントでリソースを繰り返し処理できます。
  - **単一のモジュールで見つかったすべてのリソースを繰り返し処理する:** **ModuleClassLoader.iterateResources()** メソッドは、このモジュールクラスローダー内のすべてのリソースを繰り返し処理します。このメソッドは、検索を開始するディレクトリーの名前と、サブディレクトリーで再帰的に処理するかどうかを指定するブール値の 2 つの引数を取ります。
    以下の例は、ModuleClassLoader の取得方法と、**bin/** ディレクトリーにあるリソースのイテレーターの取得方法 (サブディレクトリーを再帰的に検索) を示しています。

例: サブディレクトリーを再帰的に検索し、bin ディレクトリーのリソースを検索する

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
moduleClassLoader.iterateResources("bin",true);
```

取得されたイテレーターは、一致した各リソースをチェックし、名前とサイズのクエリー (可能な場合) を行うために使用できます。また、読み取り可能ストリームを開いたり、リソースの URL を取得するために使用できます。

- **単一モジュールで見つかったすべてのリソースとインポートされたリソースを繰り返し処理する: `Module.iterateResources()`** メソッドは、このモジュールクラスローダー内のすべてのリソース (モジュールにインポートされたリソースを含む) を繰り返し処理します。このメソッドは、前述のメソッドよりもはるかに大きなセットを返します。このメソッドには、特定パターンの結果を絞り込むフィルターである引数が必要になります。代わりに、PathFilters.acceptAll() を指定してセット全体を返すことも可能です。

例: このモジュールで、インポートを含むすべてのリソースを検索する

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.acceptAll());
```

- **パターンと一致するすべてのリソースを検索する:** デプロイメント内またはデプロイメントの完全なインポートセット内で特定のリソースのみを見つける必要がある場合は、リソースの繰り返しをフィルターする必要があります。JBoss Modules のフィルター API は、リソースの繰り返しをフィルターする複数のツールを提供します。
  - **依存関係の完全なセットをチェックする:** 依存関係の完全なセットをチェックする必要がある場合は、**`Module.iterateResources()`** メソッドの **`PathFilter`** パラメーターを使用して、一致する各リソースの名前を確認できます。
  - **デプロイメント依存関係を確認する:** デプロイメント内のみを検索する必要がある場合は、**`ModuleClassLoader.iterateResources()`** メソッドを使用しますが、追加のメソッドを使用して結果となるイテレーターをフィルターする必要があります。**`PathFilters.filtered()`** メソッドは、リソースイテレーターのフィルターされたビューを提供できます。**`PathFilters`** クラスには、さまざまな関数を実行するフィルターを作成する多くの静的メソッドが含まれています。これには、子パスや完全一致の検索、Ant 形式の「glob」パターンの一致などが含まれます。
- **リソースのフィルターに関する追加コード例:** 以下の例は、異なる基準を基にリソースをフィルターする方法を示しています。

例: デプロイメントで messages.properties という名前のファイルをすべて検索する

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
PathFilters.filtered(PathFilters.match("**/messages.properties"),
moduleClassLoader.iterateResources("", true));
```

例: デプロイメントおよびインポートで messages.properties という名前のファイルをすべて検索する

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.match("**/message.properties"));
```

例: デプロイメントで my-resources という名前のディレクトリー内にあるすべてのファイルを検索する

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
```

```
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
PathFilters.filtered(PathFilters.match("**/my-resources/**"),
moduleClassLoader.iterateResources("", true));
```

例: デプロイメントおよびインポートで message または errors という名前のファイルをすべて検索する

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Module module = moduleClassLoader.getModule();
Iterator<Resource> moduleResources =
module.iterateResources(PathFilters.any(PathFilters.match("**/messages"),
PathFilters.match("**/errors"));
```

例: デプロイメントで特定パッケージにあるすべてのファイルを検索する

```
ModuleClassLoader moduleClassLoader = (ModuleClassLoader)
TargetClass.class.getClassLoader();
Iterator<Resource> mclResources =
moduleClassLoader.iterateResources("path/form/of/packagename", false);
```

3.7. クラスローディングとサブデプロイメント

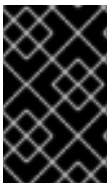
3.7.1. エンタープライズアーカイブのモジュールおよびクラスロード

エンタープライズアーカイブ (EAR) は、JAR または WAR デプロイメントのように、単一モジュールとしてロードされません。これらは、複数の一意のモジュールとしてロードされます。

以下のルールによって、EAR に存在するモジュールが決定されます。

- EAR アーカイブのルートにある **lib/** ディレクトリーの内容はモジュールです。これは、親モジュールと呼ばれます。
- 各 WAR および EJB JAR サブデプロイメントはモジュールです。これらのモジュールの動作は、他のモジュールおよび親モジュールの暗黙的な依存関係と同じです。
- サブデプロイメントでは、親モジュールとすべての他の非 WAR サブデプロイメントに暗黙的な依存関係が存在します。

JBoss EAP では、サブデプロイメントクラスローダーの分離がデフォルトで無効になるため、非 WAR サブデプロイメントの暗黙的な依存関係が発生します。



重要

サブデプロイメントでは、WAR サブデプロイメントに暗黙的な依存関係が存在しません。他のモジュールと同様に、サブデプロイメントは、別のサブデプロイメントの明示的な依存関係で設定できます。

サブデプロイメントクラスローダーの分離は、厳密な互換性が必要な場合に有効にできます。これは、単一の EAR デプロイメントまたはすべての EAR デプロイメントに対して有効にできます。Java EE 6 の仕様では、依存関係が各サブデプロイメントの **MANIFEST.MF** ファイルの **Class-Path** エントリー

として明示的に宣言されている場合を除き、移植可能なアプリケーションがお互いにアクセスできるサブデプロイメントに依存しないことが推奨されます。

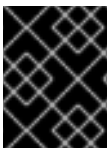
3.7.2. サブデプロイメントクラスローダーの分離

エンタープライズアーカイブ (EAR) の各サブデプロイメントは独自のクラスローダーを持つ動的モジュールです。デフォルトでは、サブデプロイメントは他のサブデプロイメントのリソースにアクセスできます。

サブデプロイメントが他のサブデプロイメントのリソースにアクセスすることが許可されていない場合は、厳格なサブデプロイメントの分離を有効にできます。

3.7.3. EAR 内のサブデプロイメントクラスローダーの分離を有効にする

このタスクでは、EAR の特別なデプロイメント記述子を使用して EAR デプロイメントのサブデプロイメントクラスローダーの分離を有効にする方法を示します。アプリケーションサーバーを変更する必要はなく、他のデプロイメントは影響を受けません。



重要

サブデプロイメントクラスローダーの分離が無効であっても、WAR を依存関係として追加することはできません。

1. **デプロイメント記述子ファイルを追加する: `jboss-deployment-structure.xml`** デプロイメント記述子ファイルを EAR の **`META-INF`** ディレクトリーへ追加し (このファイルが存在しない場合)、次の内容を追加します。

```
<jboss-deployment-structure>

</jboss-deployment-structure>
```

2. **<ear-subdeployments-isolated> 要素を追加する: `<ear-subdeployments-isolated>`** 要素を **`jboss-deployment-structure.xml`** ファイルへ追加し (この要素が存在しない場合)、内容が **`true`** となるようにします。

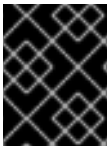
```
<ear-subdeployments-isolated>true</ear-subdeployments-isolated>
```

結果

この EAR デプロイメントに対してサブデプロイメントクラスローダーの分離が有効になります。つまり、EAR のサブデプロイメントは WAR ではないサブデプロイメントごとに自動的な依存関係を持ちません。

3.7.4. エンタープライズアーカイブのサブデプロイメント間で共有するセッションの設定

JBoss EAP では、EAR に含まれる WAR モジュールサブデプロイメント間でセッションを共有するようエンタープライズアーカイブ (EAR) を設定する機能が提供されます。この機能はデフォルトで無効になり、EAR の **`META-INF/jboss-all.xml`** ファイルで明示的に有効にする必要があります。



重要

この機能は標準的サーブレット機能ではないため、この機能が有効な場合はアプリケーションを移植できいないことがあります。

EAR 内の WAR 間で共有するセッションを有効にするには、EAR の **`META-INF/jboss-all.xml`** で **shared-session-config** 要素を宣言する必要があります。

`META-INF/jboss-all.xml` の例

```
<jboss umlns="urn:jboss:1.0">
  ...
  <shared-session-config xmlns="urn:jboss:shared-session-config:1.0">
  </shared-session-config>
  ...
</jboss>
```

shared-session-config 要素は、EAR 内のすべての WAR に対して共有セッションマネージャーを設定するために使用されます。**shared-session-config** 要素が存在する場合は、EAR 内のすべての WAR で同じセッションマネージャーが共有されます。ここで行われる変更は、EAR 内に含まれる **すべての** WAR に影響します。

3.7.4.1. 共有セッション設定オプションのリファレンス

shared-session-config 要素の構造は以下のとおりです。

- shared-session-config
  - max-active-sessions
  - session-config
    - session-timeout
    - cookie-config
      - name
      - domain
      - path
      - comment
      - http-only
      - secure
      - max-age
    - tracking-mode
  - replication-config
    - cache-name

- replication-granularity

META-INF/jboss-all.xml の例

```
<jboss umlns="urn:jboss:1.0">
  <shared-session-config xmlns="urn:jboss:shared-session-config:1.0">
    <max-active-sessions>10</max-active-sessions>
    <session-config>
      <session-timeout>0</session-timeout>
      <cookie-config>
        <name>JSESSIONID</name>
        <domain>domainName</domain>
        <path>/cookiePath</path>
        <comment>cookie comment</comment>
        <http-only>true</http-only>
        <secure>true</secure>
        <max-age>-1</max-age>
      </cookie-config>
    </session-config>
    <tracking-mode>COOKIE</tracking-mode>
  </shared-session-config>
  <replication-config>
    <cache-name>web</cache-name>
    <replication-granularity>SESSION</replication-granularity>
  </replication-config>
</jboss>
```

shared-session-config

共有セッション設定のルート要素。この要素が **META-INF/jboss-all.xml** に存在しない場合は、EAR に含まれるすべてのデプロイ済み WAR で単一のセッションマネージャーが共有されます。

max-active-sessions

許可される最大セッション数。

session-config

EAR に含まれるすべてのデプロイ済み WAR に対するセッション設定パラメーターを含みます。

session-timeout

EAR に含まれるデプロイ済み WAR で作成されたすべてのセッションに対するデフォルトのセッションタイムアウト間隔を定義します。指定されたタイムアウトは、分単位の整数で表記する必要があります。タイムアウトが **0** またはそれよりも小さい値である場合は、コンテナーにより、セッションのデフォルトの動作がタイムアウトしなくなります。この要素が指定されない場合は、コンテナーでデフォルトのタイムアウト期間を設定する必要があります。

cookie-config

EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーを含みます。

name

EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられる名前。デフォルト値は **JSESSIONID** です。

domain

EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられるドメイン名。

path

EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられるパス。

comment

EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられるコメント。

http-only

EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーを **HttpOnly** とマークするかどうかを指定します。

secure

対応するセッションを開始した要求が HTTPS ではなくプレーンな HTTP を使用している場合であっても、EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーをセキュアとマークするかどうかを指定します。

max-age

EAR に含まれるデプロイ済み WAR により作成されたセッション追跡クッキーに割り当てられる有効期間 (秒単位)。デフォルト値は **-1** です。

tracking-mode

EAR に含まれるデプロイ済み WAR により作成されたセッションの追跡モードを定義します。

replication-config

HTTP セッションクラスタリング設定を含みます。

cache-name

このオプションは、クラスタリング専用です。セッションデータを格納する Infinispan コンテナーとキャッシュの名前を指定します。デフォルト値 (明示的に設定されていない場合) は、アプリケーションサーバーによって決定されます。キャッシュコンテナー内で特定のキャッシュを使用するには、**container.cache** という形式 (たとえば、**web.dist**) を使用します。名前が修飾されてない場合は、指定されたコンテナーのデフォルトのキャッシュが使用されます。

replication-granularity

このオプションはクラスタリング専用です。セッションレプリケーションの粒度を決定します。可能な値は **SESSION** と **ATTRIBUTE** であり、デフォルト値は **SESSION** です。

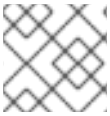
SESSION 粒度が使用される場合は、すべてのセッション属性がレプリケートされます (要求のスコープ内でいずれかのセッション属性が変更された場合)。このポリシーは、オブジェクト参照が複数のセッション属性で共有されるときに必要になります。ただし、セッション属性が非常に大きい場合や頻繁に変更されない場合は非効率になることがあります。これは、属性が変更されたかどうかに関係なく、すべての属性をレプリケートする必要があるためです。

ATTRIBUTE 粒度が使用される場合は、要求のスコープ内で変更された属性のみがレプリケートされます。オブジェクト参照が複数のセッション属性で共有される場合、このポリシーは適切ではありません。セッション属性が非常に大きい場合や頻繁に変更されない場合は **SESSION** よりも効率的になることがあります。

3.8. カスタムモードでのタグライブラリー記述子 (TLD) のデプロイ

共通のタグライブラリー記述子 (TLD) を使用する複数のアプリケーションがある場合、アプリケーションから TLD を分離し、一元的で一意な場所に置くと有用であることがあります。これにより、TLD を使用するアプリケーションごとに更新を行う必要がなくなり、TLD への追加や更新が簡単になります。

これを行うには、TLD JAR が含まれるカスタム JBoss EAP モジュールを作成し、アプリケーションでそのモジュールの依存関係を宣言します。



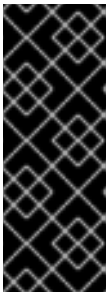
注記

少なくとも 1 つの JAR に TLD が含まれ、TLD が **`META-INF`** に含まれるようにします。

カスタムモジュールでの TLD のデプロイ

1. 管理 CLI を使用して、JBoss EAP インスタンスへ接続し、以下のコマンドを実行して TLD JAR が含まれるカスタムモジュールを作成します。

```
module add --name=MyTagLibs --resources=/path/to/TLDarchive.jar
```



重要

Using the **`module`** management CLI command to add and remove modules is provided as technology preview only. This command is not appropriate for use in a managed domain or when connecting to the management CLI remotely. Modules should be added and removed manually in a production environment. For more information, see the Create a Custom Module Manually and Remove a Custom Module Manually sections of the JBoss EAP **Configuration Guide**.

TLD が依存関係を必要とするクラスとともにパッケージ化されている場合は、**`--dependencies=`** オプションを使用して、カスタムモジュールの作成時にこれらの依存関係を指定するようにします。

モジュールを作成するときに、システムのファイルシステム固有の区切り文字を使用して複数の JAR リソースを指定できます。

- Linux の場合 - **`:`** 例、**`--resources=<path-to-jar>:<path-to-another-jar>`**
- Windows の場合 - **`;`** 例、**`--resources=<path-to-jar>;<path-to-another-jar>`**

注記

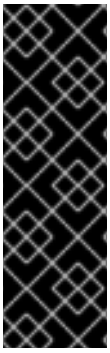
`--resources`

これは **`--module-xml`** を使用しない限り必要です。ファイルシステム固有のパス区切り文字 (たとえば、**`java.io.File.pathSeparatorChar`**) で区切られたファイルシステムパス (通常は JAR ファイル) をリストします。指定されたファイルは作成されたモジュールのディレクトリーにコピーされます。

`--resource-delimiter`

これは、リソース引数のオプションのユーザー定義パス区切り文字です。この引数が存在する場合、コマンドパーサーはファイルシステム固有のパス区切り文字の代わりにその値を使用します。これにより、**`modules`** コマンドをクロスプラットフォームのスクリプトで使用できるようになります。

2. ご使用のアプリケーションで、デプロイメントへの明示的なモジュール依存関係の追加で説明されているいずれかの方法を使用して新しい MyTagLibs カスタムモジュールの依存関係を宣言します。



重要

依存関係を宣言するときは必ず **`META-INF`** もインポートしてください。たとえば、**`MANIFEST.MF`** の場合は以下のようになります。

```
Dependencies: com.MyTagLibs meta-inf
```

`jboss-deployment-structure.xml` の場合は、**`meta-inf`** 属性を使用してください。

3.9. 参考情報

3.9.1. 暗黙的なモジュール依存関係

以下の表には、依存関係としてデプロイメントに自動的に追加されるモジュールと、依存関係をトリガーする条件が記載されています。

表3.1 暗黙的なモジュール依存関係

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
アプリケーションクライアント	• **`org.omg.api`** • **`org.jboss.xnio`**		

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
Batch	• `javax.batch.api` • `org.jberet.jberet-core` • `org.wildfly.jberet`		
Bean の検証	• `org.hibernate.validator` • `javax.validation.api`		
コアサーバー	• `javax.api` • `sun.jdk` • `org.jboss.vfs` • `ibm.jdk`		
DriverDependenciesProcessor		• `javax.transaction.api`	

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
EE	• **org.jboss.invocation** (**org.jboss.invocation.proxy.classloading** を除く) • **org.jboss.as.ee** (**org.jboss.as.ee.component.serialization**、**org.jboss.as.ee.concurrent**、**org.jboss.as.ee.concurrent.handle** を除く) • **org.wildfly.naming** • **javax.annotation.api** • **javax.enterprise.concurrent.api** • **javax.interceptor.api** • **javax.json.api** • **javax.resource.api** • **javax.rmi.api** • **javax.xml.bind.api** • **javax.api** • **org.glassfish.javax.el** • **org.glassfish.javax.enterprise.concurrent**		

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
	`javax.ejb.api` • `javax.xml.rpc.api` • `org.jboss.ejb-client` • `org.jboss.iiop-client` • `org.jboss.as.ejb3`	`org.wildfly.iiop-openjdk`	
IIOP	• `org.omg.api` • `javax.rmi.api` • `javax.orb.api`		
JAX-RS (RESTEasy)	• `javax.xml.bind.api` • `javax.ws.rs.api` • `javax.json.api` • `org.jboss.resteasy.resteasy-atom-provider` • `org.jboss.resteasy.resteasy-crypto` • `org.jboss.resteasy.resteasy-validator-provider-11` • `org.jboss.resteasy.resteasy-jaxrs` • `org.jboss.resteasy.resteasy-jaxb-provider`	• `org.jboss.resteasy.resteasy-cdi`	デプロイメントに JAX-RS アノテーションが存在すること。

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
	• org.jboss.resteasy.resteasy-jackson2-provider • org.jboss.resteasy.resteasy-jsapi • org.jboss.resteasy.resteasy-json-p-provider • org.jboss.resteasy.resteasy-multipart-provider • org.jboss.resteasy.resteasy-yaml-provider • org.codehaus.jackson.jackson-core-asl		
JCA	• javax.resource.api	• javax.jms.api • javax.validation.api • org.jboss.ironjacamar.api • org.jboss.ironjacamar.impl • org.hibernate.validator	リソースアダプター (RAR) アーカイブのデプロイメント。

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
JPA (Hibernate)	• **javax.persistence.api**	• **org.jboss.as.jpa** • **org.jboss.as.jpa.spi** • **org.javassist**	**@PersistenceUnit** または **@PersistenceContext** アノテーションが存在するか、デプロイメント記述子に **<persistence-unit-ref>** または **<persistence-context-ref>** 要素が存在すること。 JBoss EAP は永続プロバイダー名をモジュール名にマップします。**persistence.xml** ファイルで特定のプロバイダーに名前を付けると、適切なモジュールに対して依存関係が追加されます。これが希望の挙動ではない場合は、**jboss-deployment-structure.xml** を使用して除外できます。
JSF (Java Server Faces)		• **javax.faces.api** • **com.sun.jsf-impl** • **org.jboss.as.jsf** • **org.jboss.as.jsf-injection**	EAR アプリケーションに追加されます。 値が **true** の **org.jboss.jbossfaces.WAR_BUNDLES_JSF_IMPL** の **context-param** を **web.xml** ファイルで指定しない場合のみ WAR アプリケーションに追加されます。
JSR-77	• **javax.management.j2ee.api**		

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
ロギング	• `org.jboss.logging` • `org.apache.commons.logging` • `org.apache.log4j` • `org.slf4j` • `org.jboss.logging.jul-to-slf4j-stub`		
メール	• `javax.mail.api` • `javax.activation.api`		
メッセージング	• `javax.jms.api`	• `org.wildfly.extension.messaging-activemq`	
PicketLink Federation		• `org.picketlink`	
Pojo	• `org.jboss.as.pojo`		
SAR		• `org.jboss.modules` • `org.jboss.as.system-jmx` • `org.jboss.common-beans`	`jboss-service.xml` を含む SAR アーカイブのデプロイメント。

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
Seam2		• `org.jboss.vfs`	.
セキュリティー	• `org.picketbox` • `org.jboss.as.security` • `javax.security.jacc.api` • `javax.security.auth.message.api`		
ServiceActivator		• `org.jboss.msc`	
トランザクション	• `javax.transaction.api`	• `org.jboss.xts` • `org.jboss.jts` • `org.jboss.narayana.compensations`	

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
Undertow	• `javax.servlet.jstl.api` • `javax.servlet.api` • `javax.servlet.jsp.api` • `javax.websocket.api`	• `io.undertow.core` • `io.undertow.servlet` • `io.undertow.jsp` • `io.undertow.websocket` • `io.undertow.js` • `org.wildfly.clustering.web.api`	
Web Services	• `javax.jws.api` • `javax.xml.soap.api` • `javax.xml.ws.api`	• `org.jboss.ws.api` • `org.jboss.ws.spi`	アプリケーションクライアントタイプでない場合は、条件付き依存関係が追加されます。

依存関係を追加するサブシステム	常に追加されるパッケージ依存関係	条件的に追加されるパッケージ依存関係	依存関係の追加を引き起こす条件
Weld (CDI)	• `javax.enterprise.api` • `javax.inject.api`	• `javax.persistence.api` • `org.javassist` • `org.jboss.as.weld` • `org.jboss.weld.core` • `org.jboss.weld.probe` • `org.jboss.weld.api` • `org.jboss.weld.spi` • `org.hibernate.validator.cdi`	デプロイメントに `beans.xml` ファイルが存在すること。

3.9.2. 含まれるモジュール

含まれるモジュールの完全なリストとこれらのモジュールがサポートされているかについては、Red Hat カスタマーポータルの Red Hat JBoss Enterprise Application Platform 7 Included Modules を参照してください。

3.9.3. JBoss デプロイメント構造のデプロイメント記述子

このデプロイメント記述子を使用して実行できる主なタスクは次のとおりです。

- 明示的なモジュール依存関係を定義する。
- 特定の暗黙的な依存関係がロードされないようにする。
- デプロイメントのリソースより追加モジュールを定義する。
- EAR デプロイメントのサブデプロイメント分離の挙動を変更する。
- EAR のモジュールに追加のリソースルートを追加する。

第4章 ロギング

4.1. ロギング

ロギングとはアクティビティの記録 (ログ) を提供するアプリケーションからのメッセージ群を記録することです。

ログメッセージは、アプリケーションをデバッグする開発者や実稼働環境のアプリケーションを維持するシステム管理者に対して重要な情報を提供します。

ほとんどの最新の Java のロギングフレームワークには、正確な時間やメッセージの発信元などの詳細も含まれます。

4.1.1. サポート対象のアプリケーションロギングフレームワーク

JBoss LogManager は次のロギングフレームワークをサポートします。

- JBoss Logging (JBoss EAP に含まれる)
- Apache Commons Logging
- Simple Logging Facade for Java (SLF4J)
- Apache log4j
- Java SE Logging (java.util.logging)

JBoss LogManager では以下の API がサポートされます。

- JBoss Logging
- commons-logging
- SLF4J
- Log4j
- java.util.logging

JBoss LogManager では以下の SPI もサポートされます。

- java.util.logging Handler
- Log4j Appender

注記

`Log4j API` と **`Log4J Appender`** を使用している場合、オブジェクトは渡される前に **`string`** に変換されます。

4.2. JJBOSS LOGGING FRAMEWORK を用いたロギング

4.2.1. JBoss Logging について

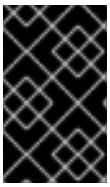
JBoss Logging は、JBoss EAP に含まれるアプリケーションロギングフレームワークです。JBoss Logging を使用すると、簡単にロギングをアプリケーションに追加できます。また、フレームワークを使用するアプリケーションにコードを追加し、定義された形式でログメッセージを送信できます。アプリケーションサーバーにアプリケーションがデプロイされると、これらのメッセージをサーバーでキャプチャーしたり、サーバーの設定に基づいて表示したり、ファイルに書き込んだりできます。

JBoss Logging では次の機能が提供されます。

- 革新的で使いやすい型指定されたロガー。型指定されたロガーは **`org.jboss.logging.annotations.MessageLogger`** でアノテートされたロガーインターフェースです。例については、国際化されたロガー、メッセージ、例外の作成を参照してください。
- 国際化およびローカリゼーションの完全なサポート。翻訳者は properties ファイルのメッセージバンドルを、開発者はインターフェースやアノテーションを使い作業を行います。詳細については、国際化と現地語化を参照してください。
- 実稼働用の型指定されたロガーを生成し、開発用の型指定されたロガーを実行時に生成する構築時ツール。

4.2.2. JBoss Logging を使用したアプリケーションへのロギングの追加

この手順では、JBoss Logging を使用してアプリケーションにロギングを追加する方法を示します。



重要

Maven を使用してプロジェクトをビルドする場合は、JBoss EAP Maven リポジトリーを使用するよう Maven を設定する必要があります。詳細については、JBoss EAP Maven リポジトリーの設定を参照してください。

1. JBoss Logging JAR ファイルがアプリケーションのビルドパスに指定されている必要があります。
   - Red Hat JBoss Developer Studio を使用してビルドする場合は、**`Project`** メニューから **`Properties`** を選択し、**`Targeted Runtimes`** を選択して JBoss EAP のランタイムにチェックが付けられていることを確認します。
   - Maven を使用してプロジェクトをビルドする場合は、**JBoss Logging** フレームワークにアクセスするために **`jboss-logging`** 依存関係をプロジェクトの **`pom.xml`** ファイルに追加します。

     ```
     <dependency>
       <groupId>org.jboss.logging</groupId>
       <artifactId>jboss-logging</artifactId>
       <version>3.3.0.Final-redhat-1</version>
       <scope>provided</scope>
     </dependency>
     ```

     jboss-javaee-7.0 BOM は **`jboss-logging`** のバージョンを管理します。詳細については、プロジェクト依存関係の管理を参照してください。アプリケーションでのロギングの例については、**`logging`** クイックスタートを参照してください。

   JAR は、JBoss EAP がデプロイされたアプリケーションに提供するため、ビルドされたアプリケーションに含める必要はありません。

2. ロギングを追加する各クラスに対して、以下の手順を実行します。

   a. 使用する JBoss Logging クラスネームスペースに対して import ステートメントを追加します。少なくとも、以下の import ステートメントが必要です。

   ```
   import org.jboss.logging.Logger;
   ```

   b. **org.jboss.logging.Logger** のインスタンスを作成し、静的メソッド **Logger.getLogger(Class)** を呼び出して初期化します。各クラスに対してこれを単一のインスタンス変数として作成することが推奨されます。

   ```
   private static final Logger LOGGER =
   Logger.getLogger(HelloWorld.class);
   ```

3. ログメッセージを送信するコードの **Logger** オブジェクトメソッドを呼び出します。**Logger** には、異なるタイプのメッセージに対して異なるパラメーターを持つさまざまなメソッドがあります。以下のメソッドを使用して対応するログレベルのログメッセージと **message** パラメーターを文字列として送信します。

   ```
   LOGGER.debug("This is a debugging message.");
   LOGGER.info("This is an informational message.");
   LOGGER.error("Configuration file not found.");
   LOGGER.trace("This is a trace message.");
   LOGGER.fatal("A fatal error occurred.");
   ```

   JBoss Logging メソッドの完全なリストについては、Logging API ドキュメンテーションを参照してください。

JBoss Logging の例

次の例では、プロパティーファイルからアプリケーションのカスタマイズされた設定がロードされます。指定されたファイルが見つからない場合は、**ERROR** レベルログメッセージが記録されます。

```
import org.jboss.logging.Logger;
public class LocalSystemConfig
{
   private static final Logger LOGGER =
Logger.getLogger(LocalSystemConfig.class);

   public Properties openCustomProperties(String configname) throws
CustomConfigFileNotFoundException
   {
      Properties props = new Properties();
      try
      {
         LOGGER.info("Loading custom configuration from "+configname);
         props.load(new FileInputStream(configname));
      }
      catch(IOException e) //catch exception in case properties file does
not exist
      {
         LOGGER.error("Custom configuration file ("+configname+") not
found. Using defaults.");
         throw new CustomConfigFileNotFoundException(configname);
      }
```

```
        return props;
    }
}
```

4.3. デプロイメントごとのロギング

デプロイメントごとのロギングを使用すると、開発者はアプリケーションのロギング設定を事前に設定できます。アプリケーションがデプロイされると、定義された設定に従ってロギングが開始されます。この設定によって作成されたログファイルにはアプリケーションの動作に関する情報のみが含まれます。

注記

デプロイメントごとのロギング設定が行われない場合、すべてのアプリケーションとサーバーには **logging** サブシステムの設定が使用されます。

この方法では、システム全体のロギングを使用する利点と欠点があります。利点は、JBoss EAP インスタンスの管理者がサーバーロギング以外のロギングを設定する必要がないことです。欠点は、デプロイメントごとのロギング設定はサーバーの起動時に読み取り専用であるため、実行時に変更できないことです。

4.3.1. デプロイメントごとのロギングをアプリケーションに追加

アプリケーションへのデプロイメントごとのロギングを設定するには、**logging.properties** 設定ファイルをデプロイメントに追加します。この設定ファイルは、JBoss Log Manager が基礎となるログマネージャーであるどのロギングファサードとも使用できるため、推奨されます。

設定ファイルが追加されるディレクトリーは、デプロイメント方法によって異なります。

- EAR デプロイメントの場合は、ロギング設定ファイルを **META-INF** ディレクトリーにコピーします。
- WAR または JAR デプロイメントの場合は、ロギング設定ファイルを **WEB-INF/classes** ディレクトリーにコピーします。

注記

Simple Logging Facade for Java (SLF4J) または **Apache log4j** を使用している場合は、**logging.properties** 設定ファイルが適しています。Apache log4j アペンダーを使用している場合は、**log4j.properties** 設定ファイルが必要になります。**jboss-logging.properties** 設定ファイルはレガシーデプロイメントのみでサポートされます。

logging.properties の設定
logging.properties ファイルはサーバーが起動し、**logging** サブシステムが起動するまで使用されます。**logging** サブシステムが設定に含まれない場合、サーバーはこのファイルの設定をサーバー全体のロギング設定として使用します。

JBoss ログマネージャーの設定オプション

ロガーオプション

- **loggers=<category>[,<category>,…]** - 設定するロガーカテゴリーのコンマ区切りのリストを指定します。ここにリストされていないカテゴリーは、以下のプロパティーから設定されません。
- **logger.<category>.level=<level>** - カテゴリーのレベルを指定します。このレベルは有効なレベルのいずれかになります。指定されない場合は、最も近い親のレベルが継承されます。
- **logger.<category>.handlers=<handler>[,<handler>,…]** - このロガーに割り当てるハンドラー名のコンマ区切りのリストを指定します。ハンドラーは、同じプロパティーファイルに設定する必要があります。
- **logger.<category>.filter=<filter>** - カテゴリーのフィルターを指定します。
- **logger.<category>.useParentHandlers=(true|false)** - ログメッセージを親ハンドラーにカスケードするかどうかを指定します。デフォルト値は **true** です。

ハンドラーオプション

- **handler.<name>=<className>** - インスタンス化するハンドラーのクラス名を指定します。このオプションは必須です。
- **handler.<name>.level=<level>** - このハンドラーのレベルを制限します。指定されない場合は、ALL のデフォルト値が保持されます。
- **handler.<name>.encoding=<encoding>** - 文字エンコーディングを指定します (このハンドラータイプによりサポートされている場合)。指定されない場合は、ハンドラー固有のデフォルト値が使用されます。
- **handler.<name>.errorManager=<name>** - 使用するエラーマネージャーの名前を指定します。エラーマネージャーは同じプロパティーファイルで設定する必要があります。指定されない場合は、エラーマネージャーが設定されません。
- **handler.<name>.filter=<name>** - カテゴリーのフィルターを指定します。フィルターの定義の詳細については、フィルター式を参照してください。
- **handler.<name>.formatter=<name>** - 使用するフォーマッターの名前を指定します (このハンドラータイプによりサポートされている場合)。フォーマッターは同じプロパティーファイルで設定する必要があります。指定されない場合、ほとんどのハンドラータイプのメッセージはログに記録されません。
- **handler.<name>.properties=<property>[,<property>,…]** - 追加的に設定する JavaBean 形式のプロパティーを指定します。該当するプロパティーが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
- **handler.<name>.constructorProperties=<property>[,<property>,…]** - 構築パラメーターとして使用する必要があるプロパティーのリストを指定します。該当するプロパティーが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
- **handler.<name>.<property>=<value>** - 名前付きプロパティーの値を設定します。

詳細については、JBoss EAP **設定ガイド**のログハンドラー属性を参照してください。

エラーマネージャーオプション

- **errorManager.<name>=<className>** - インスタンス化するエラーマネージャーのクラス名を指定します。このオプションは必須です。
- **errorManager.<name>.properties=<property>[,<property>,…]** - 追加的に設定する JavaBean 形式のプロパティーを指定します。該当するプロパティーが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
- **errorManager.<name>.<property>=<value>** - 名前付きプロパティーの値を設定します。

フォーマッターオプション

- **formatter.<name>=<className>** - インスタンス化するフォーマッターのクラス名を指定します。このオプションは必須です。
- **formatter.<name>.properties=<property>[,<property>,…]** - 追加的に設定する JavaBean 形式のプロパティーを指定します。該当するプロパティーが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
- **formatter.<name>.constructorProperties=<property>[,<property>,…]** - 構築パラメーターとして使用する必要があるプロパティーのリストを指定します。該当するプロパティーが適切に変換されるように、基本的なタイプイントロスペクションが行われます。
- **formatter.<name>.<property>=<value>** - 名前付きプロパティーの値を設定します。

以下の例は、コンソールにログ記録する **logging.properties** ファイルの最低限の設定を示しています。

```
# Additional logger names to configure (root logger is always configured)
# loggers=

# Root logger level
logger.level=INFO

# Root logger handlers
logger.handlers=CONSOLE

# Console handler configuration
handler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandler
handler.CONSOLE.properties=autoFlush
handler.CONSOLE.autoFlush=true
handler.CONSOLE.formatter=PATTERN

# Formatter pattern configuration
formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.pattern=%K{level}%d{HH:mm:ss,SSS} %-5p %C.%M(%L) [%c]
%s%e%n
```

4.4. ロギングプロファイル

ロギングプロファイルは、デプロイされたアプリケーションに割り当てることができる独立したロギング設定のセットです。通常の **logging** サブシステム同様にロギングプロファイルはハンドラー、カテゴリー、およびルートロガーを定義できますが、他のプロファイルや主要な **logging** サブシステムを

参照できません。設定が容易である点でロギングプロファイルは **logging** サブシステムと似ています。

ログインプロファイルを使用すると、管理者は他のロギング設定に影響を与えずに 1 つ以上のアプリケーションに固有なロギング設定を作成することができます。各プロファイルはサーバー設定で定義されるため、影響を受けるアプリケーションを再デプロイせずに、ロギング設定を変更できます。ただし、ロギングプロファイルは管理コンソールを使用して設定できません。詳細については、JBoss EAP **Configuration Guide** の Configure a Logging Profile を参照してください。

各ロギングプロファイルには以下の項目を設定できます。

- 一意な名前 (必須)
- 任意の数のログハンドラー
- 任意の数のログカテゴリー
- 最大 1 つのルートロガー

アプリケーションでは **Logging-Profile** 属性を使用して、**MANIFEST.MF** ファイルで使用するロギングプロファイルを指定できます。

4.4.1. アプリケーションでのロギングプロファイルの指定

アプリケーションでは、使用するロギングプロファイルを **MANIFEST.MF** ファイルで指定できます。

注記

このアプリケーションが使用するサーバー上に設定されたロギングプロファイルの名前を知っている必要があります。

ロギングプロファイル設定をアプリケーションに追加するには、**MANIFEST.MF** ファイルを編集します。

- アプリケーションに **MANIFEST.MF** ファイルがない場合は、ロギングプロファイル名を指定する以下の内容が含まれるファイルを作成します。

```
Manifest-Version: 1.0
Logging-Profile: LOGGING_PROFILE_NAME
```

- アプリケーションに **MANIFEST.MF** ファイルがすでにある場合は、ロギングプロファイル名を指定する以下の行を追加します。

```
Logging-Profile: LOGGING_PROFILE_NAME
```

注記

Maven および **maven-war-plugin** を使用している場合は、**MANIFEST.MF** ファイルを **src/main/resources/META-INF/** に置き、次の設定を **pom.xml** ファイルに追加します。

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <archive>
      <manifestFile>src/main/resources/META-
INF/MANIFEST.MF</manifestFile>
    </archive>
  </configuration>
</plugin>
```

アプリケーションがデプロイされると、ログメッセージに対して指定されたロギングプロファイルの設定が使用されます。

ロギングプロファイルとアプリケーションの設定方法の例については、JBoss EAP **Configuration Guide** の Example Logging Profile Configuration を参照してください。

4.5. 国際化と現地語化

4.5.1. はじめに

4.5.1.1. 国際化

国際化とは、技術的な変更を行わずに異なる言語や地域に対してソフトウェアを適合させるソフトウェア設計のプロセスのことです。

4.5.1.2. 多言語化

多言語化とは、特定の地域や言語に対してロケール固有のコンポーネントやテキストの翻訳を追加することで、国際化されたソフトウェアを適合させるプロセスのことです。

4.5.2. JBoss Logging Tools の国際化および現地語化

JBoss Logging Tools は、ログメッセージ、例外メッセージ、および汎用文字列の国際化や現地語化のサポートを提供する Java API です。JBoss Logging Tools は翻訳のメカニズムを提供するだけでなく、各ログメッセージに対して一意な識別子のサポートも提供します。

国際化されたメッセージと例外は、**org.jboss.logging.annotations** アノテーションが付けられたインターフェース内でメソッド定義として作成されます。インターフェースを実装する必要はありません。JBoss Logging Tools がコンパイル時にインターフェースを実装します。定義すると、これらのメソッドを使用してコードでメッセージをログに記録したり、例外オブジェクトを取得したりできます。

JBoss Logging Tools によって作成される国際化されたロギングインターフェースや例外インターフェースは、特定の言語や地域に対する翻訳が含まれる各バンドルのプロパティーファイルを作成して現地語化されます。JBoss Logging Tools は、トランスレーターが編集できる各バンドル対してテンプレートプロパティーファイルを生成できます。

JBoss Logging Tools は、プロジェクトの対象翻訳プロパティーファイルごとに各バンドルの実装を作成します。必要なのはバンドルに定義されているメソッドを使用することのみで、JBoss Logging Tools は現在の地域設定に対して正しい実装が呼び出されるようにします。

メッセージ ID とプロジェクトコードは各ログメッセージの前に付けられる一意の識別子です。この一意の識別子をドキュメントで使用すると、ログメッセージの情報を簡単に検索することができます。適切なドキュメントでは、メッセージが書かれた言語に関係なく、ログメッセージの意味を識別子から判断できます。

JBoss Logging Tools には次の機能のサポートが含まれます。

MessageLogger

`org.jboss.logging.annotations` パッケージ内のこのインターフェースは、国際化されたログメッセージを定義するために使用されます。メッセージロガーインターフェースは `@MessageLogger` でアノテートされます。

MessageBundle

このインターフェースは、翻訳可能な汎用メッセージと国際化されたメッセージが含まれる例外オブジェクトを定義するために使用できます。メッセージバンドルインターフェースは、`@MessageBundle` でアノテートされます。

国際化されたログメッセージ

これらのログメッセージは、`MessageLogger` のメソッドを定義して作成されます。メソッドは `@LogMessage` アノテーションと `@Message` アノテーションを付け、`@Message` の値属性を使用してログメッセージを指定する必要があります。国際化されたログメッセージはプロパティーファイルで翻訳を提供することによりローカライズされます。
JBoss Logging Tools はコンパイル時に各翻訳に必要なロギングクラスを生成し、ランタイム時に現ロケールに対して適切なメソッドを呼び出します。

国際化された例外

国際化された例外は、MessageBundle で定義されたメソッドから返された例外オブジェクトです。これらのメッセージバンドルは、デフォルトの例外メッセージを定義するためにアノテートできます。デフォルトのメッセージは、現在のロケールと一致するプロパティーファイルに翻訳がある場合にその翻訳に置き換えられます。国際化された例外にも、プロジェクトコードとメッセージ ID を割り当てることができます。

国際化されたメッセージ

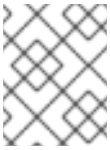
国際化されたメッセージは、`MessageBundle` で定義されたメソッドから返された文字列です。Java String オブジェクトを返すメッセージバンドルメソッドは、その文字列のデフォルトの内容 (メッセージと呼ばれます) を定義するためにアノテートできます。デフォルトのメッセージは、現在のロケールと一致するプロパティーファイルに翻訳がある場合にその翻訳に置き換えられます。

翻訳プロパティーファイル

翻訳プロパティーファイルは、1 つのロケール、国、バリアントに対する 1 つのインターフェースのメッセージの翻訳が含まれる Java プロパティーファイルです。翻訳プロパティーファイルは、メッセージを返すクラスを生成するために JBoss Logging Tools によって使用されます。

JBoss Logging Tools のプロジェクトコード

プロジェクトコードはメッセージのグループを識別する文字列です。プロジェクトコードは各ログメッセージの最初に表示され、メッセージ ID の前に付けられます。プロジェクトコードは `@MessageLogger` アノテーションの projectCode 属性で定義されます。



注記

新しいログメッセージプロジェクトコード接頭辞の完全なリストについては、JBoss EAP 7.0 で使用されているプロジェクトコードを参照してください。

JBoss Logging Tools のメッセージ ID

メッセージ ID はプロジェクトコードと組み合わせてログメッセージを一意に識別する数字です。メッセージ ID は各ログメッセージの最初に表示され、メッセージのプロジェクトコードの後に付けられます。メッセージ ID は **@Message** アノテーションの ID 属性で定義されます。

JBoss EAP に同梱される **logging-tools** クイックスタートは、JBoss Logging Tools の多くの機能の例を提供する単純な Maven プロジェクトです。以降のコード例は、**logging-tools** クイックスタートから取得されました。

4.5.3. 国際化されたロガー、メッセージ、例外の作成

4.5.3.1. 国際化されたログメッセージの作成

JBoss Logging Tools を使用して **MessageLogger** インターフェースを作成することにより、国際化されたログメッセージを作成できます。

注記

このトピックでは、ログメッセージのすべてのオプション機能または国際化について説明しません。

1. JBoss EAP Maven レポジトリーを使用するよう Maven を設定します (まだそのように設定していない場合)。
   詳細については、Maven 設定を使用した JBoss EAP Maven リポジトリーの設定を参照してください。

2. JBoss Logging Tools を使用するようプロジェクトの **pom.xml** ファイルを設定します。
   詳細については、JBoss Logging Tools の Maven 設定を参照してください。

3. ログメッセージ定義を含めるために Java インターフェースをプロジェクトに追加して、メッセージロガーインターフェースを作成します。
   定義するログメッセージの内容がわかるようインターフェースに名前を付けます。ログメッセージインターフェースの要件は次のとおりです。

   - **@org.jboss.logging.annotations.MessageLogger** でアノテートする必要があります。
   - オプションで、**org.jboss.logging.BasicLogger** を拡張できます。
   - インターフェースと同じ型のメッセージロガーであるフィールドをインターフェースで定義する必要があります。これは、**@org.jboss.logging.Logger** の **getMessageLogger()** を使用して行います。

   ロガーインターフェースのコード例

```
package com.company.accounts.loggers;

import org.jboss.logging.BasicLogger;
import org.jboss.logging.Logger;
import org.jboss.logging.annotations.MessageLogger;

@MessageLogger(projectCode="")
interface AccountsLogger extends BasicLogger {
   AccountsLogger LOGGER = Logger.getMessageLogger(
```

```
            AccountsLogger.class,
            AccountsLogger.class.getPackage().getName() );
}
```

4. 各ログメッセージのインターフェースにメソッド定義を追加します。
ログメッセージの各メソッドにその内容を表す名前を付けます。各メソッドの要件は次のとおりです。

   - メソッドは **void** を返す必要があります。
   - **@org.jboss.logging.annotation.LogMessage** アノテーションでアノテートする必要があります。
   - **@org.jboss.logging.annotations.Message** アノテーションでアノテートする必要があります。
   - デフォルトのログレベルは **INFO** です。
   - **@org.jboss.logging.annotations.Message** の値属性にはデフォルトのログインメッセージが含まれます。このメッセージは翻訳がない場合に使用されます。

   ```
   @LogMessage
   @Message(value = "Customer query failed, Database not
   available.")
   void customerQueryFailDBClosed();
   ```

5. メッセージをログに記録する必要があるコードで呼び出しをインターフェースメソッドに追加してメソッドを呼び出します。
インターフェースの実装を作成する必要はありません。これは、プロジェクトがコンパイルされる時にアノテーションプロセッサーにより行われます。

   ```
   AccountsLogger.LOGGER.customerQueryFailDBClosed();
   ```

   カスタムのロガーは **BasicLogger** からサブクラス化されるため、**BasicLogger** のロギングメソッドを使用することもできます。国際化されていないメッセージをログに記録するために他のロガーを作成する必要はありません。

   ```
   AccountsLogger.LOGGER.error("Invalid query syntax.");
   ```

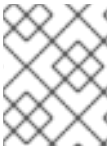
6. プロジェクトで、現地語化できる 1 つ以上の国際化されたロガーがサポートされるようになります。

注記

JBoss EAP に同梱される **logging-tools** クイックスタートは、JBoss Logging Tools の使用例を提供する単純な Maven プロジェクトです。

4.5.3.2. 国際化されたメッセージの作成と使用

この手順では、国際化された例外を作成および使用する方法を示します。



注記

本項では、これらのメッセージの現地語化に関するすべてのオプション機能またはプロセスについて説明しません。

1. JBoss EAP Maven レポジトリーを使用するよう Maven を設定します (まだそのように設定していない場合)。詳細については、Maven 設定を使用した JBoss EAP Maven リポジトリーの設定を参照してください。

2. JBoss Logging Tools を使用するようプロジェクトの **pom.xml** ファイルを設定します。詳細については、JBoss Logging Tools の Maven 設定を参照してください。

3. 例外のインターフェースを作成します。JBoss Logging Tools はインターフェースで国際化されたメッセージを定義します。含まれるメッセージのインターフェースにその内容を表す名前を付けます。インターフェースの要件は以下のとおりです。

   - **public** として宣言する必要があります。
   - **@org.jboss.logging.annotations.MessageBundle** でアノテートする必要があります。
   - インターフェースと同じ型のメッセージバンドルであるフィールドをインターフェースが定義する必要があります。

   インターフェースのコード例

   ```
   @MessageBundle(projectCode="")
   public interface GreetingMessageBundle {
      GreetingMessageBundle MESSAGES =
   Messages.getBundle(GreetingMessageBundle.class);
   }
   ```

   注記

   Messages.getBundle(GreetingMessagesBundle.class) を呼び出すのは **Messages.getBundle(GreetingMessagesBundle.class, Locale.getDefault())** を呼び出すのと同等です。

   Locale.getDefault() は、Java Virtual Machine のこのインスタンスのデフォルトロケールに対する現在の値を取得します。起動時に、ホストの環境に基づいて Java Virtual Machine によりデフォルトのロケールが設定されます。これは、ロケールが明示的に指定されない場合に、ロケールに関連する多くのメソッドにより使用され、**setDefault** メソッドを使用して変更できます。

   See Set the Default Locale of the Server in the JBoss EAP **Configuration Guide** for more information.

4. 各メッセージのインターフェースにメソッド定義を追加します。メッセージに対する各メソッドにその内容を表す名前を付けます。各メソッドの要件は以下のとおりです。

   - 型 **String** のオブジェクトを返す必要があります。
   - **@org.jboss.logging.annotations.Message** アノテーションでアノテートする必要があります。

- デフォルトメッセージに **@org.jboss.logging.annotations.Message** の値属性を設定する必要があります。翻訳がない場合にこのメッセージが使用されます。

メソッド定義のコード例

```
@Message(value = "Hello world.")
String helloworldString();
```

5. メッセージを取得する必要があるアプリケーションでインターフェースメソッドを呼び出します。

メソッド呼び出しのコード例

```
System.out.println(helloworldString());
```

プロジェクトで、現地語化できる国際化されたメッセージ文字列がサポートされるようになります。

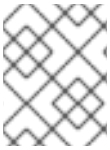
注記

使用できる完全な例については、JBoss EAP に同梱される **logging-tools** クイックスタートを参照してください。

4.5.3.3. 国際化された例外の作成

JBoss Logging Tools を使用して、国際化された例外を作成および使用できます。

以下の手順では、Red Hat JBoss Developer Studio または Maven のいずれかを使用してビルドされた既存のソフトウェアプロジェクトに、国際化された例外を追加することを前提としています。



注記

このトピックでは、これらの例外の国際化に関するすべてのオプション機能またはプロセスについて説明しません。

1. JBoss Logging Tools を使用するようプロジェクトの **pom.xml** ファイルを設定します。詳細については、JBoss Logging Tools の Maven 設定を参照してください。

2. 例外のインターフェースを作成します。JBoss Logging Tools はインターフェースで国際化されたメッセージを定義します。定義される例外のインターフェースにその内容を表す名前を付けます。インターフェースの要件は以下のとおりです。

   - **public** として宣言する必要があります。
   - **@MessageBundle** でアノテートする必要があります。
   - インターフェースと同じ型のメッセージバンドルであるフィールドをインターフェースが定義する必要があります。

```
@MessageBundle(projectCode="")
public interface ExceptionBundle {
   ExceptionBundle EXCEPTIONS =
Messages.getBundle(ExceptionBundle.class);
}
```

3. 各例外のインターフェースにメソッド定義を追加します。例外に対する各メソッドにその内容を表す名前を付けます。各メソッドの要件は以下のとおりです。

   - **Exception** オブジェクトまたは **Exception** のサブタイプを返す必要があります。
   - **@org.jboss.logging.annotations.Message** アノテーションでアノテートする必要があります。
   - デフォルトの例外メッセージに **@org.jboss.logging.annotations.Message** の値属性を設定する必要があります。このメッセージは翻訳がない場合に使用されます。
   - メッセージ文字列の他にパラメーターを必要とするコンストラクターが返される例外にある場合は、**@Param** アノテーションを使用してこれらのパラメーターをメソッド定義に提供する必要があります。パラメーターは、例外のコンストラクターと同じ型および順番である必要があります。

     ```
     @Message(value = "The config file could not be opened.")
     IOException configFileAccessError();

     @Message(id = 13230, value = "Date string '%s' was invalid.")
     ParseException dateWasInvalid(String dateString, @Param int
     errorOffset);
     ```

4. 例外を取得する必要があるコードでインターフェースメソッドを呼び出します。メソッドによって例外はスローされませんが、スローできる例外オブジェクトがメソッドによって返されます。

   ```
   try {
      propsInFile=new File(configname);
      props.load(new FileInputStream(propsInFile));
   }
   catch(IOException ioex) {
     //in case props file does not exist
      throw ExceptionBundle.EXCEPTIONS.configFileAccessError();
   }
   ```

プロジェクトで、現地語化できる国際化された例外がサポートされるようになります。

注記

使用できる完全な例については、JBoss EAP に同梱される **logging-tools** クイックスタートを参照してください。

4.5.4. 国際化されたロガー、メッセージ、例外の現地語化

4.5.4.1. Maven での新しい翻訳プロパティーファイルの作成

Maven で構築されたプロジェクトでは、含まれる **MessageLogger** と **MessageBundle** それぞれに対して空の翻訳プロパティーファイルを生成できます。これらのファイルは新しい翻訳プロパティーファイルとして使用することができます。

新しい翻訳プロパティーファイルを生成するよう Maven プロジェクトを設定する手順は次のとおりです。

前提条件

前提条件

- 作業用の Maven プロジェクトがすでに存在している必要があります。
- JBoss Logging Tools に対してプロジェクトが設定されていなければなりません。
- 国際化されたログメッセージや例外を定義する 1 つ以上のインターフェースがプロジェクトに含まれていなければなりません。

翻訳プロパティーファイルの生成

1. **-AgenereatedTranslationFilePath** コンパイラー引数を Maven コンパイラープラグイン設定に追加し、新しいファイルが作成されるパスを割り当てます。
この設定では、Maven プロジェクトの **target/generated-translation-files** ディレクトリーに新しいファイルが作成されます。

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>2.3.2</version>
   <configuration>
      <source>1.6</source>
      <target>1.6</target>
      <compilerArgument>
      -
AgeneratedTranslationFilesPath=${project.basedir}/target/generated-
translation-files
      </compilerArgument>
      <showDeprecation>true</showDeprecation>
   </configuration>
</plugin>
```

2. Maven を使用してプロジェクトをビルドします。

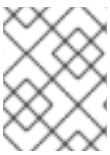
```
$ mvn compile
```

@MessageBundle または **@MessageLogger** でアノテートされた各インターフェースに対して 1 つのプロパティーファイルが作成されます。

- 各インターフェースが宣言された Java パッケージに対応するサブディレクトリーに新しいファイルが作成されます。
- 新しい各ファイルには、以下のパターンで名前が付けられます。ここで、**INTERFACE_NAME** はファイルを生成するために使用するインターフェースの名前です。

```
INTERFACE_NAME.i18n_locale_COUNTRY_VARIANT.properties
```

生成されたファイルは新しい翻訳の基礎としてプロジェクトにコピーできます。



注記

使用できる完全な例については、JBoss EAP に同梱される **logging-tools** クイックスタートを参照してください。

4.5.4.2. 国際化されたロガー、例外、またはメッセージの翻訳

プロパティーファイルは、JBoss Logging Tools を使用してインターフェースで定義されたロギングおよび例外メッセージの翻訳を提供します。

次の手順は、翻訳プロパティーファイルの作成方法と使用方法を示しています。この手順では、国際化された例外またはログメッセージに対して 1 つ以上のインターフェースがすでに定義されているプロジェクトが存在することを前提にしています。

前提条件

- 作業用の Maven プロジェクトがすでに存在している必要があります。
- JBoss Logging Tools に対してプロジェクトが設定されていなければなりません。
- 国際化されたログメッセージや例外を定義する 1 つ以上のインターフェースがプロジェクトに含まれていなければなりません。
- テンプレート翻訳プロパティーファイルを生成するようプロジェクトが設定されている必要があります。

国際化されたロガー、例外、またはメッセージの翻訳

1. 以下のコマンドを実行して、テンプレート翻訳プロパティーファイルを作成します。

```
$ mvn compile
```

2. 翻訳したいインターフェースのテンプレートを、テンプレートが作成されたディレクトリーからプロジェクトの **src/main/resources** ディレクトリーにコピーします。プロパティーファイルは翻訳するインターフェースと同じパッケージに存在する必要があります。
3. **GreeterLogger.i18n_fr_FR.properties** のように、含まれる言語を示すように、コピーされたテンプレートファイルの名前を変更します。
4. 新しい翻訳プロパティーファイルの内容を編集し、適切な翻訳が含まれるようにします。

```
# Level: Logger.Level.INFO
# Message: Hello message sent.
logHelloMessageSent=Bonjour message envoyé.
```

5. テンプレートをコピーし、バンドルの各翻訳のために変更するプロセスを繰り返します。

プロジェクトに 1 つ以上のメッセージバンドルまたはロガーバンドルに対する翻訳が含まれるようになります。プロジェクトをビルドすると、提供された翻訳が含まれるログメッセージに対して適切なクラスが生成されます。JBoss Logging Tools は、アプリケーションサーバーの現在のロケールに合わせて適切なクラスを自動的に使用するため、明示的にメソッドを呼び出したり、特定言語のパラメーターを提供したりする必要はありません。

生成されたクラスのソースコードは **target/generated-sources/annotations/** で確認できます。

4.5.5. 国際化されたログメッセージのカスタマイズ

4.5.5.1. ログメッセージへのメッセージ ID とプロジェクトコードの追加

この手順は、メッセージ ID とプロジェクトコードを JBoss Logging Tools を使用して作成された国際化済みログメッセージへ追加する方法を示しています。ログメッセージがログで表示されるようにするには、プロジェクトコードとメッセージ ID の両方が必要です。メッセージにプロジェクトコードとメッセージ ID の両方がない場合は、どちらも表示されません。

前提条件

1. 国際化されたログメッセージが含まれるプロジェクトが存在する必要があります。国際化されたログメッセージの作成を参照してください。
2. 使用するプロジェクトコードを知っている必要があります。プロジェクトコードを 1 つ使用することも、各インターフェースに異なる複数のコードを定義することも可能です。

ログメッセージへのメッセージ ID とプロジェクトコードの追加

1. カスタムのロガーインターフェースに付けられる **@MessageLogger** アノテーションの projectCode 属性を使用してプロジェクトコードを指定します。インターフェースに定義されるすべてのメッセージがこのプロジェクトコードを使用します。

```
@MessageLogger(projectCode="ACCNTS")
interface AccountsLogger extends BasicLogger {

}
```

2. メッセージを定義するメソッドに付けられる **@Message** アノテーションの **id** 属性を使用して、各メッセージのメッセージ ID を指定します。

```
@LogMessage
@Message(id=43, value = "Customer query failed, Database not
available.")  void customerQueryFailDBClosed();
```

3. メッセージ ID とプロジェクトコードの両方が関連付けられたログメッセージでは、メッセージ ID とプロジェクトコードがログに記録されたメッセージの前に付けられます。

```
10:55:50,638 INFO  [com.company.accounts.ejb] (MSC service thread 1-
4) ACCNTS000043: Customer query failed, Database not available.
```

4.5.5.2. メッセージのログレベル設定

JBoss Logging Tools のインターフェースによって定義されるメッセージのデフォルトのログレベルは **INFO** です。ロギングメソッドに付けられた **@LogMessage** アノテーションの **level** 属性を用いて異なるログレベルを指定することが可能です。異なるログレベルを指定するには、以下の手順を実行します。

1. ログメッセージメソッド定義の **@LogMessage** アノテーションに **level** 属性を追加します。
2. **level** 属性を使用してこのメッセージにログレベルを割り当てます。**level** の有効値は **org.jboss.logging.Logger.Level** で定義された **DEBUG**、**ERROR**、**FATAL**、**INFO**、**TRACE**、および **WARN** の 6 つの列挙定数です。

```
import org.jboss.logging.Logger.Level;

@LogMessage(level=Level.ERROR)
```

```
@Message(value = "Customer query failed, Database not available.")
void customerQueryFailDBClosed();
```

上記の例のロギングメソッドを呼び出すと、**ERROR** レベルのログメッセージが作成されます。

```
10:55:50,638 ERROR  [com.company.app.Main] (MSC service thread 1-4)
 Customer query failed, Database not available.
```

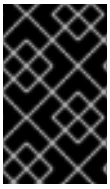
4.5.5.3. パラメーターによるログメッセージのカスタマイズ

カスタムのログインメソッドはパラメーターを定義できます。これらのパラメーターを使用してログメッセージに表示される追加情報を渡すことが可能です。ログメッセージでパラメーターが表示される場所は、明示的なインデクシングか通常のインデクシングを使用してメッセージ自体に指定されます。

パラメーターによるログメッセージのカスタマイズ

1. すべての型のパラメーターをメソッド定義に追加します。型に関係なくパラメーターの String 表現がメッセージに表示されます。
2. ログメッセージにパラメーター参照を追加します。参照は明示的なインデックスまたは通常のインデックスを使用できます。
   - 通常のインデックスを使用するには、各パラメーターを表示したいメッセージ文字列に **%s** 文字を挿入します。**%s** の最初のインスタンスにより最初のパラメーターが挿入され、2 番目のインスタンスにより 2 番目のパラメーターが挿入されます。
   - 明示的なインデックスを使用するには、文字 **%#$s** をメッセージに挿入します。ここで、# は表示したいパラメーターの数を示します。

明示的なインデックスを使用すると、メッセージのパラメーター参照の順番がメソッドで定義される順番とは異なるようになります。これは、異なるパラメーターの順番が必要になる可能性がある翻訳済みメッセージで重要になります。



重要

指定されたメッセージでは、パラメーターの数とパラメーターへの参照の数が同じでなければなりません。同じでないとコードがコンパイルされません。**@Cause** アノテーションが付けられたパラメーターはパラメーターの数には含まれません。

以下に、通常のインデックスを使用したメッセージパラメーターの例を示します。

```
@LogMessage(level=Logger.Level.DEBUG)
@Message(id=2, value="Customer query failed, customerid:%s, user:%s")
void customerLookupFailed(Long customerid, String username);
```

以下に、明示的なインデックスを使用したメッセージパラメーターの例を示します。

```
@LogMessage(level=Logger.Level.DEBUG)
@Message(id=2, value="Customer query failed, user:%2$s, customerid:%1$s")
void customerLookupFailed(Long customerid, String username);
```

4.5.5.4. 例外をログメッセージの原因として指定

JBoss Logging Tools では、カスタムログインメソッドのパラメーターの 1 つをメッセージの原因として定義することができます。定義するには、このパラメーターを **Throwable** 型またはいずれかのサブクラスにし、**@Cause** アノテーションを付ける必要があります。このパラメーターは、他のパラメーターのようにログメッセージで参照することはできず、ログメッセージの後に表示されます。

次の手順は、**@Cause** パラメーターを使用して「原因となる」例外を示し、ロギングメソッドを更新する方法を表しています。この機能に追加したい国際化されたロギングメッセージがすでに作成されていることを前提とします。

例外をログメッセージの原因として指定

1. **Throwable** 型のパラメーターまたはサブクラスをメソッドに追加します。

```
@LogMessage
@Message(id=404, value="Loading configuration failed. Config
file:%s")
void loadConfigFailed(Exception ex, File file);
```

2. パラメーターに **@Cause** アノテーションを追加します。

```
import org.jboss.logging.annotations.Cause

@LogMessage
@Message(value = "Loading configuration failed. Config file: %s")
void loadConfigFailed(@Cause Exception ex, File file);
```

3. メソッドを呼び出します。コードでメソッドが呼び出されると、正しい型のオブジェクトが渡され、ログメッセージの後に表示されます。

```
try
{
   confFile=new File(filename);
   props.load(new FileInputStream(confFile));
}
catch(Exception ex) //in case properties file cannot be read
{
     ConfigLogger.LOGGER.loadConfigFailed(ex, filename);
}
```

コードによって **FileNotFoundException** 型の例外が発生した場合、上記コード例の出力は次のようになります。

```
10:50:14,675 INFO [com.company.app.Main] (MSC service thread 1-3) Loading
configuration failed. Config file: customised.properties
java.io.FileNotFoundException: customised.properties (No such file or
directory)
   at java.io.FileInputStream.open(Native Method)
   at java.io.FileInputStream.<init>(FileInputStream.java:120)
   at com.company.app.demo.Main.openCustomProperties(Main.java:70)
   at com.company.app.Main.go(Main.java:53)
   at com.company.app.Main.main(Main.java:43)
```

4.5.6. 国際化された例外のカスタマイズ

4.5.6.1. メッセージ ID およびプロジェクトコードの例外メッセージへの追加

メッセージ ID およびプロジェクトコードは、国際化された例外によって表示される各メッセージの前に付けられる一意の識別子です。これらの識別コードによって、アプリケーションのすべての例外メッセージの参照を作成できるため、理解できない言語で書かれた例外メッセージの意味を検索できます。

以下の手順は、JBoss Logging Tools を使用して作成された国際化済み例外メッセージにメッセージ ID とプロジェクトコードを追加する方法を示しています。

前提条件

1. 国際化された例外が含まれるプロジェクトが存在する必要があります。詳細については、国際化された例外の作成を参照してください。
2. 使用するプロジェクトコードを知っている必要があります。プロジェクトコードを 1 つ使用することも、各インターフェースに異なる複数のコードを定義することも可能です。

メッセージ ID およびプロジェクトコードの例外メッセージへの追加

1. 例外バンドルインターフェースに付けられる **@MessageBundle** アノテーションの **projectCode** 属性を使用して、プロジェクトコードを指定します。インターフェースに定義されるすべてのメッセージがこのプロジェクトコードを使用します。

```
@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
   ExceptionBundle EXCEPTIONS =
Messages.getBundle(ExceptionBundle.class);
}
```

2. 例外を定義するメソッドに付けられる **@Message** アノテーションの **id** 属性を使用して、各例外に対してメッセージ ID を指定します。

```
@Message(id=143, value = "The config file could not be opened.")
IOException configFileAccessError();
```

重要

プロジェクトコードとメッセージ ID を両方持つメッセージでは、メッセージの前にプロジェクトコードとメッセージ ID が表示されます。プロジェクトコードとメッセージ ID の両方がない場合は、どちらも表示されません。

国際化された例外の例

以下の例外バンドルインターフェースの例では、プロジェクトコードが "ACCTS" であり、ID が "143" の例外メソッドが 1 つあります。

```
@MessageBundle(projectCode="ACCTS")
interface ExceptionBundle
{
    ExceptionBundle EXCEPTIONS =
Messages.getBundle(ExceptionBundle.class);
```

```
    @Message(id=143, value = "The config file could not be opened.")
    IOException configFileAccessError();
}
```

次のコードを使用すると、例外オブジェクトを取得およびスローできます。

```
throw ExceptionBundle.EXCEPTIONS.configFileAccessError();
```

これにより、次のような例外メッセージが表示されます。

```
Exception in thread "main" java.io.IOException: ACCTS000143: The config
file could not be opened.
at com.company.accounts.Main.openCustomProperties(Main.java:78)
at com.company.accounts.Main.go(Main.java:53)
at com.company.accounts.Main.main(Main.java:43)
```

4.5.6.2. パラメーターによる例外メッセージのカスタマイズ

例外を定義する例外バンドルメソッドでは、パラメーターを指定して例外メッセージに表示される追加情報を渡すことが可能です。例外メッセージでのパラメーターの正確な位置は、明示的なインデックスまたは通常のインデックスを使用してメッセージ自体に指定されます。

パラメーターによる例外メッセージのカスタマイズ

1. すべての型のパラメーターをメソッド定義に追加します。型に関係なくパラメーターの String 表現がメッセージに表示されます。
2. 例外メッセージにパラメーター参照を追加します。参照は明示的なインデックスまたは通常のインデックスを使用できます。
   - 通常のインデックスを使用するには、各パラメーターを表示したいメッセージ文字列に **%s** 文字を挿入します。**%s** の最初のインスタンスにより最初のパラメーターが挿入され、2 番目のインスタンスにより 2 番目のパラメーターが挿入されます。
   - 明示的なインデックスを使用するには、文字 **%#$s** をメッセージに挿入します。ここで、# は表示したいパラメーターの数を示します。

明示的なインデックスを使用すると、メッセージのパラメーター参照の順番がメソッドで定義される順番とは異なるようになります。これは、異なるパラメーターの順番が必要になる可能性がある翻訳済みメッセージで重要になります。

重要

指定されたメッセージでは、パラメーターの数とパラメーターへの参照の数が同じでなければなりません。同じでないとコードがコンパイルされません。**@Cause** アノテーションが付けられたパラメーターはパラメーターの数には含まれません。

以下に、通常のインデックスを使用したメッセージパラメーターの例を示します。

```
@Message(id=2, value="Customer query failed, customerid:%s, user:%s")
void customerLookupFailed(Long customerid, String username);
```

以下に、明示的なインデックスを使用したメッセージパラメーターの例を示します。

-

```
@Message(id=2, value="Customer query failed, user:%2$s, customerid:%1$s")
void customerLookupFailed(Long customerid, String username);
```

4.5.6.3. 別の例外の原因として 1 つの例外を指定

例外バンドルメソッドより返された例外に対し、他の例外を基礎となる原因として指定することができます。指定するには、パラメーターをメソッドに追加し、パラメーターに **@Cause** アノテーションを付けます。このパラメーターを使用して原因となる例外を渡します。このパラメーターを例外メッセージで参照することはできません。

次の手順は、**@Cause** パラメーターを使用して原因となる例外を示し、例外バンドルよりメソッドを更新する方法を表しています。この機能に追加したい国際化された例外バンドルがすでに作成されていることを前提とします。

1. **Throwable** 型のパラメーターまたはサブクラスをメソッドに追加します。

   ```
   @Message(id=328, value = "Error calculating: %s.")
   ArithmeticException calculationError(Throwable cause, String msg);
   ```

2. パラメーターに **@Cause** アノテーションを追加します。

   ```
   import org.jboss.logging.annotations.Cause

   @Message(id=328, value = "Error calculating: %s.")
   ArithmeticException calculationError(@Cause Throwable cause, String
   msg);
   ```

3. 例外オブジェクトを取得するため、インターフェースメソッドを呼び出します。キャッチした例外を原因として使用し、キャッチブロックより新しい例外を発生させるのが最も一般的なユースケースになります。

   ```
   try
   {
      ...
   }
   catch(Exception ex)
   {
      throw ExceptionBundle.EXCEPTIONS.calculationError(
                                          ex, "calculating payment due
   per day");
   }
   ```

以下に、例外を別の例外の原因として指定する例を示します。この例外バンドルでは、**ArithmeticException** 型の例外を返す単一のメソッドを定義します。

```
@MessageBundle(projectCode = "TPS")
interface CalcExceptionBundle
{
    CalcExceptionBundle EXCEPTIONS =
Messages.getBundle(CalcExceptionBundle.class);

    @Message(id=328, value = "Error calculating: %s.")
    ArithmeticException calcError(@Cause Throwable cause, String value);
}
```

このコード例では、整数のゼロ除算を実行しようとすると例外が発生する操作が実行されます。例外が捕捉され、その最初の例外を原因として使用して新しい例外が作成されます。

```
int totalDue = 5;
int daysToPay = 0;
int amountPerDay;

try
{
   amountPerDay = totalDue/daysToPay;
}
catch (Exception ex)
{
   throw CalcExceptionBundle.EXCEPTIONS.calcError(ex, "payments per day");
}
```

以下は、例外メッセージの例です。

```
Exception in thread "main" java.lang.ArithmeticException: TPS000328: Error
calculating: payments per day.
    at com.company.accounts.Main.go(Main.java:58)
    at com.company.accounts.Main.main(Main.java:43)
Caused by: java.lang.ArithmeticException: / by zero
    at com.company.accounts.Main.go(Main.java:54)
    ... 1 more
```

4.5.7. 参考資料

4.5.7.1. JBoss Logging Tools の Maven 設定

以下の手順では、国際化のために JBoss Logging と JBoss Logging Tools を使用するよう Maven プロジェクトを設定します。

1. JBoss EAP レポジトリーを使用するよう Maven を設定します (まだそのように設定していない場合)。詳細については、Maven 設定を使用した JBoss EAP Maven リポジトリーの設定を参照してください。
   pom.xml ファイルの **<dependencyManagement>** セクションに **jboss-eap-javaee7** BOM を含めます。

   ```
   <dependencyManagement>
     <dependencies>
       <!-- JBoss distributes a complete set of Java EE APIs including
         a Bill of Materials (BOM). A BOM specifies the versions of a
   "stack" (or
         a collection) of artifacts. We use this here so that we always
   get the correct versions of artifacts.
         Here we use the jboss-javaee-7.0 stack (you can
         read this as the JBoss stack of the Java EE APIs). You can
   actually
         use this stack with any version of JBoss EAP that implements
   Java EE. -->
       <dependency>
         <groupId>org.jboss.bom</groupId>
   ```

```
        <artifactId>jboss-eap-javaee7</artifactId>
        <version>${version.jboss.bom.eap}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    <dependencies>
<dependencyManagement>
```

2. Maven 依存関係をプロジェクトの **pom.xml** ファイルに追加します。

   a. JBoss Logging フレームワークにアクセスするために **jboss-logging** 依存関係を追加します。

   b. JBoss Logging Tools を使用する場合は、**jboss-logging-processor** 依存関係も追加します。
   これら両方の依存関係は、前の手順で追加された JBoss EAP BOM で利用できます。したがって、各依存関係のスコープ要素は示されているように **provided** に設定できます。

```
<!-- Add the JBoss Logging Tools dependencies -->
<!-- The jboss-logging API -->
<dependency>
   <groupId>org.jboss.logging</groupId>
   <artifactId>jboss-logging</artifactId>
   <scope>provided</scope>
</dependency>
<!-- Add the jboss-logging-tools processor if you are using JBoss
Tools  -->
<dependency>
   <groupId>org.jboss.logging</groupId>
   <artifactId>jboss-logging-processor</artifactId>
   <scope>provided</scope>
</dependency>
```

3. maven-compiler-plugin のバージョンは **3.1** 以上であり、**1.8** のターゲットソースおよび生成されたソースに対して設定する必要があります。

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.1</version>
   <configuration>
      <source>1.8</source>
      <target>1.8</target>
   </configuration>
</plugin>
```

注記

JBoss Logging Tools を使用するよう設定された **pom.xml** ファイルの完全な例については、JBoss EAP に同梱される **logging-tools** クイックスタートを参照してください。

4.5.7.2. 翻訳プロパティーファイルの形式

JBoss Logging Tools でのメッセージの翻訳に使用されるプロパティーファイルは標準的な Java プロパティーファイルです。このファイルの形式は、**java.util.Properties** クラスドキュメンテーションに記載されている単純な行指向の **key=value** ペア形式です。

ファイル名の形式は次のようになります。

```
InterfaceName.i18n_locale_COUNTRY_VARIANT.properties
```

- **InterfaceName** は翻訳が適用されるインターフェースの名前です。
- **locale**、**COUNTRY**、および **VARIANT** は翻訳が適用される地域設定を識別します。
- **locale** と **COUNTRY** は ISO-639 および ISO-3166 言語および国コードを使用して言語と国を指定します。**COUNTRY** は任意です。
- **VARIANT** は特定のオペレーティングシステムまたはブラウザーのみに適用される翻訳を識別するために使用できる任意の識別子です。

翻訳ファイルに含まれるプロパティーは翻訳されるインターフェースのメソッドの名前です。プロパティーに割り当てられた値が翻訳になります。メソッドがオーバーロードされる場合、これはドットとパラメーターの数を名前に付加することによって示されます。翻訳のメソッドは、異なる数のパラメーターを提供することによってのみオーバーロードできます。

翻訳プロパティーファイルの例

ファイル名: **GreeterService.i18n_fr_FR_POSIX.properties**

```
# Level: Logger.Level.INFO
# Message: Hello message sent.
logHelloMessageSent=Bonjour message envoyé.
```

4.5.7.3. JBoss Logging Tools のアノテーションに関するリファレンス

JBoss Logging では、ログメッセージや文字列、例外の国際化や現地語化に使用する以下のアノテーションが定義されています。

表4.1 JBoss Logging Tools のアノテーション

アノテーション	ターゲット	説明	属性
@MessageBundle	インターフェース	インターフェースをメッセージバンドルとして定義します。	**projectCode**
@MessageLogger	インターフェース	インターフェースをメッセージロガーとして定義します。	**projectCode**

アノテーション	ターゲット	説明	属性
@Message	メソッド	メッセージバンドルとメッセージロガーで使用できます。メッセージバンドルでは、現地語化された String または Exception オブジェクトを返すメソッドとして定義されます。メッセージロガーでは、現地語化されたロガーとしてメソッドが定義されます。	**value**、**id**
@LogMessage	メソッド	メッセージロガーのメソッドをロギングメソッドとして定義します。	**level** (デフォルト値は **INFO**)
@Cause	パラメーター	ログメッセージまたは他の例外が発生したときに例外を渡すパラメーターとして定義します。	-
@Param	パラメーター	例外のコンストラクターへ渡されるパラメーターとして定義します。	-

4.5.7.4. JBoss EAP で使用されるプロジェクトコード

以下の表は、 JBoss EAP 7.0 で使用されるすべてのプロジェクトコードとそのプロジェクトコードが属す Maven モジュールの一覧です。

表4.2 JBoss EAP で使用されるプロジェクトコード

Maven モジュール	プロジェクトコード
appclient	WFLYAC
batch/extension-jberet	WFLYBATCH
batch/extension	WFLYBATCH-DEPRECATED
batch/jberet	WFLYBAT
bean-validation	WFLYBV
controller-client	WFLYCC
controller	WFLYCTL
clustering/common	WFLYCLCOM
clustering/ejb/infinispan	WFLYCLEJBINF

Maven モジュール	プロジェクトコード
clustering/infinispan/extension	WFLYCLINF
clustering/jgroups/extension	WFLYCLJG
clustering/server	WFLYCLSV
clustering/web/infinispan	WFLYCLWEBINF
connector	WFLYJCA
deployment-repository	WFLYDR
deployment-scanner	WFLYDS
domain-http	WFLYDMHTTP
domain-management	WFLYDM
ee	WFLYEE
ejb3	WFLYEJB
embedded	WFLYEMB
host-controller	WFLYDC
host-controller	WFLYHC
iiop-openjdk	WFLYIIOP
io/subsystem	WFLYIO
jaxrs	WFLYRS
jdr	WFLYJDR
jmx	WFLYJMX
jpa/hibernate5	JIPI
jpa/spi/src/main/java/org/jipijapa/JipiLogger.java	JIPI
jpa/subsystem	WFLYJPA
jsf/subsystem	WFLYJSF

Maven モジュール	プロジェクトコード
jsr77	WFLYEEMGMT
launcher	WFLYLNCHR
legacy	WFLYORB
legacy	WFLYMSG
legacy	WFLYWEB
logging	WFLYLOG
mail	WFLYMAIL
management-client-content	WFLYCNT
messaging-activemq	WFLYMSGAMQ
mod_cluster/extension	WFLYMODCLS
naming	WFLYNAM
network	WFLYNET
patching	WFLYPAT
picketlink	WFLYPL
platform-mbean	WFLYPMB
pojo	WFLYPOJO
process-controller	WFLYPC
protocol	WFLYPRT
remoting	WFLYRMT
request-controller	WFLYREQCON
rts	WFLYRTS
sar	WFLYSAR

Maven モジュール	プロジェクトコード
security-manager	WFLYSM
security	WFLYSEC
server	WFLYSRV
system-jmx	WFLYSYSJMX
threads	WFLYTHR
transactions	WFLYTX
undertow	WFLYUT
webservices/server-integration	WFLYWS
weld	WFLYWELD
xts	WFLYXTS

第5章 リモート JNDI ルックアップ

5.1. JNDI へのオブジェクトの登録

Java Naming and Directory Interface (JNDI) は、Java ソフトウェアクライアントが名前でオブジェクトを検出およびルックアップすることを可能にするディレクトリーサービスの Java API です。

JNDI に登録されるオブジェクトがリモート JNDI クライアント (つまり、別の JVM で実行されるクライアント) によりルックアップされる場合は、オブジェクトを **java:jboss/exported** コンテキストで登録する必要があります。

たとえば、**messaging-activemq** サブシステムの JMS キューをリモート JNDI クライアントに公開する必要がある場合は、 JMS キューを **java:jboss/exported/jms/queue/myTestQueue** のように JNDI に登録する必要があります。リモート JNDI クライアントは、名前 **jms/queue/myTestQueue** で JMS キューをルックアップできます。

例: standalone-full(-ha).xml のキューの設定

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
  <server name="default">
    ...
    <jms-queue name="myTestQueue"
entries="java:jboss/exported/jms/queue/myTestQueue"/>
    ...
  </server>
</subsystem>
```

5.2. リモート JNDI の設定

リモート JNDI クライアントは接続し、JNDI からの名前でオブジェクトをルックアップできます。**jboss-client.jar** がクラスパスに指定されている必要があります。**jboss-client.jar** は **EAP_HOME/bin/client/jboss-client.jar** で利用できます。

以下の例は、リモート JNDI クライアントの JNDI から **myTestQueue** キューをルックアップする方法を示しています。

例: MDB リソースアダプターの設定

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
properties.put(Context.PROVIDER_URL, "http-remoting://<hostname>:8080");
context = new InitialContext(properties);
Queue myTestQueue = (Queue) context.lookup("jms/queue/myTestQueue");
```

第6章 WEB アプリケーションのクラスター化

6.1. セッションレプリケーション

6.1.1. HTTP セッションレプリケーション

セッションレプリケーションは、配布可能なアプリケーションのクライアントセッションが、クラスター内のノードのフェイルオーバーによって中断されないようにします。クラスター内の各ノードは実行中のセッションの情報を共有するため、ノードが消滅してもセッションを引き継くことができます。

セッションレプリケーションは、mod_cluster、mod_jk、mod_proxy、ISAPI、および NSAPI クラスターにより高可用性を確保する仕組みのことです。

6.1.2. アプリケーションにおけるセッションレプリケーションの有効化

JBoss EAP の高可用性 (HA) 機能を利用し、Web アプリケーションのクラスタリングを有効にするには、アプリケーションが配布可能になるよう設定する必要があります。

アプリケーションを配布可能にする

1. アプリケーションが配布可能であることを示します。アプリケーションが配布可能とマークされていない場合は、セッションが配布されません。アプリケーションの **web.xml** 記述子ファイルの **<web-app>** タグ内に **<distributable/>** 要素を追加します。

   例: 配布可能なアプリケーションの最低限の設定

   ```
   <?xml version="1.0"?>
   <web-app  xmlns="http://java.sun.com/xml/ns/j2ee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                                 http://java.sun.com/xml/ns/j2ee/web-
   app_3_0.xsd"
             version="3.0">

         <distributable/>

   </web-app>
   ```

2. 次に、必要な場合はデフォルトのレプリケーションの動作を変更します。セッションレプリケーションに影響する値を変更する場合は、アプリケーションの **WEB-INF/jboss-web.xml** ファイルにある **<jboss-web>** 内の **<replication-config>** 要素内でこれらの値をオーバーライドできます。該当する要素で、デフォルト値をオーバーライドする場合のみ値を含めます。

   例: <replication-config> 値

   ```
   <jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
   http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd">
      <replication-config>
   ```

```
        <replication-granularity>SESSION</replication-granularity>
    </replication-config>
</jboss-web>
```

<replication-granularity> パラメーターは、レプリケートされるデータの粒度を決定します。デフォルト値は **SESSION** ですが、**ATTRIBUTE** を設定すると、ほとんどの属性は変更されずにセッションのパフォーマンスを向上させることができます。

<replication-granularity> の有効値は以下のとおりです。

- **SESSION**: デフォルト値です。属性がダーティーである場合に、セッションオブジェクト全体がレプリケートされます。このポリシーは、オブジェクト参照が複数のセッション属性で共有される 場合に必要です。共有されるオブジェクト参照は、1 つのユニットでセッション全体がシリアライズされるため、リモートノードで維持されます。

- **ATTRIBUTE**: これは、セッションのダーティーな属性と一部のセッションデータ (最後にアクセスされたタイムスタンプなど) にのみに使用できる値です。

変更不能なセッション属性
JBoss EAP7 の場合、セッションレプリケーションはセッションが変更された場合、またはセッションの変更可能な属性がアクセスされた場合にトリガーされます。以下のいずれかの条件に該当しない限り、セッション属性は変更可能であると見なされます。

- 値が既知の変更不能な値である
  - **null**
  - **java.util.Collections.EMPTY_LIST**、**EMPTY_MAP**、**EMPTY_SET**
- 値の型がまたは既知の変更不能な型である、または既知の変更不能な型を実装する
  - **java.lang.Boolean**、**Character**、**Byte**、**Short**、**Integer**、**Long**、**Float**、**Double**
  - **java.lang.Class**、**Enum**、**StackTraceElement**、**String**
  - **java.io.File**、**java.nio.file.Path**
  - **java.math.BigDecimal**、**BigInteger**、**MathContext**
  - **java.net.Inet4Address**、**Inet6Address**、**InetSocketAddress**、**URI**、**URL**
  - **java.security.Permission**
  - **java.util.Currency**、**Locale**、**TimeZone**、**UUID**
  - **java.time.Clock**、**Duration**、**Instant**、**LocalDate**、**LocalDateTime**、**LocalTime**、**MonthDay**、**Period**、**Year**、**YearMonth**、**ZoneId**、**ZoneOffset**、**ZonedDateTime**
  - **java.time.chrono.ChronoLocalDate**、**Chronology**、**Era**
  - **java.time.format.DateTimeFormatter**、**DecimalStyle**
  - **java.time.temporal.TemporalField**、**TemporalUnit**、**ValueRange**、**WeekFields**

    - **java.time.zone.ZoneOffsetTransition**、**ZoneOffsetTransitionRule**、**ZoneRules**

- 値の型が以下のアノテートでアノテートされる
    - **@org.wildfly.clustering.web.annotation.Immutable**
    - **@net.jcip.annotations.Immutable**

6.2. HTTP セッションパッシベーションおよびアクティベーション

6.2.1. HTTP セッションパッシベーションおよびアクティベーション

パッシベーションとは、比較的利用されていないセッションをメモリーから削除し、永続ストレージへ保存することでメモリーの使用量を制御するプロセスのことです。

アクティベーションとは、パッシベートされたデータを永続ストレージから取得し、メモリーに戻すことです。

パッシベーションは HTTP セッションのライフタイムの異なるタイミングで実行されます。

- コンテナーが新規セッションの作成を要求するときに現在アクティブなセッションの数が設定上限を超えている場合、サーバーはセッションの一部をパッシベートして新規セッションのスペースを作成しようとします。
- Web アプリケーションがデプロイされ、他のサーバーでアクティブなセッションのバックアップコピーが、新たにデプロイされる Web アプリケーションのセッションマネージャーによって取得された場合、セッションはパッシベートされることがあります。

アクティブなセッションが設定可能な最大数を超えると、セッションはパッシベートされます。

セッションは常に LRU (Least Recently Used) アルゴリズムを使ってパッシベートされます。

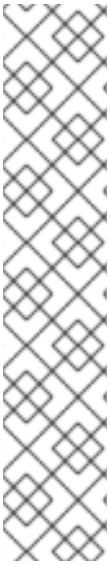
6.2.2. アプリケーションでの HTTP セッションパッシベーションの設定

HTTP セッションパッシベーションは、アプリケーションの **WEB-INF/jboss-web.xml** および **META-INF/jboss-web.xml** ファイルで設定されます。

例: jboss-web.xml ファイル

```
<jboss-web xmlns="http://www.jboss.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-web_10_0.xsd">

   <max-active-sessions>20</max-active-sessions>
</jboss-web>
```

<max-active-sessions/> 要素により、許可されるアクティブなセッションの最大数が設定され、セッションパッシベーションが有効になります。セッションの作成によって、アクティブなセッションの数が **<max-active-sessions/>** を超える場合は、新しいセッションのスペースを作成するために、セッションマネージャーが認識する最も古いセッションがパッシベートされます。



注記

メモリーのセッションの合計数には、このノードでアクセスされていない他のクラスターノードからレプリケートされたセッションが含まれます。これを考慮して **<max-active-sessions>** を設定してください。また、他のノードからレプリケートされるセッションの数は、**REPL** または **DIST** キャッシュモードが有効であるかどうかによっても異なります。**REPL** キャッシュモードでは、各セッションは各ノードにレプリケートされます。**DIST** キャッシュモードでは、各セッションは、**owners** パラメーターによって指定された数のノードにのみレプリケートされます。セッションキャッシュモードの設定については、JBoss EAP **Config Guide** の Configure the Cache Mode を参照してください。たとえば、各ノードが 100 ユーザーからの要求を処理する 8 つのノードクラスターがあるとします。この場合、**REPL** キャッシュモードでは、各ノードのメモリーに 800 のセッションが格納されます。**DIST** キャッシュモードが有効であり、デフォルトの **owners** 設定が **2** であるときは、各ノードのメモリーには 200 のセッションが格納されます。

6.3. クラスタリングサービスのパブリック API

JBoss EAP 7 には、アプリケーションが使用する改善されたパブリッククラスタリング API が導入されました。新しいサービスは、ライトウェイトで簡単に挿入できるよう設計されています。外部の依存関係は必要ありません。

`org.wildfly.clustering.group.Group`

グループサービスは、JGroups チャネルのクラスタートポロジーを参照し、トポロジーの変更時に通知するメカニズムを提供します。

```
@Resource(lookup = "java:jboss/clustering/group/channel-name")
private Group channelGroup;
```

`org.wildfly.clustering.dispatcher.CommandDispatcher`

`CommandDispatcherFactory` サービスは、クラスター内のノードでコマンドを実行するためにディスパッチャーを作成するメカニズムを提供します。結果として得られる **`CommandDispatcher`** は、以前の JBoss EAP リリースのリフレクションベースの **`GroupRpcDispatcher`** に類似したコマンドパターンです。

```
@Resource(lookup = "java:jboss/clustering/dispatcher/channel-name")
private CommandDispatcherFactory factory;

public void foo() {
    String context = "Hello world!";
    try (CommandDispatcher<String> dispatcher =
this.factory.createCommandDispatcher(context)) {
        dispatcher.executeOnCluster(new StdOutCommand());
    }
}

public static class StdOutCommand implements Command<Void, String> {
    @Override
    public Void execute(String context) {
        System.out.println(context);
        return null;
    }
}
```

6.4. HA シングルトンサービス

クラスター化されたシングルトンサービス (高可用性 (HA) シングルトンとも呼ばれます) は、クラスターの複数のノードにデプロイされるサービスです。このサービスはいずれかのノードでのみ提供されます。シングルトンサービスが実行されているノードは、通常**マスター**ノードと呼ばれます。

マスターノードが失敗またはシャットダウンした場合に、残りのノードから別のマスターが選択され、新しいマスターでサービスが再開されます。マスターが停止し、別のマスターが引き継ぐまでの短い間を除き、サービスは 1 つのノードのみによって提供されます。

HA シングルトン ServiceBuilder API

JBoss EAP 7 では、プロセスを大幅に簡略化するシングルトンサービスを構築するための新しいパブリック API が導入されます。

`SingletonServiceBuilder` 実装により、サービスは非同期的に開始されるようインストールされ、Modular Service Container (MSC) のデッドロックが回避されます。

HA シングルトンサービス選択ポリシー

ノードが ha-singleton を起動する優先度がある場合は、`ServiceActivator` クラスで選択ポリシーを設定できます。

JBoss EAP は、以下の 2 つの選択ポリシーを提供します。

1. 単純な選択ポリシー
   単純な選択ポリシーでは、相対的な経過時間に基づいてマスターノードが選択されます。必要な経過時間は、利用可能なノードのリストのインデックスである position プロパティーで設定されます。この場合は、以下のように設定されます。

   - position = 0 – 最も古いノードを参照する (デフォルト)
   - position = 1 – 2 番目に古いノードを参照する
     position をマイナスにして最も新しいノードを示すこともできます。
   - position = -1 – 最も新しいノードを参照する
   - position = -2 – 2 番目に新しいノードを参照する

2. ランダムな選択ポリシー
   ランダムな選択ポリシーでは、シングルトンサービスのプロバイダーとなるランダムなメンバーが選択されます。

HA シングルトンサービスアプリケーションの作成

以下に、アプリケーションを作成し、クラスター全体シングルトンサービスとしてデプロイするのに必要な手順の簡単な例を示します。この例のサービスにより、クラスターで 1 度だけ開始されるスケジュールタイマーがアクティブ化されます。

1. `org.jboss.msc.service.Service` インターフェースを実装し、**`getValue()`**、**`start()`**、および **`stop()`** メソッドを含む **`HATimerService`** サービスを作成します。

   サービスクラスコード例

```
public class HATimerService implements Service<String> {
    private static final Logger LOGGER =
Logger.getLogger(HATimerService.class.toString());
    public static final ServiceName SINGLETON_SERVICE_NAME =
```

```
ServiceName.JBOSS.append("quickstart", "ha", "singleton", "timer");

    /**
     * A flag whether the service is started.
     */
    private final AtomicBoolean started = new AtomicBoolean(false);

    /**
     * @return the name of the server node
     */
    public String getValue() throws IllegalStateException,
IllegalArgumentException {
        LOGGER.info(String.format("%s is %s at %s",
HATimerService.class.getSimpleName(), (started.get() ? "started" :
"not started"), System.getProperty("jboss.node.name")));
        return System.getProperty("jboss.node.name");
    }

    public void start(StartContext arg0) throws StartException {
        if (!started.compareAndSet(false, true)) {
            throw new StartException("The service is still
started!");
        }
        LOGGER.info("Start HASingleton timer service '" +
this.getClass().getName() + "'");

        final String node = System.getProperty("jboss.node.name");
        try {
            InitialContext ic = new InitialContext();
            ((Scheduler) ic.lookup("global/jboss-cluster-ha-
singleton-
service/SchedulerBean!org.jboss.as.quickstarts.cluster.hasingleton.s
ervice.ejb.Scheduler"))
                    .initialize("HASingleton timer @" + node + " " +
new Date());
        } catch (NamingException e) {
            throw new StartException("Could not initialize timer",
e);
        }
    }

    public void stop(StopContext arg0) {
        if (!started.compareAndSet(true, false)) {
            LOGGER.warning("The service '" +
this.getClass().getName() + "' is not active!");
        } else {
            LOGGER.info("Stop HASingleton timer service '" +
this.getClass().getName() + "'");
            try {
                InitialContext ic = new InitialContext();
                ((Scheduler) ic.lookup("global/jboss-cluster-ha-
singleton-
service/SchedulerBean!org.jboss.as.quickstarts.cluster.hasingleton.s
ervice.ejb.Scheduler")).stop();
            } catch (NamingException e) {
                LOGGER.info("Could not stop timer:" +
```

```
e.getMessage());
            }
        }
    }
}
```

2. **org.jboss.msc.service.ServiceActivator** インターフェースを実装し、**activate()** メソッドで **HATimerService** をクラスタ化されたシングルトンとしてインストールするサービスアクティベーターを作成します。この例では、**node1** がシングルトンサービスを開始するよう指定します。

サービスアクティベーターコード例

```
public class HATimerServiceActivator implements ServiceActivator {
    private final Logger log =
Logger.getLogger(this.getClass().toString());

    @Override
    public void activate(ServiceActivatorContext context) {
        log.info("HATimerService will be installed!");

        HATimerService service = new HATimerService();
        ServiceName factoryServiceName =
SingletonServiceName.BUILDER.getServiceName("server", "default");
        ServiceController<?> factoryService =
context.getServiceRegistry().getRequiredService(factoryServiceName);
        SingletonServiceBuilderFactory factory =
(SingletonServiceBuilderFactory) factoryService.getValue();
        ServiceName ejbComponentService = ServiceName.of("jboss",
"deployment", "unit", "jboss-cluster-ha-singleton-service.jar",
"component", "SchedulerBean", "START");

factory.createSingletonServiceBuilder(HATimerService.SINGLETON_SERVI
CE_NAME, service)
            .electionPolicy(new PreferredSingletonElectionPolicy(new
SimpleSingletonElectionPolicy(), new
NamePreference("node1/singleton")))
            .build(new
DelegatingServiceContainer(context.getServiceTarget(),
context.getServiceRegistry()))
            .setInitialMode(ServiceController.Mode.ACTIVE)
            .addDependency(ejbComponentService)
            .install();
    }
}
```

3. アプリケーションの **META-INF/services/** ディレクトリーで **org.jboss.msc.service.ServiceActivator** という名前のファイルを作成し、前の手順で作成された **ServiceActivator** クラスの完全修飾名を含む行を追加します。

META-INF/services/org.jboss.msc.service.ServiceActivator ファイル例

```
org.jboss.as.quickstarts.cluster.hasingleton.service.ejb.HATimerServ
iceActivator
```

4. **initialize()** および **stop()** メソッドを含む **Scheduler** インターフェースを作成します。

スケジューラーインターフェースコード例

```
public interface Scheduler {

    void initialize(String info);

    void stop();

}
```

5. **Scheduler** インターフェースを実装する **Singleton** Bean を作成します。この Bean はクラスター全体のシングルトンタイマーとして使用されます。

重要

Singleton Bean はリモートインターフェースを持たないようにし、すべてのアプリケーションの別の EJB からローカルインターフェースを参照しないようにする必要があります。これにより、クライアントや他のコンポーネントをルックアップできなくなり、**HATimerService** が **Singleton** を完全に制御するようになります。

シングルトン Bean コード例

```
@Singleton
public class SchedulerBean implements Scheduler {
    private static Logger LOGGER =
Logger.getLogger(SchedulerBean.class.toString());
    @Resource
    private TimerService timerService;

    @Timeout
    public void scheduler(Timer timer) {
        LOGGER.info("HASingletonTimer: Info=" + timer.getInfo());
    }

    @Override
    public void initialize(String info) {
        ScheduleExpression sexpr = new ScheduleExpression();
        // set schedule to every 10 seconds for demonstration
        sexpr.hour("*").minute("*").second("0/10");
        // persistent must be false because the timer is started by
the HASingleton service
        timerService.createCalendarTimer(sexpr, new
TimerConfig(info, false));
    }

    @Override
    public void stop() {
        LOGGER.info("Stop all existing HASingleton timers");
        for (Timer timer : timerService.getTimers()) {
            LOGGER.fine("Stop HASingleton timer: " +
timer.getInfo());
```

```
                timer.cancel();
            }
        }
    }
```

このアプリケーションの完全な稼働サンプルについては、JBoss EAP に同梱される **cluster-ha-singleton** クイックスタートを参照してください。クイックスタートは、アプリケーションをビルドおよびデプロイする詳細な手順を提供します。

6.5. HA シングルトンデプロイメント

JBoss EAP 7 には、該当するアプリケーションを**シングルトンデプロイメント**としてデプロイする機能が追加されます。

クラスタ化されたサーバーのグループにデプロイされる場合、シングルトンデプロイメントでは該当するタイミングで単一のノードにのみデプロイされます。デプロイメントがアクティブなノードが停止または失敗すると、デプロイメントは別のノードで自動的に開始されます。

HA シングルトンの動作を制御するポリシーは、新しい**シングルトン**サブシステムによって管理されます。デプロイメントは特定のシングルトンポリシーを指定するか、デフォルトのサブシステムポリシーを使用します。デプロイメントは、デプロイメントオーバーレイとして既存のデプロイメントに最も簡単に適用される **/META-INF/singleton-deployment.xml** デプロイメント記述子を使用してシングルトンデプロイメントとして識別されます。また、必要なシングルトン設定を既存の **jboss-all.xml** 内に組み込むこともできます。

シングルトンデプロイメントの定義または選択

- デプロイメントをシングルトンデプロイメントとして定義するには、アプリケーションアーカイブに **/META-INF/singleton-deployment.xml** 記述子を含めます。
  シングルトンデプロイメント記述子の例:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <singleton-deployment xmlns="urn:jboss:singleton-deployment:1.0"/>
  ```

  特定のシングルトンポリシーを使用したシングルトンデプロイメント記述子の例:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <singleton-deployment policy="my-new-policy"
  xmlns="urn:jboss:singleton-deployment:1.0"/>
  ```

- または、**singleton-deployment** 要素を **jboss-all.xml** 記述子に追加することもできます。
  jboss-all.xml の **singleton-deployment** 要素の例:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <jboss xmlns="urn:jboss:1.0">
      <singleton-deployment xmlns="urn:jboss:singleton-
  deployment:1.0"/>
  </jboss>
  ```

  特定のシングルトンポリシーを使用した **jboss-all.xml** の **singleton-deployment** 要素の例:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss xmlns="urn:jboss:1.0">
    <singleton-deployment policy="my-new-policy"
xmlns="urn:jboss:singleton-deployment:1.0"/>
</jboss>
```

シングルトンデプロイメントの作成

JBoss EAP は、以下の 2 つの選択ポリシーを提供します。

- 単純な選択ポリシー
  simple-election-policy は **position** 属性で示された特定のメンバーを選択します (該当するアプリケーションがデプロイされます)。**position** 属性は、降順の経過時間でソートされた候補のリストから選択するノードのインデックスを決定します (**0** は最も古いノード、**1** は 2 番目に古いノード、**-1** は最も新しいノード、**-2** は 2 番目に新しいノードを示します)。指定された位置が候補の数を超えると、モジュロ演算が適用されます。

  管理 CLI で **simple-election-policy** と位置を **-1** に設定した状態で新しいシングルトンポリシーを作成する例:

```
batch
/subsystem=singleton/singleton-policy=my-new-policy:add(cache-
container=server)
/subsystem=singleton/singleton-policy=my-new-policy/election-
policy=simple:add(position=-1)
run-batch
```

注記

新しく作成されたポリシー **my-new-policy** をデフォルトとして設定するには、以下のコマンドを実行します。

```
/subsystem=singleton:write-attribute(name=default,
value=my-new-policy)
```

  standalone-ha.xml を使用して位置が **-1** に設定された状態で **simple-election-policy** を設定する例:

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
   <singleton-policies default="my-new-policy">
      <singleton-policy name="my-new-policy" cache-
container="server">
         <simple-election-policy position="-1"/>
      </singleton-policy>
   </singleton-policies>
</subsystem>
```

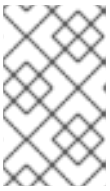
- ランダムな選択ポリシー
  random-election-policy は、該当するアプリケーションがデプロイされるランダムなメンバーを選択します。

  管理 CLI で **random-election-policy** を使用して新しいシングルトンポリシーを作成する例:

```
batch
/subsystem=singleton/singleton-policy=my-other-new-policy:add(cache-
container=server)
/subsystem=singleton/singleton-policy=my-other-new-policy/election-
policy=random:add()
run-batch
```

standalone-ha.xml を使用して **random-election-policy** を設定する例:

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
   <singleton-policies default="my-other-new-policy">
      <singleton-policy name="my-other-new-policy" cache-
container="server">
         <random-election-policy/>
      </singleton-policy>
   </singleton-policies>
</subsystem>
```



注記

ポリシーを追加する前に、**cache-container** の **default-cache** 属性を定義する必要があります。定義しないと、カスタムキャッシュコンテナーを使用する場合に、エラーメッセージが表示されることがあります。

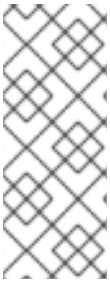
設定
また、シングルトン選択ポリシーを使用して、クラスターの 1 人または複数のメンバーの優先順位を指定することもできます。優先順位は、ノード名または送信ソケットバインド名を使用して定義できます。ノード優先順位は、常に選択ポリシーの結果よりも優先されます。

管理 CLI で既存のシングルトンポリシーの優先順位を指定する例:

```
/subsystem=singleton/singleton-policy=foo/election-policy=simple:list-
add(name=name-preferences, value=nodeA)

/subsystem=singleton/singleton-policy=bar/election-policy=random:list-
add(name=socket-binding-preferences, value=binding1)
```

管理 CLI で **simple-election-policy** と **name-preferences** を使用して新しいシングルトンポリシーを作成する例:

```
batch
/subsystem=singleton/singleton-policy=my-new-policy:add(cache-
container=server)
/subsystem=singleton/singleton-policy=my-new-policy/election-
policy=simple:add(name-preferences=[node1, node2, node3, node4])
run-batch
```



注記

新しく作成されたポリシー **my-new-policy** をデフォルトとして設定するには、以下のコマンドを実行します。

```
/subsystem=singleton:write-attribute(name=default, value=my-
new-policy)
```

standalone-ha.xml で **socket-binding-preferences** を使用して **random-election-policy** を設定する例:

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
   <singleton-policies default="my-other-new-policy">
      <singleton-policy name="my-other-new-policy" cache-
container="server">
         <random-election-policy>
            <socket-binding-preferences>binding1 binding2 binding3
binding4</socket-binding-preferences>
         </random-election-policy>
      </singleton-policy>
   </singleton-policies>
</subsystem>
```

クォーラム
ネットワークパーティションは、シングルトンデプロイメントに対して特に問題となります。これは、同時に実行する同じデプロイメントに対して複数のシングルトンプロバイダーをトリガーできるためです。このシナリオを回避するために、シングルトンポリシーにより、シングルトンプロバイダー選択が行われる前に、存在する必要があるノードの最小数を規定するクォーラムを定義できます。通常のデプロイメントシナリオでは、N/2 + 1 のクォーラムが使用されます (ここで、N は予想されるクラスターサイズです)。この値は実行時に更新でき、各シングルトンポリシーを使用するすべてのシングルトンデプロイメントにすぐに影響を与えます。

standalone-ha.xml ファイルでのクォーラム宣言の例:

```
<subsystem xmlns="urn:jboss:domain:singleton:1.0">
   <singleton-policies default="default">
      <singleton-policy name="default" cache-container="server"
quorum="4">
         <simple-election-policy/>
      </singleton-policy>
   </singleton-policies>
</subsystem>
```

管理 CLI を使用したクォーラム宣言の例:

```
/subsystem=singleton/singleton-policy=foo:write-attribute(name=quorum,
value=3)
```

6.6. APACHE MOD_CLUSTER-MANAGER アプリケーション

6.6.1. mod_cluster-manager アプリケーション

mod_cluster-manager アプリケーションは、Apache HTTP サーバーで利用可能な管理 Web ページです。接続されたワーカーノードを監視し、コンテキストの有効化または無効化やクラスター内のワーカーノードの負荷分散プロパティーの設定などのさまざまな管理タスクを実行するために使用されます。

mod_cluster-manager アプリケーションの使用
mod_cluster-manager アプリケーションは、ワーカーノードでさまざまな管理タスクを実行するために使用されます。

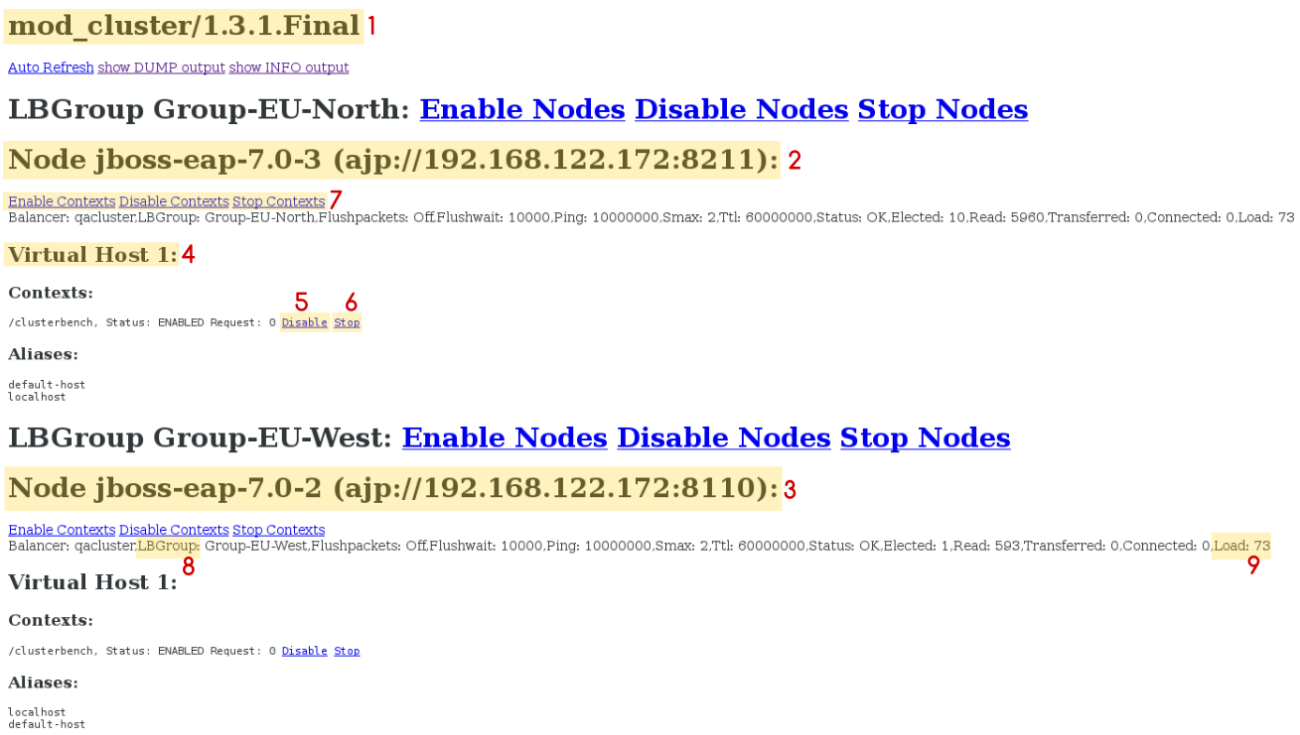
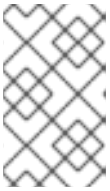


図 - mod_cluster 管理 Web ページ

- [1] **mod_cluster/1.3.1.Final**: mod_cluster ネイティブライブラリーのバージョン。
- [2] **ajp://192.168.122.204:8099**: 使用されるプロトコル (AJP、HTTP、または HTTPS)、ワーカーノードのホスト名または IP アドレス、およびポート。
- [3] **jboss-eap-7.0-2**: ワーカーノードの JVMRoute。
- [4] **Virtual Host 1**: ワーカーノードで設定された仮想ホスト。
- [5] **Disable**: 特定のコンテキストで新しいセッションの作成を無効にするために使用できる管理オプション。ただし、現在のセッションは無効にされず、そのまま処理されます。
- [6] **Stop**: コンテキストへのセッション要求のルーティングを停止するために使用できる管理オプション。`sticky-session-force` プロパティーが `true` に設定されない限り、残りのセッションは別のノードにフェイルオーバーされます。
- [7] **Enable Contexts Disable Contexts Stop Contexts**: ノード全体で実行できる操作。これらのいずれかのオプションを選択すると、すべての仮想ホストのノードのコンテキストすべてが影響を受けます。
- [8] **Load balancing group (LBGroup)**: すべてのワーカーノードをカスタム負荷分散グループにグループ化するために、 `load-balancing-group` プロパティーは JBoss EAP 設定の `modcluster` サブシステムで設定されます。負荷分散グループ (LBGroup) は、設定されたすべ

ての負荷分散グループに関する情報を提供する情報フィールドです。このフィールドが設定されていないと、すべてのワーカーノードは単一のデフォルト負荷分散グループにグループ化されます。

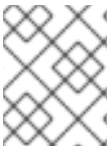


注記

これは唯一の情報フィールドであるため、**`load-balancing-group`** プロパティーの設定に使用できません。このプロパティーは JBoss EAP 設定の **`modcluster`** サブシステムで設定する必要があります。

- [9] **Load (value)**: ワーカーノードの負荷係数。負荷係数は以下のように評価されます。

> `-load > 0 :` 負荷係数の値が `1` であると、ワーカーノードがオーバーロードされます。負荷係数が `100` の場合は、負荷がないノードであることを意味します。
> `-load = 0 :` 負荷係数の値が `0` であると、ワーカーノードはスタンドバイモードになります。これは、他のノードが使用できなくなるまで (および他のノードが利用不可でない限り) セッション要求がこのノードにルーティングされないことを意味します。
> `-load = -1 :` 負荷係数の値が `-1` の場合は、ワーカーノードがエラー状態にあることを示します。
> `-load = -2 :` 負荷係数の値が `-2` の場合は、ワーカーノードが `CPing/CPong` を実行中であり、遷移状態にあることを示します。



注記

JBoss EAP 7.0 の場合は、ロードバランサーとして Undertow を使用することもできます。

第7章 コンテキストおよび依存関係の挿入 (CDI)

7.1. CDI の概要

7.1.1. コンテキストと依存関係の注入 (CDI)

Contexts and Dependency Injection (CDI) は、Enterprise Java Beans (EJB) 3 コンポーネントを Java Server Faces (JSF) 管理対象 Bean として使用できるよう設計された仕様であり、2 つのコンポーネントモデルを統合し、Java を使用した Web ベースのアプリケーション向けプログラミングモデルを大幅に簡略化します。CDI 1.2 リリースは、1.1 のメンテナンスリリースとして扱われます。 CDI 1.1 の詳細については、JSR 346: Contexts and Dependency Injection for Java™ EE 1.1 を参照してください。

JBoss EAP には、JSR-346:Contexts and Dependency Injection for Java™ EE 1.1 の参照実装である Weld が含まれます。

CDI の利点
CDI には以下のような利点があります。

- 多くのコードをアノテーションに置き換えることにより、コードベースが単純化および削減されます。
- 柔軟であり、インジェクションおよびイベントを無効または有効にしたり、代替の Bean を使用したり、非 CDI オブジェクトを簡単にインジェクトしたりできます。
- デフォルト値と異なるよう設定をカスタマイズする必要がある場合に、オプションで、`beans.xml` を `META-INF/` または `WEB-INF/` ディレクトリーに含めることができます。
- パッケージ化とデプロイメントが簡略化され、デプロイメントに追加する必要がある XML の量が減少します。
- コンテキストを使用したライフサイクル管理が提供されます。インジェクションを要求、セッション、会話、またはカスタムコンテキストに割り当てることができます。
- 文字列ベースのインジェクションよりも安全かつ簡単にデバッグを行える、タイプセーフな依存関係の注入が提供されます。
- インターセプターと Bean が切り離されます。
- 複雑なイベント通知が提供されます。

7.1.2. Weld、Seam 2、および JavaServer Faces 間の関係

Weld は JSR 346: Contexts and Dependency Injection for Java™ EE 1.1 で定義されている CDI の参照実装です。Weld は、Seam 2 と他の依存関係注入フレームワークの影響を受けており、JBoss EAP 6 に含まれています。

Seam 2 の目的は、Enterprise Java Bean と JavaServer Faces 管理対象 Bean を統合することでした。

JavaServer Faces 2.2 では、JSR-344: JavaServer™ Faces 2.2 が実装されます。これは、サーバーサイドユーザーインターフェースをビルドするための API です。

7.2. CDI の有効化

CDI は、JBoss EAP の中核的なテクノロジーの 1 つであり、デフォルトで有効になります。CDI は、設定ファイルの該当するセクションをコメントアウトまたは削除することにより無効になっている場合があります。有効にする必要がある場合は、以下の手順を実行します。

JBoss EAP での CDI の有効化

1. JBoss EAP を停止します。
   JBoss EAP により実行中に設定ファイルが変更されるため、設定ファイルを直接編集する前にサーバーを停止します。

2. 適切な設定ファイルを編集します。
   スタンドアロンサーバーの場合は **`EAP_HOME/standalone/configuration/standalone.xml`**、管理対象ドメインの場合は **`EAP_HOME/domain/configuration/domain.xml`** を編集します。

3. CDI 拡張機能を追加します。
   `org.jboss.as.weld` 拡張機能がコメントアウトされた場合は、コメント解除します。全体が削除された場合は、以下の行をファイルの **`</extensions>`** タグのすぐ上に新しい行で追加することにより復元します。

   ```
   <extension module="org.jboss.as.weld"/>
   ```

4. CDI サブシステムを追加します。
   `weld` サブシステムがコメントアウトされた場合は、コメント解除します。全体が削除された場合は、以下の行を **`<profiles>`** セクションの該当するプロファイルに追加することにより復元します。

   ```
   <subsystem xmlns="urn:jboss:domain:weld:3.0"/>
   ```

5. 更新された設定で JBoss EAP を起動します。

JBoss EAP が起動するとき、CDI サブシステムは有効になります。

7.3. CDI を使用したアプリケーションの開発

コンテキストと依存関係の注入 (CDI: Contexts and Dependency Injection) を使用すると、アプリケーションの開発、コードの再利用、デプロイメント時または実行時のコードの調整、およびユニットテストを非常に柔軟に実行できます。JBoss EAP には、CDI の参照実装である Weld が含まれます。これらのタスクは、エンタープライズアプリケーションで CDI を使用する方法を示しています。

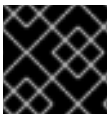
7.3.1. デフォルトの Bean 検出モード

bean アーカイブのデフォルトの bean 検出モードは **`annotated`** です。このような bean アーカイブは **`implicit bean archive`** と呼ばれます。

bean 検出モードが **`annotated`** の場合:

- **`bean defining annotation`** がなく、セッション bean の bean クラスでない bean クラスが検出されません。
- セッション bean 上になく、bean クラスが bean を定義するアノテーションを持たないプロデューサーメソッドが検出されません。

- セッション bean 上になく、bean クラスが bean を定義するアノテーションを持たないプロデューサーフィールドが検出されません。
- セッション bean 上になく、bean クラスが bean を定義するアノテーションを持たないディスポーザーメソッドが検出されません。
- セッション bean 上になく、bean クラスが bean を定義するアノテーションを持たないオブザーバーメソッドが検出されません。



重要

CDI セクションのすべての例は、検出モードが **all** に設定された場合にのみ有効です。

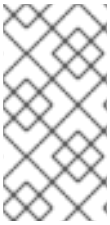
bean を定義するアノテーション
bean クラスは **bean defining annotation** を持つことがあり、Bean アーカイブで定義されたようにアプリケーションのどこにでも配置することができます。bean を定義するアノテーションを持つ bean クラスは暗黙的な bean と呼ばれます。

bean を定義するアノテーションのセットには以下のものが含まれます。

- **@ApplicationScoped**、**@SessionScoped**、**@ConversationScoped**、および **@RequestScoped** アノテーション
- その他すべての通常スコープタイプ
- **@Interceptor** および **@Decorator** アノテーション
- **@Stereotype** アノテーションが付けられた stereotype アノテーションすべて
- **@Dependent** スコープアノテーション

これらのアノテーションのいずれかが bean クラスで宣言された場合、その bean クラスは bean 定義アノテーションを持っていることになります。たとえば、以下の依存スコープ bean は bean 定義アノテーションを持っています。

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



注記

他の JSR-330 実装との互換性を確保するために、**@Dependent** を除くすべての pseudo-scope アノテーションは bean 定義アノテーションではありません。ただし、pseudo-scope アノテーションを含む stereotype アノテーションは bean 定義アノテーションです。

7.3.2. スキャンプロセスからの Bean の除外

除外フィルターは、Bean アーカイブの **beans.xml** ファイルの **<exclude>** 要素によって **<scan>** 要素の子として定義されます。デフォルトでは、除外フィルターはアクティブです。定義に以下のものが含まれる場合、除外フィルターは非アクティブになります。

- **name** 属性を含む、**<if-class-available>** という名前の子要素。Bean アーカイブのクラスローダーはこの名前のクラスをロードできません。
- **name** 属性を含む、**<if-class-not-available>** という名前の子要素。Bean アーカイブのクラスローダーはこの名前のクラスをロードできます。
- **name** 属性を含む、**<if-system-property>** という名前の子要素。この名前に対して定義されたシステムプロパティーはありません。
- **name** 属性と値属性を含む、**<if-system-property>** という名前の子要素。この名前とこの値に対して定義されたシステムプロパティーはありません。

フィルターがアクティブな場合、タイプは検出から除外され、以下のいずれかの状態になります。

- 検出されるタイプの完全修飾名が、除外フィルターの名前属性の値に一致します。
- 検出されるタイプのパッケージ名が、除外フィルターの接尾辞 ".*" を含む名前属性の値に一致します。
- 検出されるタイプのパッケージ名が、除外フィルターの接尾辞 ".*" を含む名前属性の値で始まります。

たとえば、以下の **beans.xml** ファイルを考えてみます。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee">

    <scan>
        <exclude name="com.acme.rest.*" /> 1

        <exclude name="com.acme.faces.**"> 2
            <if-class-not-available
name="javax.faces.context.FacesContext"/>
        </exclude>

        <exclude name="com.acme.verbose.*"> 3
            <if-system-property name="verbosity" value="low"/>
        </exclude>

        <exclude name="com.acme.ejb.**"> 4
            <if-class-available name="javax.enterprise.inject.Model"/>
            <if-system-property name="exclude-ejbs"/>
        </exclude>
    </scan>

</beans>
```

1 最初の除外フィルターにより、**com.acme.rest** パッケージ内のすべてのクラスが除外されます。

2 2 番目の除外フィルターにより、**com.acme.faces** パッケージとすべてのサブパッケージ内のすべてのクラスが除外されます (JSF が利用可能でない場合のみ)。

3 3 番目の除外フィルターにより、システムプロパティー **verbosity** が値 **low** を持つ場合に、**com.acme.verbose** パッケージ内のすべてのクラスが除外されます。

4 4 番目の除外フィルターにより、システムプロパティー **exclude-ejbs** が任意の値で設定され、**javax.enterprise.inject.Model** クラスがクラスローダーでも利用可能な場合に、**com.acme.ejb** パッケージとすべてのサブパッケージ内のすべてのクラスが除外されます。

注記

Java EE コンポーネントが Bean と見なされないように、Java EE コンポーネントは **@Vetoed** でアノテートすることが安全です。イベントは **@Vetoed** でアノテートされたタイプに対して、または **@Vetoed** でアノテートされたパッケージで発生しません。詳細については、**@Vetoed** を参照してください。

7.3.3. インジェクションを使用した実装の拡張

インジェクションを使用して、既存のコードの機能を追加または変更できます。

この例では、既存のクラスに翻訳機能を追加します。メソッド **buildPhrase** を持つ Welcome クラスがすでにあることを前提とします。**buildPhrase** メソッドは、都市の名前を引数として取得し、"Welcome to Boston!" などのフレーズを出力します。

例: Translator Bean を Welcome クラスにインジェクトする

以下のコードにより、想像上の **Translator** オブジェクトが **Welcome** クラスにインジェクトされます。**Translator** オブジェクトは、文をある言語から別の言語に翻訳できる EJB ステートレス Bean または別のタイプの Bean になります。この例では、**Translator** は挨拶全体を翻訳するために使用され、元の **Welcome** クラスは変更されません。**Translator** は、**buildPhrase** メソッドが呼び出される前にインジェクトされます。

```
public class TranslatingWelcome extends Welcome {

    @Inject Translator translator;

    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
    }
    ...
}
```

7.4. あいまいな依存関係または満たされていない依存関係

コンテナーが 1 つの Bean への注入を解決できない場合、依存関係があいまいとなります。

コンテナーがいずれの Bean に対しても注入の解決をできない場合、依存関係が満たされなくなります。

コンテナーは以下の手順を踏み、依存関係の解決をはかります。

1. インジェクションポイントの Bean 型を実装する全 Bean にある修飾子アノテーションを解決します。
2. 無効となっている Bean をフィルタリングします。無効な Bean とは、明示的に有効化されていない **@Alternative** Bean のことです。

依存関係があいまいな場合、あるいは満たされない場合は、コンテナーはデプロイメントを中断して例外を発生させます。

あいまいな依存関係を修正するには、修飾子を使用したあいまいなインジェクションの解決を参照してください。

7.4.1. 修飾子

修飾子は、コンテナーが複数の Bean を解決できるときにあいまいな依存関係を回避するために使用されるアノテーションであり、インジェクションポイントに含められます。インジェクションポイントで宣言された修飾子は、同じ修飾子を宣言する有効な Bean セットを提供します。

修飾子は、以下の例で示されたように Retention と Target を使用して宣言する必要があります。

例: @Synchronous 修飾子と @Asynchronous 修飾子の定義

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

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

例: @Synchronous 修飾子と @Asynchronous 修飾子の使用

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

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

'@Any'
Bean またはインジェクションポイントにより修飾子が明示的に宣言されない場合、コンテナーにより修飾子は **@Default** と見なされます。場合によっては、修飾子を指定せずにインジェクションポイントを宣言する必要があります。この場合も修飾子が存在します。すべての Bean には修飾子 **@Any** が含まれます。したがって、インジェクションポイントで **@Any** を明示的に指定することにより、インジェクションが可能な Bean を制限せずにデフォルトの修飾子を抑制できます。

これは、特定の Bean タイプを持つすべての Bean に対して繰り返し処理を行う場合に特に役に立ちます。

```
import javax.enterprise.inject.Instance;
...

@Inject
```

```
void initServices(@Any Instance<Service> services) {

   for (Service service: services) {

      service.init();

   }

}
```

各 Bean は修飾子 **@Any** を持ちます (この修飾子が明示的に指定されていない場合でも)。

各イベントも修飾子 **@Any** を持ちます (この修飾子を明示的に宣言せずにイベントが発生した場合でも)。

```
@Inject @Any Event<User> anyUserEvent;
```

@Any 修飾子により、インジェクションポイントが特定の Bean タイプのすべての Bean またはイベントを参照できます。

```
@Inject @Delegate @Any Logger logger;
```

7.4.2. 修飾子を使用したあいまいなインジェクションの解決

修飾子を使用してあいまいなインジェクションを解決できます。あいまいなインジェクションについては、あいまいな依存関係または満たされていない依存関係をお読みください。

以下の例はあいまいであり、**Welcome** の 2 つの実装 (翻訳を行う 1 つと翻訳を行わないもう 1 つ) を含みます。翻訳を行う **Welcome** を使用するには、インジェクションを指定する必要があります。

例: あいまいなインジェクション

```
public class Greeter {
  private Welcome welcome;

  @Inject
  void init(Welcome welcome) {
    this.welcome = welcome;
  }
  ...
}
```

修飾子を使用したあいまいなインジェクションの解決

1. あいまいなインジェクションを解決するには、**@Translating** という名前の修飾子アノテーションを作成します。

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

2. **@Translating** アノテーションを使用して、翻訳を行う **Welcome** をアノテートします。

   ■

```
@Translating
public class TranslatingWelcome extends Welcome {
    @Inject Translator translator;
    public String buildPhrase(String city) {
        return translator.translate("Welcome to " + city + "!");
    }
    ...
}
```

3. インジェクションで翻訳を行う **Welcome** を要求します。ファクトリーメソッドパターンの場合と同様に、修飾された実装を明示的に要求する必要があります。あいまいさはインジェクションポイントで解決されます。

```
public class Greeter {
  private Welcome welcome;
  @Inject
  void init(@Translating Welcome welcome) {
    this.welcome = welcome;
  }
  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase("San Francisco"));
  }
}
```

7.5. 管理 BEAN

Java EE では管理対象 Bean 仕様の共通定義が確立されています。管理対象 Bean は、プログラミングの制限が最小限であるコンテナー管理オブジェクトとして定義され、POJO (Plain Old Java Object) として知られるようになりました。管理対象 Bean はリソースのインジェクション、ライフサイクルコールバック、インターセプターなどの基本サービスの小さなセットをサポートします。EJBや CDI などのコンパニオン仕様は、この基本モデルに基づいて構築されます。

ごくわずかな例外を除き、パラメーターのないコンストラクター (または **@Inject** アノテーションが指定されたコンストラクター) を持つ具象 Java クラスは Bean になります。これには、すべての Java Bean と EJB セッション Bean が含まれます。

7.5.1. Bean であるクラスのタイプ

管理対象 Bean は Java クラスです。管理対象 Bean の基本的なライフサイクルやセマンティクスは、管理対象 Bean の仕様で定義されています。Bean クラス **@ManagedBean** をアノテートすることで明示的に管理対象 Bean を宣言できますが、CDI ではその必要はありません。この仕様によると、CDI コンテナーでは、以下の条件を満たすクラスはすべて管理対象 Bean として扱われます。

- 非静的な内部クラスではないこと。
- 具象クラス、あるいは **@Decorator** アノテーションが付与されている。
- EJB コンポーネントを定義するアノテーションが付与されていないこと、あるいは **ejb-jar.xml** で EJB Bean クラスとして宣言されていること。
- インターフェース **javax.enterprise.inject.spi.Extension** が実装されていないこと。

- パラメーターのないコンストラクターか、**@Inject** アノテーションが付与されたコンストラクターがあること。
- アノテートされた **@Vetoed** でないこと、または **@Vetoed** でアノテートされたパッケージ内にないこと。

管理対象 Bean の Bean 型で無制限のものには、直接的あるいは間接的に実装する Bean クラス、全スーパークラス、および全インターフェースが含まれます。

管理対象 Bean にパブリックフィールドがある場合は、デフォルトの **@Dependent** スコープが必要です。

@Vetoed
CDI 1.1 には、新しいアノテーションである **@Vetoed** が導入されました。このアノテーションを追加することにより、Bean をインジェクションから除外できます。

```
@Vetoed
public class SimpleGreeting implements Greeting {
    ...
}
```

このコードでは、**SimpleGreeting** Bean はインジェクションの対象となりません。

パッケージ内のすべての Bean をインジェクションから除外できます。

```
@Vetoed
package org.sample.beans;

import javax.enterprise.inject.Vetoed;
```

org.sample.beans パッケージ内の **package-info.java** のこのコードにより、このパッケージ内のすべての Bean がインジェクションから除外されます。

ステートレス EJB や JAX-RS リソースエンドポイントなどの Java EE コンポーネントは、Bean と見なされないように **@Vetoed** でマークできます。**@Vetoed** アノテーションをすべての永続エンティティーに追加すると、**BeanManager** がエンティティーを CDI Bean として管理することを回避できます。エンティティーが **@Vetoed** とアノテートされた場合は、インジェクションが行われません。この理由は、JPA プロバイダーが破損する原因となる操作を **BeanManager** が実行することを防ぐことです。

7.5.2. CDI を用いたオブジェクトの Bean へのインジェクト

CDI コンポーネントがアプリケーションで検出されると、CDI は自動的にアクティベートされます。デフォルト値と異なるよう設定をカスタマイズする場合は、デプロイメントアーカイブに **META-INF/beans.xml** または **WEB-INF/beans.xml** を含めることができます。

他のオブジェクトにオブジェクトをインジェクトする

1. クラスのインスタンスを取得するには、Bean 内で **@Inject** を使用してフィールドをアノテートします。

```
public class TranslateController {
   @Inject TextTranslator textTranslator;
   ...
```

2. インジェクトしたオブジェクトのメソッドを直接使用します。**TextTranslator** にメソッド **translate** があることを前提とします。

```
// in TranslateController class

public void translate() {
   translation = textTranslator.translate(inputText);
}
```

3. Bean のコンストラクターでインジェクションを使用します。ファクトリーやサービスロケーターを使用して作成する代わりに、Bean のコンストラクターへオブジェクトをインジェクトできます。

```
public class TextTranslator {

   private SentenceParser sentenceParser;
   private Translator sentenceTranslator;

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

   // Methods of the TextTranslator class
   ...
}
```

4. **Instance(<T>)** インターフェースを使用してインスタンスをプログラムにより取得します。Bean 型でパラメーター化されると、**Instance** インターフェースは **TextTranslator** のインスタンスを返すことができます。

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

オブジェクトを Bean にインジェクトすると、Bean は全オブジェクトのメソッドとプロパティーを使用できるようになります。Bean のコンストラクターにインジェクトするときに、インジェクションがすでに存在するインスタンスを参照する場合以外は、Bean のコンストラクターが呼び出されるとインジェクトされたオブジェクトのインスタンスが作成されます。たとえば、セッションの存続期間内にセッションスコープの Bean をインジェクトしても、新しいインスタンスは作成されません。

7.6. コンテキストおよびスコープ

CDI では、特定のスコープに関連付けられた Bean のインスタンスを保持するストレージ領域をコンテキストと呼びます。

A scope is the link between a bean and a context. A scope/context combination may have a specific lifecycle. Several predefined scopes exist, and you can create your own. Examples of predefined scopes are **@RequestScoped**, **@SessionScoped**, and **@ConversationScope**.

表7.1 利用可能なスコープ

範囲	説明
@Dependent	Bean は、参照を保持する Bean のライフサイクルにバインドされます。インジェクション Bean のデフォルトのスコープは **@Dependent** です。
@ApplicationScoped	Bean はアプリケーションのライフサイクルにバインドされます。
@RequestScoped	Bean はリクエストのライフサイクルにバインドされます。
@SessionScoped	Bean はセッションのライフサイクルにバインドされます。
@ConversationScoped	Bean は会話のライフサイクルにバインドされます。会話スコープは、リクエストの長さとセッションの間であり、アプリケーションによって制御されます。
カスタムスコープ	上記のコンテキストで対応できない場合は、カスタムスコープを定義できます。

7.7. 名前付き BEAN

Bean には、**@Named** アノテーションを使用して名前を付けることができます。Bean を命名することにより、Bean を Java Server Faces (JSF) と Expression Language (EL) で直接使用できるようになります。

@Named アノテーションは、Bean 名であるオプションパラメーターを取ります。このパラメーターが省略された場合、Bean 名はデフォルトで最初の文字が小文字に変換された Bean のクラス名に設定されます。

7.7.1. 名前付き Bean の使用

@Named Annotation を使用した Bean 名の設定

1. **@Named** アノテーションを使用して名前を Bean に割り当てます。

```
@Named("greeter")
public class GreeterBean {
  private Welcome welcome;

  @Inject
  void init (Welcome welcome) {
    this.welcome = welcome;
  }

  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase("San Francisco"));
  }
}
```

上記の例では、名前が指定されていない場合、デフォルトの名前は **greeterBean** になります。

2. JSF ビューで名前付き Bean を使用します。

```
<h:form>
  <h:commandButton value="Welcome visitors" action="#
{greeter.welcomeVisitors}"/>
</h:form>
```

7.8. BEAN ライフサイクル

このタスクは、リクエストの残存期間の間 Bean を保存する方法を示しています。

インジェクトされた Bean のデフォルトのスコープは **@Dependent** です。つまり、Bean のライフサイクルは、参照を保持する Bean のライフサイクルに依存します。他の複数のスコープが存在し、独自のスコープを定義できます。詳細については、コンテキストおよびスコープを参照してください。

Bean ライフサイクルの管理

1. 必要なスコープで Bean をアノテートします。

```
@RequestScoped
@Named("greeter")
public class GreeterBean {
  private Welcome welcome;
  private String city; // getter & setter not shown
  @Inject   void init(Welcome welcome) {
    this.welcome = welcome;
  }
  public void welcomeVisitors() {
    System.out.println(welcome.buildPhrase(city));
  }
}
```

2. Bean が JSF ビューで使用されると、Bean は状態を保持します。

```
<h:form>
  <h:inputText value="#{greeter.city}"/>
  <h:commandButton value="Welcome visitors" action="#
{greeter.welcomeVisitors}"/>
</h:form>
```

Bean は、指定するスコープに関連するコンテキストに保存され、スコープが適用される限り存続します。

7.8.1. プロデューサーメソッドの使用

プロデューサーメソッドは、Bean インスタンスのソースとして動作するメソッドです。指定されたコンテキストにインスタンスが存在しない場合は、メソッド宣言自体で Bean が定義され、コンテナーによって Bean のインスタンスを取得するメソッドが呼び出されます。プロデューサーメソッドにより、アプリケーションは Bean インスタンス化プロセスを完全に制御できるようになります。

このタスクは、インジェクション用の Bean ではないさまざまなオブジェクトを生成するプロデューサーメソッドを使用する方法を示しています。

例: 代替の代わりにプロデューサーメソッドを使用してデプロイメント後のポリモーフィズムを可能にする

例の **@Preferred** アノテーションは、修飾子アノテーションです。修飾子の詳細については、修飾子を参照してください。

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

以下のインジェクションポイントは、プロデューサーメソッドと同じタイプおよび修飾子アノテーションを持つため、通常の CDI インジェクションルールを使用してプロデューサーメソッドに解決されます。プロデューサーメソッドは、このインジェクションポイントを処理するインスタンスを取得するためにコンテナーにより呼び出されます。

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

例: スコープをプロデューサーメソッドに割り当てる

プロデューサーメソッドのデフォルトのスコープは **@Dependent** です。スコープを Bean に割り当てた場合、スコープは適切なコンテキストにバインドされます。この例のプロデューサーメソッドは、1 つのセッションあたり一度だけ呼び出されます。

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

例: プロデューサーメソッド内部でのインジェクションの使用

アプリケーションにより直接インスタンス化されたオブジェクトは、依存関係の注入を利用できず、インターセプターを持ちません。ただし、プロデューサーメソッドへの依存関係の注入を使用して Bean インスタンスを取得できます。

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

リクエストスコープの Bean をセッションスコープのプロデューサーにインジェクトする場合は、プロ

デューサーメソッドにより、現在のリクエストスコープのインスタンスがセッションスコープにプロモートされます。これは、適切な動作ではないため、プロデューサーメソッドをこのように使用する場合は注意してください。

注記

プロデューサーメソッドのスコープは、プロデューサーメソッドを宣言する Bean から継承されません。

プロデューサーメソッドを使用して、Bean ではないオブジェクトをインジェクトし、コードを動的に変更できます。

7.9. 代替の BEAN

実装が特定のクライアントモジュールまたはデプロイメントシナリオに固有である Bean が代替となります。

デフォルトでは、**@Alternative** Bean が無効になります。これらは、**beans.xml** ファイルを編集することにより、特定の Bean アーカイブに対して有効になります。ただし、このアクティベーションは、そのアーカイブの Bean に対してのみ適用されます。CDI 1.1 以降、代替の Bean は、**@Priority** アノテーションを使用してアプリケーション全体に対して有効にできます。

代替の定義例

この代替により、**@Synchronous** 代替と**@Asynchronous** 代替を使用して **PaymentProcessor** クラスの実装が定義されます。

```
@Alternative @Synchronous @Asynchronous

public class MockPaymentProcessor implements PaymentProcessor {

   public void process(Payment payment) { ... }

}
```

beans.xml で @Alternative を有効にする例

```
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
   <alternatives>
         <class>org.mycompany.mock.MockPaymentProcessor</class>
   </alternatives>
</beans>
```

選択された代替の宣言

@Priority アノテーションにより、アプリケーション全体に対して代替を有効にすることができるようになります。代替にはアプリケーションの優先度を割り当てることができます。

- 管理対象 Bean またはセッション Bean の Bean クラスに **@Priority** アノテーションを置く、または

- プロデューサーメソッド、フィールド、またはリソースを宣言する Bean クラスに **@Priority** アノテーションを置く

7.9.1. 代替を用いたインジェクションのオーバーライド

代替の Bean を使用すると、既存の Bean をオーバーライドできます。これらは、同じ役割を満たすクラスをプラグインする方法として考慮できますが、動作が異なります。代替の Bean はデフォルトで無効になります。

このタスクは、代替を指定し、有効にする方法を示しています。

インジェクションのオーバーライド
このタスクでは、プロジェクトに **TranslatingWelcome** クラスがすでにあることを前提としています。ただし、これを "mock" **TranslatingWelcome** クラスでオーバーライドするとします。これは、実際の **Translator** Bean を使用できないテストデプロイメントのケースに該当します。

1. 代替を定義します。

```
@Alternative
@Translating
public class MockTranslatingWelcome extends Welcome {
  public String buildPhrase(string city) {
    return "Bienvenue Ã  " + city + "!");
  }
}
```

2. 置換実装をアクティベートするために、完全修飾クラス名を **META-INF/beans.xml** または **WEB-INF/beans.xml** ファイルに追加します。

```
<beans>
  <alternatives>
    <class>com.acme.MockTranslatingWelcome</class>
  </alternatives>
</beans>
```

元の実装の代わりに代替実装が使用されます。

7.10. ステレオタイプ

多くのシステムでは、アーキテクチャーパターンを使用して繰り返し発生する Bean ロールのセットを生成します。ステレオタイプを使用すると、このようなロールを指定し、中心的な場所で、このロールを持つ Bean に対する共通メタデータを宣言できます。

ステレオタイプにより、以下のいずれかの組み合わせがカプセル化されます。

- デフォルトスコープ
- インターセプターバインディングのセット

また、ステレオタイプにより、以下の 2 つのいずれかを指定できます。

- ステレオタイプがデフォルトの Bean EL 名であるすべての Bean
- ステレオタイプが代替であるすべての Bean

Bean は、ステレオタイプを 0 個以上宣言できます。ステレオタイプは、他の複数のアノテーションをパッケージ化する **@Stereotype** アノテーションです。ステレオタイプアノテーションは、Bean クラス、プロデューサーメソッド、またはフィールドに適用できます。

ステレオタイプからスコープを継承するクラスは、そのステレオタイプをオーバーライドし、Bean で直接スコープを指定できます。

また、ステレオタイプが **@Named** アノテーションを持つ場合、配置された Bean はデフォルトの Bean 名を持ちます。この Bean は、**@Named** アノテーションが Bean で直接指定された場合に、この名前をオーバーライドできます。名前付き Bean の詳細については、名前付き Beanを参照してください。

7.10.1. ステレオタイプの使用

ステレオタイプを使用しないと、アノテーションが煩雑になる可能性があります。このタスクは、ステレオタイプを使用して煩雑さとコードを減らす方法を示しています。

例: アノテーションの煩雑さ

```
@Secure
@Transactional
@RequestScoped
@Named
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

ステレオタイプの定義および使用

1. ステレオタイプを定義します。

```
@Secure
@Transactional
@RequestScoped
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface BusinessComponent {
 ...
}
```

2. ステレオタイプを使用します。

```
@BusinessComponent
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

7.11. オブザーバーメソッド

オブザーバーメソッドは、イベント発生時に通知を受け取ります。

また、CDI は、イベントが発生したトランザクションの完了前または完了後フェーズ中にイベント通知を受け取るトランザクションオブザーバーメソッドを提供します。

7.11.1. イベントの発生と確認

例: イベントの発生

以下のコードは、メソッドでインジェクトおよび使用されるイベントを示しています。

```
public class AccountManager {
  @Inject Event<Withdrawal> event;

  public boolean transfer(Account a, Account b) {
    ...
    event.fire(new Withdrawal(a));
  }
}
```

例: 修飾子を使用したイベントの発生

修飾子を使用すると、より具体的にイベントのインジェクションにアノテーションを付けられます。修飾子の詳細については、修飾子を参照してください。

```
public class AccountManager {
  @Inject @Suspicious Event <Withdrawal> event;

  public boolean transfer(Account a, Account b) {
    ...
    event.fire(new Withdrawal(a));
  }
}
```

例: イベントの確認

イベントを確認するには、**@Observes** アノテーションを使用します。

```
public class AccountObserver {
  void checkTran(@Observes Withdrawal w) {
    ...
  }
}
```

修飾子を使用すると、特定の種類のイベントのみを確認できます。

```
public class AccountObserver {
  void checkTran(@Observes @Suspicious Withdrawal w) {
    ...
  }
}
```

7.11.2. トランザクションオブザーバー

トランザクションオブザーバーは、イベントが発生したトランザクションの完了フェーズ前または完了フェーズ後にイベント通知を受け取ります。トランザクションオブザーバーは、単一のアトミックトランザクションよりも状態が保持される期間が長いため、トランザクションオブザーバーはステートフルオブジェクトモデルで重要になります。

トラザクションオブザーバーには 5 つの種類があります。

- **IN_PROGRESS**: デフォルトではオブザーバーは即座に呼び出されます。
- **AFTER_SUCCESS**: トランザクションが正常に完了する場合のみ、オブザーバーはトランザクションの完了フェーズの後に呼び出されます。
- **AFTER_FAILURE**: トランザクションの完了に失敗する場合のみ、オブザーバーはトランザクションの完了フェーズの後に呼び出されます。
- **AFTER_COMPLETION**: オブザーバーはトランザクションの完了フェーズの後に呼び出されます。
- **BEFORE_COMPLETION**: オブザーバーはトランザクションの完了フェーズの前に呼び出されます。

以下のオブザーバーメソッドは、カテゴリーツリーを更新するトランザクションが正常に実行される場合のみアプリケーションコンテキストにキャッシュされたクエリー結果セットを更新します。

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

アプリケーションスコープで JPA クエリー結果セットをキャッシュしたことを仮定します。

```
import javax.ejb.Singleton;
import javax.enterprise.inject.Produces;

@ApplicationScoped @Singleton

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

Product はときどき作成および削除されます。**Product** が作成または削除されると **Product** カタログを更新する必要がありますが、トランザクションが正常に完了した後に更新を行う必要があります。

以下は、イベントを引き起こす **Products** を作成および削除する Bean の例になります。

```
import javax.enterprise.event.Event;
```

```
@Stateless

public class ProductManager {
   @PersistenceContext EntityManager em;
   @Inject @Any Event<Product> productEvent;
   public void delete(Product product) {
      em.delete(product);
      productEvent.select(new AnnotationLiteral<Deleted>()
{}).fire(product);
   }

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

トランザクションが正常に完了した後に、**Catalog** がイベントを監視できるようになりました。

```
import javax.ejb.Singleton;

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

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

}
```

7.12. インターセプター

インターセプターを使用すると、Bean のメソッドを直接変更せずに Bean のビジネスメソッドに機能を追加できます。インターセプターは、Bean のビジネスメソッドの前に実行されます。インターセプターは、JSR 318: Enterprise JavaBeans™ 3.1 仕様の一部として定義されています。

CDI により、インターセプターと Bean をバインドするアノテーションを利用できるため、この機能が強化されます。

インターセプションポイント

- ビジネスメソッドインターセプション: ビジネスメソッドのインターセプターは、Bean のクライアントによる Bean のメソッド呼び出しに適用されます。
- ライフサイクルコールバックインターセプション: ライフサイクルコールバックインターセプションは、コンテナーによるライフサイクルコールバックの呼び出しに適用されます。

- タイムアウトメソッドインターセプター: タイムアウトメソッドインターセプターは、コンテナーによる EJB タイムアウトメソッドの呼び出しに適用されます。

インターセプターの有効化
デフォルトでは、すべてのインターセプターが無効になります。インターセプターは、Bean アーカイブの **beans.xml** 記述子を使用して有効にすることができます。ただし、このアクティベーションは、そのアーカイブの Bean に対してのみ適用されます。CDI 1.1 以降、インターセプターは、**@Priority** アノテーションを使用してアプリケーション全体に対して有効にできます。

```
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
   <interceptors>
      <class>org.mycompany.myapp.TransactionInterceptor</class>
   </interceptors>
</beans>
```

XML 宣言を使用すると、以下の 2 つのことが可能になります。

- システムでインターセプターの順序を指定して、決定論的な動作を得ることができます。
- デプロイメント時にインターセプタークラスを有効または無効にすることができます。

@Priority を使用して有効にされたインターセプターは、**beans.xml** ファイルを使用して有効にされたインターセプターよりも前に呼び出されます。

注記

インターセプターが **@Priority** により有効にされ、同時に **beans.xml** により呼び出されると、移植不可能な動作になります。したがって、異なる CDI 実装で整合性のある動作を維持するには、この組み合わせの有効化を回避する必要があります。

7.12.1. CDI とのインターセプターの使用

CDI により、インターセプターコードが単純化され、ビジネスコードへの適用が簡単になります。

CDI がない場合、インターセプターには 2 つの問題があります。

- Bean は、インターセプター実装を直接指定する必要があります。
- アプリケーションの各 Bean は、インターセプターの完全なセットを適切な順序で指定する必要があります。この場合、アプリケーション全体でインターセプターを追加または削除するには時間がかかり、エラーが発生する傾向があります。

例: CDI のないインターセプター

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

```
@Stateful public class BusinessComponent {
  ...
}
```

CDI とのインターセプターの使用

1. インターセプターバインディングタイプを定義します。

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

2. インターセプター実装をマークします。

```
@Secure
@Interceptor
public class SecurityInterceptor {
  @AroundInvoke
  public Object aroundInvoke(InvocationContext ctx) throws Exception
{
    // enforce security ...
    return ctx.proceed();
    }
}
```

3. ビジネスコードでインターセプターを使用します。

```
@Secure
public class AccountManager {
  public boolean transfer(Account a, Account b) {
    ...
  }
}
```

4. インターセプターを **META-INF/beans.xml** または **WEB-INF/beans.xml** に追加することにより、インターセプターをデプロイメントで有効にします。

```
<beans>
  <interceptors>
    <class>com.acme.SecurityInterceptor</class>
    <class>com.acme.TransactionInterceptor</class>
  </interceptors>
</beans>
```

インターセプターは、リストされた順序で適用されます。

7.13. デコレーター

デコレーターは、特定の Java インターフェースからの呼び出しをインターセプトし、そのインターフェースに割り当てられたすべてのセマンティクスを認識します。デコレーターは、何らかの業務をモデル化するのに役に立ちますが、インターセプターの一般性を持ちません。デコレーターは Bean また

は抽象クラスであり、デコレートするタイプを実装し、**@Decorator** アノテーションが付けられます。CDI アプリケーションでデコレーターを呼び出すには、**beans.xml** ファイルで指定する必要があります。

beans.xml でデコレーターを呼び出す例

```
<beans
   xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd">
   <decorators>
         <class>org.mycompany.myapp.LargeTransactionDecorator</class>
   </decorators>
</beans>
```

この宣言には 2 つの主な目的があります。

- システムでデコレーターの順序を指定して、決定論的な動作を得ることができます。
- デプロイメント時にデコレータークラスを有効または無効にすることができます。

デコレートされたオブジェクトへの参照を取得するために、デコレーターには **@Delegate** インジェクションポイントが 1 つ必要になります。

デコレーターの例

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

   @Inject @Delegate @Any Account account;
   @PersistenceContext EntityManager em;

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

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

CDI 1.1 以降、デコレーターは、**@Priority** アノテーションを使用してアプリケーション全体に対して有効にできます。

@Priority を使用して有効にされたデコレーターは、**beans.xml** を使用して有効にされたデコレーターよりも前に呼び出されます。低い優先度値が最初に呼び出されます。

注記

デコレーターが **@Priority** により有効にされ、同時に **beans.xml** により呼び出されると、移植不可能な動作になります。したがって、異なる CDI 実装で整合性のある動作を維持するには、この組み合わせの有効化を回避する必要があります。

7.14. 移植可能な拡張機能

CDI は、フレームワーク、拡張機能、および他のテクノロジーとの統合の基礎となることを目的としています。したがって、CDI は、移植可能な CDI の拡張機能の開発者が使用する SPI のセットを公開します。

拡張機能は、以下のような種類の機能を提供できます。

- ビジネスプロセス管理エンジンとの統合
- Spring、Seam、GWT、Wicket などのサードパーティーフレームワークとの統合
- CDI プログラミングモデルに基づく新しいテクノロジー

JSR-346 仕様に基づいて、移植可能な拡張機能は次の方法でコンテナーと統合できます。

- 独自の Bean、インターセプター、およびデコレーターをコンテナーに提供します。
- 依存関係注入サービスを使用した独自のオブジェクトへの依存関係のインジェクション
- カスタムスコープのコンテキスト実装を提供します。
- アノテーションベースのメタデータを別のソースからのメタデータで拡大またはオーバーライドします。

7.15. BEAN プロキシ

通常、インジェクトされた Bean のクライアントは Bean インスタンスへの直接参照を保持しません。Bean が依存オブジェクト (スコープ **@Dependent**) でない場合、コンテナーはプロキシオブジェクトを使用して、インジェクトされたすべての参照を Bean にリダイレクトする必要があります。

この Bean プロキシはクライアントプロキシと呼ばれ、メソッド呼び出しを受け取る Bean インスタンスが、現在のコンテキストと関連するインスタンスになるようにします。またクライアントプロキシは、他のインジェクトされた Bean を再帰的にシリアライズせずに、セッションコンテキストなどのコンテキストにバインドされる Bean をディスクへシリアライズできるようにします。

Java の制限により、コンテナーによるプロキシの作成が不可能な Java の型があります。これらの型の 1 つで宣言されたインジェクションポイントが、**@Dependent** 以外のスコープを持つ Bean に解決すると、コンテナーがデプロイメントをアボートします。

特定の Java の型ではコンテナーによってプロキシを作成できません。これらの型には次のようなものがあります。

- パラメーターのない非プライベートコンストラクターを持たないクラス
- **final** が宣言されたクラスまたは **final** メソッドを持つクラス
- 配列およびプリミティブ型

7.16. インジェクションでのプロキシの使用

各 Bean のライフサイクルが異なる場合に、インジェクションにプロキシが使用されます。プロキシは起動時に作成された Bean のサブクラスで、Bean クラスのプライベートメソッド以外のメソッドをすべて上書きします。プロキシは実際の Bean インスタンスへ呼び出しを転送します。

この例では、**PaymentProcessor** インスタンスは直接 **Shop** へインジェクトされません。その代わり

にプロキシがインジェクトされ、**processPayment()** メソッドが呼び出されるとプロキシが現在の **PaymentProcessor** Bean インスタンスをルックアップし、**processPayment()** メソッドを呼び出します。

例: プロキシインジェクション

```
@ConversationScoped
class PaymentProcessor
{
  public void processPayment(int amount)
  {
    System.out.println("I'm taking $" + amount);
  }
}

@ApplicationScoped
public class Shop
{

  @Inject
  PaymentProcessor paymentProcessor;

  public void buyStuff()
  {
    paymentProcessor.processPayment(100);
  }
}
```

第8章 JBOSS EAP MBEAN サービス

管理対象 Bean (単に MBean と呼ばれることもあります) は、依存関係インジェクションで作成された JavaBean の型です。MBean サービスは JBoss EAP サーバーの中心的な要素です。

8.1. JBOSS MBEAN SERVICE の記述

JBoss サービスに依存するカスタム MBean サービスを記述するには、サービスインターフェースメソッドパターンが必要です。JBoss MBean のサービスインターフェースメソッドパターンは **create**、**start**、**stop**、および **destroy** が実行可能である場合に MBean サービスに通知する複数のライフサイクル操作で構成されます。

以下の方法を使用すると依存関係の状態を管理できます。

- MBean で特定のメソッドを呼び出したい場合は、これらのメソッドを MBean インターフェースで宣言します。この方法では、MBean 実装で JBoss 固有クラスの依存関係を回避できます。
- JBoss 固有クラスの依存関係を気にしない場合は、MBean インターフェースで **ServiceMBean** インターフェースおよび **ServiceMBeanSupport** クラスを拡張できます。**ServiceMBeanSupport** クラスは create、start、および stop などのサービスライフサイクルメソッドの実装を提供します。**start()** イベントなどの特定のイベントを処理するには、**ServiceMBeanSupport** クラスによって提供される **startService()** メソッドをオーバーライドする必要があります。

8.1.1. 標準の MBean の例

本項では、サービスアーカイブ (**.sar**) で一緒にパッケージ化される 2 つの MBean 例の開発について説明します。

ConfigServiceMBean インターフェースは **start**、**getTimeout**、および **stop** などの特定のメソッドを宣言し、JBoss 固有のクラスを使用せずに MBean に対して **start**、**hold**、および **stop** を実行します。**ConfigService** クラスは **ConfigServiceMBean** インターフェースを実装した後、このインターフェース内で使用されたメソッドを実装します。

PlainThread クラスは **ServiceMBeanSupport** クラスを拡張し、**PlainThreadMBean** インターフェースを実装します。**PlainThread** はスレッドを開始し、**ConfigServiceMBean.getTimeout()** を使用してスレッドのスリープ時間を決定します。

MBean サービスの例

```
package org.jboss.example.mbean.support;
public interface ConfigServiceMBean {
    int getTimeout();
    void start();
    void stop();
}
package org.jboss.example.mbean.support;
public class ConfigService implements ConfigServiceMBean {
    int timeout;
    @Override
    public int getTimeout() {
        return timeout;
    }
```

```
    @Override
    public void start() {
        //Create a random number between 3000 and 6000 milliseconds
        timeout = (int)Math.round(Math.random() * 3000) + 3000;
        System.out.println("Random timeout set to " + timeout + "
seconds");
    }
    @Override
    public void stop() {
        timeout = 0;
    }
}

package org.jboss.example.mbean.support;
import org.jboss.system.ServiceMBean;
public interface PlainThreadMBean extends ServiceMBean {
    void setConfigService(ConfigServiceMBean configServiceMBean);
}

package org.jboss.example.mbean.support;
import org.jboss.system.ServiceMBeanSupport;
public class PlainThread extends ServiceMBeanSupport implements
PlainThreadMBean {
    private ConfigServiceMBean configService;
    private Thread thread;
    private volatile boolean done;
    @Override
    public void setConfigService(ConfigServiceMBean configService) {
        this.configService = configService;
    }
    @Override
    protected void startService() throws Exception {
        System.out.println("Starting Plain Thread MBean");
        done = false;
        thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (!done) {
                        System.out.println("Sleeping....");
                        Thread.sleep(configService.getTimeout());
                        System.out.println("Slept!");
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
        thread.start();
    }
    @Override
    protected void stopService() throws Exception {
        System.out.println("Stopping Plain Thread MBean");
        done = true;
    }
}
```

jboss-service.xml 記述子は、**inject** タグを使用して **ConfigService** クラスが **PlainThread** クラスにインジェクトされる方法を示します。**inject** タグは **PlainThreadMBean** と **ConfigServiceMBean** 間の依存関係を確立し、**PlainThreadMBean** が簡単に **ConfigServiceMBean** を使用できるようにします。

JBoss-service.xml サービス記述子

```
<server xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:jboss:service:7.0 jboss-service_7_0.xsd"
        xmlns="urn:jboss:service:7.0">
 <mbean code="org.jboss.example.mbean.support.ConfigService"
name="jboss.support:name=ConfigBean"/>
 <mbean code="org.jboss.example.mbean.support.PlainThread"
name="jboss.support:name=ThreadBean">
  <attribute name="configService">
   <inject bean="jboss.support:name=ConfigBean"/>
  </attribute>
 </mbean>
</server>
```

MBean の例を記述した後、クラスと **jboss-service.xml** 記述子をサービスアーカイブ (**.sar**) の **META-INF** でパッケージ化できます。

8.2. JBOSS MBEAN サービスのデプロイ

管理対象ドメインでのサンプル MBean のデプロイおよびテスト

管理対象ドメインでサンプル MBean (**ServiceMBeanTest.sar**) をデプロイするには、以下のコマンドを使用します。

```
deploy ~/Desktop/ServiceMBeanTest.sar --all-server-groups
```

スタンドアロンサーバーでのサンプル MBean のデプロイおよびテスト

スタンドアロンサーバーでサンプル MBean (**ServiceMBeanTest.sar**) をビルドおよびデプロイするには、以下のコマンドを使用します。

```
deploy ~/Desktop/ServiceMBeanTest.sar
```

サンプル MBean のデプロイ解除

サンプル MBean をデプロイ解除するには、以下のコマンドを使用します。

```
ServiceMBeanTest.sar のデプロイ解除
```

第9章 コンカレンシーユーティリティー

コンカレンシーユーティリティーは、Java SE コンカレンシーユーティリティーを Java EE アプリケーション環境仕様で使用できるようにする API であり、JSR 236: Concurrency Utilities for Java™ EE で定義されています。JBoss EAP では、EE コンカレンシーユーティリティーのインスタンスを作成、編集、および削除でき、その結果、これらのインスタンスがアプリケーションで使用できるようになります。

コンカレンシーユーティリティーを使用すると、既存のコンテキストのアプリケーションスレッドをプルし、独自のスレッドで使用することにより、呼び出しコンテキストを拡張できるようになります。呼び出しコンテキストのこの拡張には、デフォルトでクラスローディング、JNDI、およびセキュリティーコンテキストが含まれます。

コンカレンシーユーティリティーのタイプには以下のものが含まれます。

- コンテキストサービス
- 管理対象スレッドファクトリー
- 管理対象エグゼキューターサービス
- 管理対象スケジュール済みエグゼキューターサービス

standalone.xml のコンカレンシーユーティリティー

```
<subsystem xmlns="urn:jboss:domain:ee:4.0">
            <spec-descriptor-property-replacement>false</spec-descriptor-
property-replacement>
            <concurrent>
                <context-services>
                    <context-service name="default" jndi-
name="java:jboss/ee/concurrency/context/default" use-transaction-setup-
provider="true"/>
                </context-services>
                <managed-thread-factories>
                    <managed-thread-factory name="default" jndi-
name="java:jboss/ee/concurrency/factory/default" context-
service="default"/>
                </managed-thread-factories>
                <managed-executor-services>
                    <managed-executor-service name="default" jndi-
name="java:jboss/ee/concurrency/executor/default" context-
service="default" hung-task-threshold="60000" keepalive-time="5000"/>
                </managed-executor-services>
                <managed-scheduled-executor-services>
                    <managed-scheduled-executor-service name="default"
jndi-name="java:jboss/ee/concurrency/scheduler/default" context-
service="default" hung-task-threshold="60000" keepalive-time="3000"/>
                </managed-scheduled-executor-services>
            </concurrent>
            <default-bindings context-
service="java:jboss/ee/concurrency/context/default"
datasource="java:jboss/datasources/ExampleDS" managed-executor-
service="java:jboss/ee/concurrency/executor/default" managed-scheduled-
```

```
executor-service="java:jboss/ee/concurrency/scheduler/default" managed-
thread-factory="java:jboss/ee/concurrency/factory/default"/>
</subsystem>
```

9.1. コンテキストサービス

コンテキストサービス (**javax.enterprise.concurrent.ContextService**) を使用すると、既存のオブジェクトからコンテキストプロキシをビルドできます。コンテキストプロキシにより、コンテキストが作成または呼び出されたとき (呼び出しが元のオブジェクトに転送される前) に他のコンカレンシーユーティリティーによって使用される呼び出しコンテキストが準備されます。

コンテキストサービスコンカレンシーユーティリティーの属性には以下のものが含まれます。

- **name**: すべてのコンテキストサービス内の一意の名前。
- **jndi-name**: JNDI でコンテキストサービスを配置する場所を定義します。
- **use-transaction-setup-provider**: オプション。プロキシオブジェクトを呼び出す場合に、コンテキストサービスによってビルドされたコンテキストプロキシがコンテキストでトランザクションを一時停止するかどうかを示します。デフォルト値は **false** ですが、デフォルトのコンテキストサービスの値は **true** です。

コンテキストサービスコンカレンシーユーティリティーの使用方法については、上記の例を参照してください。

新しいコンテキストサービスの追加

```
/subsystem=ee/context-service=newContextService:add(jndi-
name=java:jboss/ee/concurrency/contextservice/newContextService)
```

コンテキストサービスの変更

```
/subsystem=ee/context-service=newContextService:write-attribute(name=jndi-
name,
value=java:jboss/ee/concurrency/contextservice/changedContextService)
```

この操作にはリロードが必要です。

コンテキストサービスの削除

```
/subsystem=ee/context-service=newContextService:remove()
```

この操作にはリロードが必要です。

9.2. 管理対象スレッドファクトリー

管理対象スレッドファクトリー **(javax.enterprise.concurrent.ManagedThreadFactory)** コンカレンシーユーティリティーを使用すると、Java EE アプリで Java スレッドを作成できます。JBoss EAP は管理対象スレッドファクトリーインスタンスを処理するため、Java EE アプリケーションはライフサイクル関連メソッドを呼び出すことができません。

管理対象スレッドファクトリーコンカレンシーユーティリティーの属性には以下のものがあります。

- **context-service**: すべての管理対象スレッドファクトリー内の一意の名前。
- **jndi-name**: JNDI で管理対象スレッドファクトリーを配置する場所を定義します。
- **priority**: オプション。ファクトリーにより作成された新しいスレッドの優先度を示します。デフォルト値は 5 です。

新しい管理対象スレッドファクトリーの追加

```
/subsystem=ee/managed-thread-factory=newManagedTF:add(context-
service=newContextService, jndi-
name=java:jboss/ee/concurrency/threadfactory/newManagedTF, priority=2)
```

管理対象スレッドファクトリーの変更

```
/subsystem=ee/managed-thread-factory=newManagedTF:write-
attribute(name=jndi-name,
value=java:jboss/ee/concurrency/threadfactory/changedManagedTF)
```

この操作にはリロードが必要です。同様に、他の属性を変更することもできます。

管理対象スレッドファクトリーの削除

```
/subsystem=ee/managed-thread-factory=newManagedTF:remove()
```

この操作にはリロードが必要です。

9.3. 管理対象エグゼキューターサービス

管理対象エグゼキューターサービス **(javax.enterprise.concurrent.ManagedExecutorService)** を使用すると、Java EE アプリケーションで非同期実行向けタスクを送信できます。JBoss EAP は管理対象エグゼキューターサービスインスタンスを処理するため、Java EE アプリケーションはライフサイクル関連メソッドを呼び出すことができません。

管理対象エグゼキューターサービスコンカレンシーユーティリティーの属性には以下のものがあります。

- **context-service**: オプション。既存のコンテキストサービスをその名前で参照します。指定された場合は、参照されたコンテキストサービスがタスクをエグゼキューターに送信したときに存在する呼び出しコンテキストを取得します (このコンテキストはタスクの実行時に使用されます)。
- **jndi-name**: JNDI で管理対象スレッドファクトリーを配置する場所を定義します。
- **max-threads**: エグゼキューターにより使用されるスレッドの最大数を定義します。デフォルト値は **Integer.MAX_VALUE** です。
- **thread-factory**: 既存の管理対象スレッドファクトリーをその名前で参照して内部スレッドの作成を処理します。指定されない場合は、デフォルト設定の管理対象スレッドファクトリーが作成され、内部で使用されます。
- **core-threads**: エグゼキューターのプールに保持するスレッド数を提供します (スレッドはアイドル状態であるものも含みます)。値が 0 の場合は、制限がないことを意味します。

- **keepalive-time**: 内部スレッドをアイドル状態にできる時間 (ミリ秒単位) を定義します。属性のデフォルト値は 60000 です。
- **queue-length**: 入力キューに格納できるタスクの数を示します。デフォルト値は 0 であり、キューの容量が無制限であることを意味します。
- **hung-task-threshold**: ミリ秒単位の時間を定義します。この時間が経過すると、管理対象エグゼキューターサービスによってタスクがハング状態にあると見なされ、強制終了します。値が 0 (デフォルト値) の場合、タスクはハング状態にあると見なされません。
- **long-running-tasks**: 長時間実行中のタスクの実行の最適化を推奨します。デフォルト値は false です。
- **reject-policy**: タスクがエグゼキューターにより拒否されたときに使用するポリシーを定義します。属性値はデフォルトの **ABORT** (例外がスローされます) または **RETRY_ABORT** (エグゼキューターは例外をスローする前にもう一度例外を送信しようとします) のいずれかになります。

新しい管理対象エグゼキューターサービスの追加

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:add(jndi-
name=java:jboss/ee/concurrency/executor/newManagedExecutorService, core-
threads=7, thread-factory=default)
```

管理対象エグゼキューターサービスの変更

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:write-
attribute(name=core-threads,value=10)
```

この操作にはリロードが必要です。同様に、他の属性を変更することもできます。

管理対象エグゼキューターサービスの削除

```
/subsystem=ee/managed-executor-service=newManagedExecutorService:remove()
```

この操作にはリロードが必要です。

9.4. 管理対象スケジュール済みエグゼキューターサービス

管理対象スケジュール済みエグゼキューターサービス **(javax.enterprise.concurrent.ManagedScheduledExecutorService)** を使用すると、Java EE アプリケーションで非同期実行向けタスクをスケジュールできます。JBoss EAP は管理対象スケジュール済みエグゼキューターサービスインスタンスを処理するため、Java EE アプリケーションはライフサイクル関連メソッドを呼び出すことができません。

管理対象エグゼキューターサービスコンカレンシーユーティリティーの属性には以下のものがあります。

- **context-service**: 既存のコンテキストサービスをその名前で参照します。指定された場合は、参照されたコンテキストサービスがタスクをエグゼキューターに送信したときに存在する呼び出しコンテキストを取得します (このコンテキストはタスクの実行時に使用されます)。
- **hung-task-threshold**: ミリ秒単位の時間を定義します。この時間が経過すると、管理対象スケジュール済みエグゼキューターサービスによってタスクがハング状態にあると見なされ、

強制終了します。値が 0 (デフォルト値) の場合、タスクはハング状態にあると見なされません。

- **keepalive-time**: 内部スレッドをアイドル状態にできる時間 (ミリ秒単位) を定義します。属性のデフォルト値は 60000 です。
- **reject-policy**: タスクがエグゼキューターにより拒否されたときに使用するポリシーを定義します。属性値はデフォルトの **ABORT** (例外がスローされます) または **RETRY_ABORT** (エグゼキューターは例外をスローする前にもう一度例外を送信しようとします) のいずれかになります。
- **core-threads**: エグゼキューターのプールに保持するスレッド数を提供します (スレッドはアイドル状態であるものも含みます)。値が 0 の場合は、制限がないことを意味します。
- **jndi-name**: JNDI で管理対象スケジュール済みエグゼキューターサービスを配置する場所を定義します。
- **long-running-tasks**: 長時間実行中のタスクの実行の最適化を推奨します。デフォルト値は false です。
- **thread-factory**: 既存の管理対象スレッドファクトリーをその名前で参照して内部スレッドの作成を処理します。指定されない場合は、デフォルト設定の管理対象スレッドファクトリーが作成され、内部で使用されます。

新しい管理対象スケジュール済みエグゼキューターサービスの追加

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:add(jndi-
name=java:jboss/ee/concurrency/scheduledexecutor/newManagedScheduledExecut
orService, core-threads=7, context-service=default)
```

この操作にはリロードが必要です。

管理対象スケジュール済みエグゼキューターサービスの変更

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:write-attribute(name=core-
threads, value=10)
```

この操作にはリロードが必要です。同様に、他の属性を変更することができます。

管理対象スケジュール済みエグゼキューターサービスの削除

```
/subsystem=ee/managed-scheduled-executor-
service=newManagedScheduledExecutorService:remove()
```

この操作にはリロードが必要です。

第10章 UNDERTOW

10.1. UNDERTOW ハンドラーについて

Undertow は、ブロックタスクと非ブロックタスクの両方に使用するよう設計された Web サーバーです。JBoss EAP 7 では JBoss Web は Undertow に置き換わります。主な機能の一部は以下のとおりです。

- ハイパフォーマンス
- 組み込み可能
- Servlet 3.1
- Web ソケット
- リバースプロキシー

リクエストライフサイクル
クライアントがサーバーに接続するときに、Undertow によって **`io.undertow.server.HttpServerConnection`** が作成されます。クライアントがリクエストを送信するときに、リクエストは Undertow パーサーによって解析され、生成される **`io.undertow.server.HttpServerExchange`** はルートハンドラーに渡されます。ルートハンドラーが完了すると、以下の 4 つのいずれかのことが起こります。

交換が完了する

リクエストチャネルと応答チャネルが完全に読み取られたり、書き込まれた場合に、交換が完了したと見なされます。リクエスト側は、GET や HEAD などのコンテンツがないリクエストの場合に、自動的に完全に読み取られたと見なされます。読み取り側は、ハンドラーが完全な応答を書き込み、応答チャネルを閉じ、応答チャネルを完全にフラッシュしたときに、完了したと見なされます。交換がすでに完了した場合は、どんなアクションも行われません。

交換を完了せずにルートハンドラーが通常どおり返される

この場合、交換は **`HttpServerExchange.endExchange()`** を呼び出して完了します。

ルートハンドラーが例外で返される

この場合、**`500`** の応答コードが設定され、**`HttpServerExchange.endExchange()`** を使用して交換が終了します。

ルートハンドラーは、`HttpServerExchange.dispatch()` が呼び出された後、または非同期 IO が開始された後に返されることがあります。

この場合、ディスパッチされたタスクはディスパッチエグゼキューターに送信されます。また、非同期 IO がリクエストチャネルまたは応答チャネルのいずれかで開始された場合は、このタスクが開始されます。この場合、交換は完了しません。非同期タスクによって、処理が完了したときに交換が完了します。

`HttpServerExchange.dispatch()` の最も一般的な使用方法は、ワーカースレッドに対してブロックが許可されない IO スレッドから実行を移動することです (この結果、ブロック操作が許可されます)。このパターンは通常以下のようになります。

ワーカースレッドへのディスパッチ:

```
public void handleRequest(final HttpServerExchange exchange) throws
Exception {
    if (exchange.isInIoThread()) {
      exchange.dispatch(this);
```

```
            return;
        }
        //handler code
    }
```

交換は呼び出しスタックが返されるまで実際にはディスパッチされないため、交換で一度に複数のスレッドがアクティブにならないようにすることができます。交換はスレッドセーフではありません。ただし、交換は、両方のスレッドが一度に変更しようとしない限り、複数のスレッド間で渡すことができます。

交換の終了

交換を終了するには、リクエストチャネルを読み取り、応答チャネルで **`shutdownWrites()`** を呼び出し、フラッシュする方法と **`HttpServerExchange.endExchange()`** を呼び出す方法の 2 つがあります。**`endExchange()`** が呼び出された場合、Undertow はコンテンツが生成されたかどうかを確認します。生成された場合、Undertow はリクエストチャネルを単にドレインし、応答チャネルを閉じ、フラッシュします。生成されず、交換で登録されたデフォルトの応答リスナーがある場合は、Undertow によってそれらの各応答リスナーがデフォルトの応答を生成できるようになります。このメカニズムにより、デフォルトのエラーページが生成されます。

Undertow の詳細については、JBoss EAP **Configuration Guide** の Configuring the Web Server を参照してください。

10.2. デプロイメントでの既存の UNDERTOW ハンドラーの使用

Undertow は、JBoss EAP にデプロイされたアプリケーションで使用できるハンドラーのデフォルトセットを提供します。利用可能なハンドラーとその属性の完全なリストはここで確認できます。

デプロイメントでハンドラーを使用するには、**`WEB-INF/undertow-handlers.conf`** ファイルを追加する必要があります。

`WEB-INF/undertow-handlers.conf` の例

```
allowed-methods(methods='GET')
```

すべてのハンドラーでは、特定のケースでそのハンドラーを適用するためにオプションの述語を指定することもできます。

オプションの述語がある `WEB-INF/undertow-handlers.conf` の例

```
path('/my-path') -> allowed-methods(methods='GET')
```

上記の例では、**`allowed-methods`** ハンドラーのみがパス **`/my-path`** に適用されます。

一部のハンドラーにはデフォルトのパラメーターがあり、名前を使用せずにハンドラー定義でそのパラメーターの値を指定できます。

デフォルトのパラメーターを使用した `WEB-INF/undertow-handlers.conf` の例

```
path('/a') -> redirect('/b')
```

また、**`WEB-INF/jboss-web.xml`** ファイルを更新して 1 つまたは複数のハンドラーの定義を含めることもできます (ただし、**`WEB-INF/undertow-handlers.conf`** を使用することが推奨されます)。

WEB-INF/jboss-web.xml の例

```
<jboss-web>
    <http-handler>
        <class-
name>io.undertow.server.handlers.AllowedMethodsHandler</class-name>
        <param>
            <param-name>methods</param-name>
            <param-value>GET</param-value>
        </param>
    </http-handler>
</jboss-web>
```

提供された Undertow ハンドラーの完全なリストは、ここで確認できます。

10.3. カスタムハンドラーの作成

- カスタムハンドラーは **WEB-INF/jboss-web.xml** ファイルで定義できます。

```
<jboss-web>
    <http-handler>
        <class-name>org.jboss.example.MyHttpHandler</class-name>
    </http-handler>
</jboss-web>
```

ハンドラークラスの例:

```
package org.jboss.example;

import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;

public class MyHttpHandler implements HttpHandler {
    private HttpHandler next;

    public MyHttpHandler(HttpHandler next) {
        this.next = next;
    }

    public void handleRequest(HttpServerExchange exchange) throws
Exception {
        // do something
        next.handleRequest(exchange);
    }
}
```

- カスタムハンドラーには、**WEB-INF/jboss-web.xml** ファイルを使用してパラメーターを設定することもできます。

```
<jboss-web>
    <http-handler>
        <class-name>org.jboss.example.MyHttpHandler</class-name>
        <param>
```

```
            <param-name>myParam</param-name>
            <param-value>foobar</param-value>
        </param>
    </http-handler>
</jboss-web>
```

これらのパラメーターが機能するには、ハンドラークラスに対応するセッターが必要です。

```
package org.jboss.example;

import io.undertow.server.HttpHandler;
import io.undertow.server.HttpServerExchange;

public class MyHttpHandler implements HttpHandler {
    private HttpHandler next;
    private String myParam;

    public MyHttpHandler(HttpHandler next) {
        this.next = next;
    }

    public void setMyParam(String myParam) {
        this.myParam = myParam;
    }

    public void handleRequest(HttpServerExchange exchange) throws
Exception {
        // do something, use myParam
        next.handleRequest(exchange);
    }
}
```

- ハンドラーの定義に **WEB-INF/jboss-web.xml** を使用する代わりに、ハンドラーは **WEB-INF/undertow-handlers.conf**ファイルで定義することもできます。

  ```
  myHttpHandler(myParam='foobar')
  ```

  WEB-INF/undertow-handlers.conf で定義されたハンドラーが機能するには、以下の 2 つのものを作成する必要があります。

  1. **HandlerWrapper** にラップされた **HandlerBuilder** (**undertow-handlers.conf** 向けの対応する構文を定義し、**HttpHandler** を作成します)。

     ```
     package org.jboss.example;

     import io.undertow.server.HandlerWrapper;
     import io.undertow.server.HttpHandler;
     import io.undertow.server.handlers.builder.HandlerBuilder;

     import java.util.Collections;
     import java.util.Map;
     import java.util.Set;

     public class MyHandlerBuilder implements HandlerBuilder {
         public String name() {
     ```

```
        return "myHttpHandler";
    }

    public Map<String, Class<?>> parameters() {
        return Collections.<String, Class<?
>>singletonMap("myParam", String.class);
    }

    public Set<String> requiredParameters() {
        return Collections.emptySet();

    }

    public String defaultParameter() {
        return null;

    }

    public HandlerWrapper build(final Map<String, Object> config)
{
        return new HandlerWrapper() {
            public HttpHandler wrap(HttpHandler handler) {
                MyHttpHandler result = new
MyHttpHandler(handler);
                result.setMyParam((String)
config.get("myParam"));
                return result;
            }
        };
    }
}
```

2. ファイル **META-INF/services/io.undertow.server.handlers.builder.HandlerBuilder** 内のエントリー。このファイルは、たとえば **WEB-INF/classes** のクラスパス上にある必要があります。

```
org.jboss.example.MyHandlerBuilder
```

第11章 JAVA トランザクション API (JTA)

11.1. 概要

11.1.1. Java トランザクション API (JTA) の概要

はじめに

これらのトピックは、 Java トランザクション API (JTA) の基礎的な内容について取り上げます。

- Java Transactions API (JTA) について
- トランザクションライフサイクル
- JTA トランザクションの例

11.2. トランザクションの概念

11.2.1. トランザクション

トランザクションは 2 つ以上のアクションで構成されており、アクションすべてが成功または失敗する必要があります。成功した場合はコミット、失敗した場合はロールバックが結果的に実行されます。ロールバックでは、トランザクションがコミットを試行する前に、各メンバーの状態が元の状態に戻ります。

よく設計されたトランザクションの通常の標準は Atomic, Consistent, Isolated, and Durable (ACID) です。

11.2.2. トランザクションの ACID プロパティー

ACID は 原子性 (**`Atomicity`**)、一貫性 (**`Consistency`**)、独立性 (**`Isolation`**)、永続性 (**`Durability`**) の略語です。通常、この用語はデータベースやトランザクション操作において使用されます。

原子性 (Atomicity)

トランザクションの原子性を保つには、すべてのトランザクションメンバーが同じ決定を行う必要があります。これらのメンバーはコミットまたはロールバックを行います。原子性が保たれない場合の結果はヒューリスティックな結果と呼ばれます。

一貫性

一貫性とは、データベーススキーマの観点から、データベースに書き込まれたデータが有効なデータであることを保証するという意味です。データベースあるいは他のデータソースは常に一貫した状態でなければなりません。一貫性のない状態の例には、操作が中断される前にデータの半分が書き込まれてしまったフィールドなどがあります。すべてのデータが書き込まれた場合や、書き込みが完了しなかった時に書き込みがロールバックされた場合に、一貫した状態となります。

独立性 (Isolation)

独立性とは、トランザクションのスコープ外のプロセスがデータを変更できないように、トランザクションで操作されたデータが変更前にロックされる必要があることを意味します。

永続性 (Durability)

永続性とは、トランザクションのメンバーにコミットの指示を出してから外部で問題が発生した場合、問題が解決されると全メンバーがトランザクションのコミットを継続できるという意味です。ここで言う問題とは、ハードウェア、ソフトウェア、ネットワークなどのシステムが関係する問題のことです。

11.2.3. トラザクションコーディネーターまたはトランザクションマネージャー

JBoss EAP のトランザクションでは、トランザクションコーディネーターとトランザクションマネージャー (TM) という言葉は、ほとんど同じことを意味します。トランザクションコーディネーターという言葉は通常、分散 JTS トランザクションのコンテキストで使用されます。

JTA トランザクションでは、TM は JBoss EAP 内で実行され、2 フェーズコミットのプロトコルでトランザクションの参加者と通信します。

TM はトランザクションの参加者に対して、他のトランザクションの参加者の結果に従い、データをコミットするか、ロールバックするか指示します。こうすることで、確実にトランザクションが ACID 標準に準拠するようにします。

- トランザクションの参加者
- トランザクションの ACID プロパティー
- 2 フェーズコミットプロトコル

11.2.4. トランザクションの参加者

トランザクションの参加者は、状態をコミットまたはロールバックできるトランザクション内のリソースであり、一般的にデータベースまたは JMS ブローカーを生成します。ただし、トランザクションインターフェースを実装することにより、ユーザーコードがトランザクションの参加者として動作することもできます。トランザクションの各参加者は、状態をコミットまたはロールバックできるかどうかを独自に決定します。すべての参加者がコミットできる場合のみ、トランザクション全体が成功します。コミットできない参加者がある場合は、各参加者がそれぞれの状態をロールバックし、トランザクション全体が失敗します。TM は、コミットおよびロールバック操作を調整し、トランザクションの結果を判断します。

11.2.5. Java Transactions API (JTA)

Java Transactions API (JTA) は Java Enterprise Edition 仕様の一部で、JSR-907 に定義されています。

JTA の実装は、JBoss EAP アプリケーションサーバーの Narayana プロジェクトに含まれる TM を使用して実行されます。TM により、単一のグローバルトランザクションを使用してアプリケーションがさまざまなリソース (データベースや JMS ブローカーなど) を割り当てることができるようになります。グローバルトランザクションは XA トランザクションと呼ばれます。一般的に、このようなトランザクションには XA 機能を持つリソースが含まれますが、XA 以外のリソースをグローバルトランザクションに含めることもできます。XA 以外のリソースを XA 対応リソースとして動作させるのに役に立つ複数の最適化があります。詳細については、1 フェーズコミットの LRCO 最適化 を参照してください。

本書では、JTA という用語は以下の 2 つのことを意味します。

1. Java EE 仕様で定義された Java トランザクション API
2. TM がトランザクションをどのように処理するかを示します。
   TM は JTA トランザクションモードで動作し、データはメモリーによって共有されます。また、トランザクションコンテキストはリモート EJB 呼び出しによって転送されます。JTS モードでは、データは CORBA (Common Object Request Broker Architecture) メッセージを送信して共有され、トランザクションコンテキストは IIOP 呼び出しによって転送されます。複数の JBoss EAP サーバー上におけるトランザクションの分散は両方のモードでサポートされます。

   - 分散トランザクション
   - XA データソースおよび XA トランザクション

11.2.6. Java Transaction Service (JTS)

Java Transaction Service (JTS) は、Object Transaction Service (OTS) と Java のマッピングです。Java EE アプリケーションは JTA API を使用してトランザクションを管理します。JTA API はトランザクションマネージャーが JTS モードに切り替わったときに JTS トランザクション実装と対話します。JTS は IIOP プロトコル上で動作します。JTS を使用するトランザクションマネージャーは Object Request Broker (ORB) と呼ばれるプロセスと Common Object Request Broker Architecture (CORBA) と呼ばれる通信標準を使用してお互いに通信します。詳細については、JBoss EAP **Configuration Guide** の ORB Configuration を参照してください。

アプリケーションの観点で JTA API を使用すると、JTS トラザクションは JTA トランザクションと同じように動作します。

注記

JBoss EAP に含まれる JTS の実装は、分散トランザクションをサポートします。完全準拠の JTS トランザクションとの違いは、外部のサードパーティー ORB との相互運用性です。この機能は、JBoss EAP ではサポートされません。サポートされる設定では、複数の JBoss EAP コンテナーでのみトランザクションが分散されます。

11.2.7. XA リソースおよび XA トランザクション

XA は eXtended Architecture を表し、複数のバックエンドデータストアを使用するトランザクションを定義するために X/Open Group によって開発されました。XA 標準は、グローバル TM とローカルリソースマネージャーとの間のインターフェースを定義します。XA では、4 つの ACID プロパティーすべてを保持しながらアプリケーションサーバー、データベース、キャッシュ、メッセージキューなどの複数のリソースが同じトランザクションに参加できるようにします。4 つの ACID プロパティーの 1 つは原子性であり、これは参加者の 1 つが変更のコミットに失敗した場合に他の参加者がトランザクションを中止し、トランザクションが発生する前の状態に戻すことを意味します。XA リソースは XA グローバルトランザクションに参加できるリソースです。

XA トランザクションは、複数のリソースにまたがることができるトランザクションです。これには、コーディネートを行う TM が関係します。この TM は、すべてが 1 つのグローバル XA トランザクションに関与する 1 つ以上のデータベースまたは他のトランザクションリソースを持ちます。

11.2.8. XA リカバリー

TM は X/Open XA 仕様を実装し、複数の XA リソースで XA トランザクションをサポートします。

XA リカバリーは、トランザクションの参加者であるリソースのいずれかがクラッシュしたり使用できなくなったりしても、トランザクションの影響を受けたすべてのリソースが確実に更新またはロールバックされるようにするプロセスのことです。JBoss EAP の範囲内では、XA データソース、JMS メッセージキュー、JCA リソースアダプターなどの XA リソースまたはサブシステムに対して、`transactions` サブシステムが XA リカバリーのメカニズムを提供します。

XA リカバリーはユーザーの介入がなくても実行されます。XA リカバリーに失敗すると、エラーがログ出力に記録されます。サポートが必要な場合は、Red Hat グローバルサポートサービスまでご連絡ください。XA リカバリープロセスは、デフォルトで 2 分ごとに開始される定期リカバリースレッドにより開始されます。定期リカバリースレッドにより、未完了のすべてのトランザクションが処理されます。

注記

不明なトランザクションのリカバリーを完了するには 4〜8 分かかることがあります。これはリカバリープロセスを複数回実行する必要がある場合があるためです。

11.2.9. XA リカバリープロセスの制限

XA リカバリーには以下の制限があります。

トランザクションログが正常にコミットされたトランザクションから消去されないことがある

XAResource のコミットメソッドが正常に完了し、トランザクションをコミットしてからコーディネーターがログをアップデートするまでに JBoss EAP サーバーがクラッシュすると、サーバーの再起動時に以下の警告メッセージが表示されることがあります。

```
ARJUNA016037: Could not find new XAResource to use for recovering non-
serializable XAResource XAResourceRecord
```

これは、リカバリー時に JBoss トランザクションマネージャー (TM) はログのトランザクション参加者を確認し、コミットを再試行しようとするからです。最終的に、JBoss TM はリソースがコミットされたと見なし、コミットを再試行しなくなります。このような場合、トランザクションはコミットされデータの損失はないため、警告を無視しても問題ありません。

警告が表示されないようにするには、**`com.arjuna.ats.jta.xaAssumeRecoveryComplete`** プロパティーの値を **`true`** に設定します。このプロパティーは、登録された XAResourceRecovery インスタンスから新しい XAResource インスタンスが見つからないとチェックされます。**`true`** に設定すると、リカバリーで、以前のコミットの試行が成功したと見なされ、これ以上リカバリーを試行しなくてもインスタンスをログから削除できます。このプロパティーはグローバルであり、適切に使用しないと XAResource インスタンスがコミットされていない状態のままになるため、注意して使用する必要があります。

注記

JBoss EAP 7.0 には、トランザクションが正常にコミットされた後にトランザクションログを消去する拡張機能が実装されているため、上記の状況は頻繁に発生しません。

XAResource.prepare() の最後にサーバーがクラッシュすると、JTS トランザクションに対するロールバックは呼び出されません。

XAResource **`prepare()`** メソッド呼び出しの完了後に JBoss EAP サーバーがクラッシュすると、参加している XAResources はすべて準備済みの状態でロックされ、サーバーの再起動時にその状態のままになります。トランザクションがタイムアウトするか、DBA で手動でリソースをロールバックしてトランザクションログを消去するまで、トランザクションはロールバックされずリソースはロックされたままになります。詳細については、https://issues.jboss.org/browse/JBTM-2124 を参照してください。

周期リカバリーはコミットされたトランザクションで発生する可能性があります。

サーバーに過剰な負荷がかかっている場合、サーバーログには以下の警告メッセージとそれに続くスタックトレースが含まれる場合があります。

```
ARJUNA016027: Local XARecoveryModule.xaRecovery got XA exception
XAException.XAER_NOTA: javax.transaction.xa.XAException
```

負荷が大きいと、トランザクションの処理時間と周期リカバリープロセスの活動が重なることがあります。周期リカバリープロセスは進行中のトランザクションを検出してロールバックを実行しようとしますが、トランザクションは完了するまで続行されます。周期リカバリーがロールバックの試行に失敗すると、ロールバックの失敗をサーバーログに記録します。この問題の原因は今後のリリースで修正される予定ですが、問題の回避策を適用できます。

`com.arjuna.ats.jta.orphanSafetyInterval` プロパティーの値をデフォルト値の 10000 ミ

リ秒よりも大きくし、リカバリープロセスの 2 つのフェーズの間隔を増やします。40000 ミリ秒の値が推奨されます。この設定では問題は解決されず、問題が発生して警告メッセージがログに記録される可能性が減少することに注意してください。詳細については、https://developer.jboss.org/thread/266729 を参照してください。

11.2.10. 2 フェーズコミットプロトコル

2 フェーズコミット (2PC) プロトコルは、トランザクションの結果を決定するアルゴリズムを参照します。2PC は XA トランザクションを完了するプロセスとしてトランザクションマネージャー (TM) によって開始されます。

フェーズ 1: 準備
最初のフェーズでは、トランザクションをコミットできるか、あるいはロールバックする必要があるかをトランザクションの参加者がトランザクションコーディネーターに通知します。

フェーズ 2: コミット
2 番目のフェーズでは、トランザクションコーディネーターがトランザクション全体をコミットするか、またはロールバックするかを決定します。いずれの参加者もコミットできない場合、トランザクションはロールバックしなければなりません。それ以外の場合、トランザクションはコミットできます。コーディネーターは何を行うかをリソースに指示し、リソースはその完了時にコーディネーターに通知します。この時点で、トランザクションは完了します。

11.2.11. トランザクションタイムアウト

原子性を確保し、トランザクションを ACID 標準に準拠させるために、トランザクションの一部が長期間実行される場合があります。トランザクションの参加者は、コミット時にデータベーステーブルまたはキュー内のメッセージの一部である XA リソースをロックする必要があります。また、TM は各トランザクション参加者からの応答を待ってからすべての参加者にコミットあるいはロールバックの指示を出す必要があります。ハードウェアあるいはネットワークの障害のため、リソースが永久にロックされることがあります。

トランザクションのタイムアウトをトランザクションと関連付け、ライフサイクルを制御することができます。タイムアウトのしきい値がトランザクションのコミットあるいはロールバック前に渡された場合、タイムアウトにより、自動的にトランザクションがロールバックされます。

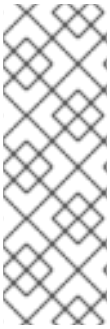
トランザクションサブシステム全体に対しデフォルトのタイムアウト値を設定できます。または、デフォルトのタイムアウト値を無効にし、トランザクションごとにタイムアウトを指定できます。

11.2.12. 分散トランザクション

分散トランザクションは、複数の JBoss EAP サーバー上に参加者が存在するトランザクションです。Java Transaction Service (JTS) 仕様では、異なるベンダーのアプリケーションサーバー間で JTS トランザクションを分散可能にすることが規定されています。Java Transaction API (JTA) はこれを定義していませんが、JBoss EAP は JBoss EAP サーバー間での分散 JTA トランザクションをサポートしています。

注記

異なるベンダーのサーバー間でのトランザクション分散はサポートされません。



注記

他のベンダーのアプリケーションサーバーのドキュメントでは、分散トランザクションという用語が XA トランザクションを意味することがあります。JBoss EAP のドキュメントでは、複数の JBoss EAP アプリケーションサーバー間で分散されるトランザクションを分散トランザクションと呼びます。また、本書では、異なるリソースで構成されるトランザクション (データベースリソースや jms リソースなど) を XA トランザクションと呼びます。詳細については、Java Transaction Service (JTS) および XA データソースおよび XA トランザクション を参照してください。

11.2.13. ORB 移植性 API

Object Request Broker (ORB) とは、複数のアプリケーションサーバーで分散されるトランザクションの参加者、コーディネーター、リソース、および他のサービスにメッセージを送受信するプロセスのことです。ORB は標準的なインターフェース記述言語 (IDL) を使用してメッセージを通信し解釈します。Common Object Request Broker Architecture (CORBA) は JBoss EAP の ORB によって使用される IDL です。

ORB を使用する主なタイプのサービスは、Java トランザクションサービス (JTS) 仕様を使用する分散 Java トランザクションのシステムです。レガシーシステムなどの他のシステムは、通信にリモートエンタープライズ JavaBeans や JAX-WS または JAX-RS Web サービスなどの他のメカニズムではなく ORB を使用することがあります。

ORB 移植性 API は ORB とやりとりするメカニズムを提供します。この API は ORB への参照を取得するメソッドや、ORB からの受信接続をリッスンするモードにアプリケーションを置くメソッドを提供します。API のメソッドの一部はすべての ORB によってサポートされていません。このような場合、例外が発生します。

API は 2 つの異なるクラスによって構成されます。

- **`com.arjuna.orbportability.orb`**
- **`com.arjuna.orbportability.oa`**

ORB 移植性 API に含まれるメソッドとプロパティーの詳細については、Red Hat カスタマーポータルの JBoss EAP Javadocs バンドルを参照してください。

11.3. トランザクションの最適化

11.3.1. トランザクション最適化の概要

JBoss EAP のトランザクションマネージャー (TM) には、アプリケーションで利用できる複数の最適化機能が含まれています。

最適化機能は、特定のケースで 2 フェーズコミットプロトコルを拡張するために使用します。一般的に、TM によって、2 フェーズコミットを介して渡されるグローバルトランザクションが開始されます。ただし、特定のケースでこれらのトランザクションを最適化する場合は、TM によって完全な 2 フェーズコミットを行う必要がないため、処理は速くなります。

TM により使用される別の最適化機能は、以下で詳細に説明されています。

- 1 フェーズコミット (1PC) の LRCO 最適化
- 推定中止 (presumed-abort) の最適化

- 読み取り専用の最適化

11.3.2. 1 フェーズコミット (1PC) の LRCO 最適化

1 フェーズコミット (1PC)
トランザクションでは、2 フェーズコミットプロトコル (2PC) がより一般的に使用されますが、両フェーズに対応する必要がなかったり、対応できない場合もあります。そのような場合、1 フェーズコミット (1PC) プロトコルを使用できます。1 フェーズコミットプロトコルは、XA または非 XA リソースの 1 つがグローバルトランザクションの一部である場合に使用されます。

一般的に、準備フェースでは、第 2 フェーズが処理されるまでリソースがロックされます。1 フェーズコミットは、準備フェースが省略され、コミットのみがリソースに対して処理されることを意味します。指定されない場合は、グローバルトランザクションに参加者が 1 つだけ含まれるときに 1 フェーズコミット最適化機能が自動的に使用されます。

最終リソースコミット最適化 (LRCO: Last Resource Commit Optimization)
非 XA データソースが XA トランザクションに参加する場合は、最終リソースコミット最適化 (LRCO) と呼ばれる最適化機能が使用されます。このプロトコルにより、ほとんどのトランザクションは正常に完了しますが、一部のエラーによってトランザクションの結果の一貫性が失われることがあります。そのため、この方法は最終手段として使用してください。

非 XA リソースは準備フェーズの終了時に処理され、コミットが試行されます。コミットに成功すると、トランザクションログが書き込まれ、残りのリソースがコミットフェーズに移動します。最終リソースがコミットに失敗すると、トランザクションはロールバックされます。

ローカルの TX データソースが 1 つのみトランザクションで使用されると、LRCO が自動的に適用されます。

これまでは、LRCO メソッドを使用して非 XA リソースを XA トランザクションに追加していました。ただし、この場合は LRCO によって失敗することがあります。LRCO メソッドを用いて非 XA リソースを XA トランザクションに追加するには、以下の手順に従います。

1. XA トランザクションの準備
2. LRCO のコミット
3. tx ログの書き込み
4. XA トランザクションのコミット

手順 2 と手順 3 の間でクラッシュした場合、データが不整合になり、XA トランザクションをコミットできなくなることがあります。データの不整合が発生する理由は、LRCO 非 XA リソースがコミットされるが、XA リソースの準備に関する情報が記録されなかったことです。リカバリーマネージャーはサーバーの起動後にリソースをロールバックします。CMR ではこの制限がなくなり、非 XA が XA トランザクションに安定して参加できるようになります。

注記

CMR は特別な LRCO 最適化機能であり、データソースにのみ使用できます。非 XA リソースには適していません。

- 2 フェーズコミットプロトコル

11.3.2.1. Commit Markable Resource (CMR)

概要
Commit Markable Resource (CMR) インターフェースを使用してリソースマネージャーへのアクセスを設定すると、非 XA データソースを XA (2PC) トランザクションに安定的に登録できます。これは、非 XA リソースを完全にリカバリー可能にする LRCO アルゴリズムの実装です。

CMR を設定するには、以下のことを行う必要があります。

1. データベースでテーブルを作成します。
2. データソースを接続可能にします。
3. 参照を **transactions** サブシステムに追加します。

データベースでテーブルを作成する
トランザクションには CMR リソースを 1 つだけ含めることができます。以下の SQL が機能するテーブルを作成する必要があります。

```
SELECT xid,actionuid FROM _tableName_ WHERE transactionManagerID IN
(String[])
DELETE FROM _tableName_ WHERE xid IN (byte[[]])
INSERT INTO _tableName_ (xid, transactionManagerID, actionuid) VALUES
(byte[],String,byte[])
```

SQL クエリーの例

Sybase:

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64),
actionuid varbinary(28))
```

Oracle:

```
CREATE TABLE xids (xid RAW(144), transactionManagerID varchar(64),
actionuid RAW(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

IBM:

```
CREATE TABLE xids (xid VARCHAR(255) for bit data not null,
transactionManagerID
varchar(64), actionuid VARCHAR(255) for bit data not null)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

SQL Server:

```
CREATE TABLE xids (xid varbinary(144), transactionManagerID varchar(64),
actionuid varbinary(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

Postgres:

```
CREATE TABLE xids (xid bytea, transactionManagerID varchar(64), actionuid
bytea)
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

MariaDB:

```
CREATE TABLE xids (xid BINARY(144), transactionManagerID varchar(64),
actionuid BINARY(28))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

MySQL:

```
CREATE TABLE xids (xid VARCHAR(255), transactionManagerID varchar(64),
actionuid VARCHAR(255))
CREATE UNIQUE INDEX index_xid ON xids (xid)
```

データソースを接続可能にする
デフォルトでは、CMR 機能はデータソースに対して無効になっています。有効にするには、データソースの設定を作成または変更し、**connectible** 属性を **true** に設定する必要があります。以下に、サーバーの XML 設定ファイルのデータソースセクションの例を示します。

```
<datasource enabled="true" jndi-
name="java:jboss/datasources/ConnectableDS" pool-name="ConnectableDS"
jta="true" use-java-context="true" connectable="true"/>
```

注記

この機能は XA データソースには適用されません。

また、以下のように管理 CLI を使用してリソースマネージャーを CMR として有効にすることもできます。

```
/subsystem=datasources/data-source=ConnectableDS:add(enabled="true", jndi-
name="java:jboss/datasources/ConnectableDS", jta="true", use-java-
context="true", connectable="true", connection-url="validConnectionURL",
exception-
sorter="org.jboss.jca.adapters.jdbc.extensions.mssql.MSSQLExceptionSorter"
, driver-name="h2")
```

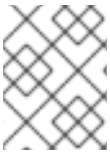
新しい CMR 機能を使用するために既存のリソースを更新
CMR 機能を使用するために既存のデータソースのみを更新する必要がある場合は、**connectable** 属性を変更します。

```
/subsystem=datasources/data-source=ConnectableDS:write-
attribute(name=connectable,value=true)
```

参照をトランザクションサブシステムに追加する
transactions サブシステムは、以下のような **transactions** サブシステム設定セクションへのエントリーを用いて CMR 対応のデータソースを特定します。

```
<subsystem xmlns="urn:jboss:domain:transactions:3.0">
    ...
    <commit-markable-resources>
        <commit-markable-resource jndi-
name="java:jboss/datasources/ConnectableDS">
            <xid-location name="xids" batch-size="100" immediate-
cleanup="false"/>
```

```
        </commit-markable-resource>
        ...
    </commit-markable-resources>
</subsystem>
```



注記

transactions サブシステムで CMR 参照を追加したら、サーバーを再起動する必要があります。

注記

データソース設定で **exception-sorter** パラメーターを使用します。詳細については、JBoss EAP **Configuration Guide** の Example Datasource Configurations を参照してください。

11.3.3. 推定中止 (presumed-abort) の最適化

トランザクションをロールバックする場合、この情報をローカルで記録し、エンリストされたすべての参加者に通知します。この通知は形式的なもので、トランザクションの結果には影響しません。すべての参加者が通知されると、このトランザクションに関する情報を削除できます。

トランザクションのステータスに対する後続の要求が行われる場合、利用可能な情報はありません。このような場合、要求側はトランザクションが中断され、ロールバックされたと見なします。推定中止 (presumed-abort) の最適化は、トランザクションがコミットの実行を決定するまで参加者に関する情報を永続化する必要がないことを意味します。これは、トランザクションがコミットの実行を決定する前に発生した障害はトランザクションの中止であると推定されるためです。

11.3.4. 読み取り専用の最適化

参加者は、準備するよう要求されると、トランザクション中に変更したデータがないことをコーディネーターに伝えることができます。参加者が最終的にどうなってもトランザクションに影響を与えることはないため、このような参加者にトランザクションの結果について通知する必要はありません。この読み取り専用の参加者はコミットプロトコルの第 2 フェーズから省略可能です。

11.4. トランザクションの結果

11.4.1. トランザクションの結果

可能なトランザクションの結果は次の 3 つになります。

コミット

トランザクションの参加者すべてがコミットできる場合、トランザクションコーディネーターはコミットの実行を指示します。詳細については、トランザクションのコミット を参照してください。

ロールバック

トランザクションの参加者のいずれかがコミットできなかったり、トランザクションコーディネーターが参加者にコミットを指示できない場合は、トランザクションがロールバックされます。詳細については、トランザクションロールバックを参照してください。

ヒューリスティックな結果

トランザクションの参加者の一部がコミットし、他の参加者がロールバックした場合をヒューリスティックな結果と呼びます。詳細については、ヒューリスティックな結果を参照してください。

11.4.2. トランザクションのコミット

トランザクションの参加者がコミットすると、新規の状態が永続化されます。新規の状態はトランザクションで作業を行った参加者により作成されます。トランザクションのメンバーがデータベースに記録を書き込む時などが最も一般的な例になります。

コミット後、トランザクションの情報はトランザクションコーディネーターから削除され、新たに書き込まれた状態が永続状態となります。

11.4.3. トランザクションロールバック

トランザクションの参加者はトランザクションの開始前に、状態を反映するため状態をリストアし、ロールバックを行います。ロールバック後の状態はトランザクション開始前の状態と同じになります。

11.4.4. ヒューリスティックな結果

ヒューリスティックな結果 (アトミックでない結果) は、トランザクションでの参加者の決定がトランザクションマネージャーのものとは異なる状況です。ヒューリスティックな結果が起こると、システムの整合性が保たれなくなることがあり、通常、解決に人的介入が必要になります。ヒューリスティックな結果に依存するようなコードは記述しないようにしてください。

通常、ヒューリスティックな結果は、2 フェーズコミット (2PC) プロトコルの 2 番目のフェーズで発生します。まれにですが、この結果は 1PC で発生することがあります。多くの場合、これは基盤のハードウェアまたは基盤のサーバーの通信サブシステムの障害によって引き起こされます。

ヒューリスティックな結果は、さまざまなサブシステムまたはリソースのタイムアウトにより可能になります (トランザクションマネージャーと完全なクラッシュリカバリーを使用)。何らかの形の分散契約が必要なシステムでは、グローバルな結果という点でシステムのいくつかの部分が分岐する状況が発生することがあります。

ヒューリスティックな結果には 4 種類あります。

ヒューリスティックロールバック
コミット操作はリソースをコミットできませんでしたが、すべての参加者はロールバックでき、アトミックな結果が実現されました。

ヒューリスティックコミット
参加者のすべてが一方的にコミットしたため、ロールバック操作に失敗します。たとえば、コーディネーターが正常にトランザクションを準備したにも関わらず、ログ更新の失敗などでコーディネーター側で障害が発生したため、ロールバックの実行を決定した場合などに発生します。暫定的に参加者がコミットの実行を決定する場合があります。

ヒューリスティック混合
一部の参加者がコミットし、その他の参加者はロールバックした状態です。

ヒューリスティックハザード
更新の一部の配置が不明な状態です。既知の更新結果はすべてコミットまたはロールバックされています。

- 2 フェーズコミットプロトコル

11.4.5. JBoss Transactions エラーと例外

UserTransaction クラスのメソッドがスローする例外に関する詳細については、http://docs.oracle.com/javaee/7/api/javax/transaction/UserTransaction.html の **UserTransaction API** の仕様を参照してください。

11.5. トランザクションライフサイクルの概要

11.5.1. トランザクションライフサイクル

Java Transactions API (JTA) の詳細については、Java Transactions API (JTA) を参照してください。

リソースがトランザクションへの参加を要求すると、一連のイベントが開始されます。トランザクションマネージャー (TM) は、アプリケーションサーバー内に存在するプロセスであり、トランザクションを管理します。トランザクションの参加者は、トランザクションに参加するオブジェクトです。リソースは、データソース、JMS 接続ファクトリー、または他の JCA 接続です。

1. **アプリケーションが新しいトランザクションを開始する**
   トランザクションを開始するために、アプリケーションは JNDI から (EJB の場合はアノテーションから) UserTransaction クラスのインスタンスを取得します。UserTransaction インターフェースには、トップレベルのトランザクションを開始、コミット、およびロールバックするメソッドが含まれています。新規作成されたトランザクションは、そのトランザクションを呼び出すスレッドと自動的に関連付けされます。ネストされたトランザクションは JTA ではサポートされないため、すべてのトランザクションがトップレベルのトランザクションとなります。

   `UserTransaction.begin()` メソッドが呼び出されると、EJB がトランザクションを開始します。このトランザクションのデフォルトの動作は **`TransactionAttribute`** アノテーションまたは **`ejb.xml`** 記述子の使用によって影響を受けることがあります。この時点以降に使用されたリソースは、このトランザクションと関連付けられます。2 つ以上のリソースが登録された場合、トランザクションは XA トランザクションになり、コミット時に 2 フェーズコミットプロトコルに参加します。

   注記

   デフォルトでは、トランザクションは EJB のアプリケーションコンテナーによって駆動されます。これは **Container Managed Transaction (CMT)** と呼ばれます。トランザクションをユーザー駆動にするには、**`Transaction Management`** を **Bean Managed Transaction (BMT)** に変更する必要があります。BMT では、**`UserTransaction`** オブジェクトはユーザーがトランザクションを管理するために使用できます。

2. **アプリケーションが状態を変更する**
   次の手順では、アプリケーションが作業を実行して状態を変更します (登録されたリソースに対してのみ)。

3. **アプリケーションがコミットまたはロールバックすることを決定する**
   アプリケーションの状態の変更が完了すると、アプリケーションはコミットするか、またはロールバックするかを決定し、適切なメソッド (**`UserTransaction.commit()`** または **`UserTransaction.rollback()`**) を呼び出します。CMT の場合、このプロセスは自動的に駆動されますが、BMT の場合は、**`UserTransaction`** のメソッドコミットまたはロールバックを明示的に呼び出す必要があります。

4. **TM がレコードからトランザクションを削除する**
   コミットあるいはロールバックが完了すると、TM はレコードをクリーンアップし、トランザクションログからトランザクションに関する情報を削除します。

障害リカバリー

リソース、トランザクションの参加者、またはアプリケーションサーバーがクラッシュするか、使用できなくなった場合は、障害が解決され、リソースが再度使用できるようになったときに **`Transaction`**

Manager がリカバリーを実行します。このプロセスは自動的に実行されます。詳細については、XA リカバリーを参照してください。

11.6. トランザクションサブシステムの設定

11.6.1. トランザクション設定の概要

はじめに
transactions サブシステムでは、統計、タイムアウト値、トランザクションロギングなどのトランザクションマネージャー (TM) のオプションを設定できます。

詳細については、JBoss EAP **Configuration Guide** の Transactions Subsystem Configuration を参照してください。

11.6.2. トランザクションマネージャーの設定

トランザクションマネージャーは、Web ベースの管理コンソールまたはコマンドライン管理 CLI を使用して設定できます。

管理コンソールを使用したトランザクションマネージャーの設定
以下の手順は、Web ベースの管理コンソールを使用してトランザクションマネージャーを設定する方法を示しています。

1. 画面上部の **Configuration** タブを選択します。
2. JBoss EAP を管理対象ドメインとして実行している場合は、変更する任意のプロファイルを選択します。
3. **Subsystem** リストから、**Transactions** を選択し、**View** をクリックします。
4. 編集する設定に応じたタブ (リカバリーオプションの場合の **Recovery** など) で **Edit** をクリックします。
5. 必要な変更を行い、**Save** をクリックして変更を保存します。
6. **Need Help?** をクリックしてヘルプテキストを表示します。

管理 CLI を使用したトランザクションマネージャーの設定
管理 CLI で一連のコマンドを使用してトランザクションマネージャーを設定できます。これらのコマンドはすべて **/subsystem=transactions** (スタンドアロンサーバー向け) または **/profile=default/subsystem=transactions/** (管理対象ドメインの **default** プロファイル向け) で始まります。

トランザクションマネージャーのすべての設定オプションの詳細なリストについては、トランザクションマネージャーの設定オプションを参照してください。

11.6.3. トランザクションロギング

11.6.3.1. トランザクションログメッセージ

トランザクションロガーに **DEBUG** ログレベルを使用することにより、ログファイルを読み取り可能な状態に保ちつつトランザクションを追跡できます。詳細なデバッグの場合は、**TRACE** ログレベルを使用します。トランザクションロガーの設定に関する詳細については、トランザクションサブシステムのロギング設定を参照してください。

TRACE ログレベルでログを記録するよう設定すると、トランザクションマネージャー (TM) は多くのロギング情報を生成できます。最も一般的なメッセージの一部は次のとおりです。このリストは包括的ではなく、他のメッセージが表示されることもあります。

表11.1 トランザクション状態の変更

トランザクションの開始	トランザクションが開始されたら、クラス **com.arjuna.ats.arjuna.coordinator.BasicAction** のメソッド **Begin** が実行され、メッセージ **BasicAction::Begin() for action-id <transaction uid>** でログに示されます。
トランザクションのコミット	トランザクションがコミットされたら、クラス **com.arjuna.ats.arjuna.coordinator.BasicAction** のメソッド **Commit** が実行され、メッセージ **BasicAction::Commit() for action-id <transaction uid>** でログに示されます。
トランザクションのロールバック	トランザクションがロールバックされたら、クラス **com.arjuna.ats.arjuna.coordinator.BasicAction** のメソッド **Rollback** が実行され、メッセージ **BasicAction::Rollback() for action-id <transaction uid>** でログに示されます。
トランザクションのタイムアウト	トランザクションがタイムアウトすると、**com.arjuna.ats.arjuna.coordinator.TransactionReaper** のメソッド **doCancellations** が実行され、**Reaper Worker <thread id> attempting to cancel <transaction uid>** とログに示されます。この結果、上記のように同じスレッドがトランザクションをロールバックすることが示されます。

11.6.3.2. トランザクションサブシステムのロギング設定

JBoss EAP の他のログ設定に依存せずにログに記録されたトランザクションに関する情報の量を制御できます。ログ設定は、管理コンソールまたは管理 CLI を使用して設定できます。

管理コンソールを使用したトランザクションロガーの設定

1. **ロギング**サブシステム設定に移動します。
   a. 管理コンソールで、**Configuration** タブをクリックします。管理対象ドメインを使用する場合は、最初に適切なサーバープロファイルを選択する必要があります。
   b. **Logging** サブシステムを選択し、**View** をクリックします。
2. **com.arjuna** 属性を編集します。
   Log Categories タブを選択します。**com.arjuna** エントリーがすでに存在します。**com.arjuna** を選択し、**Attributes** セクションの **Edit** をクリックします。ログレベルを変更し、親ハンドラーを使用するかどうかを選択できます。
   - ログレベル:
     トランザクションにより大量のロギング出力が生成されることがあるため、サーバーのログがトランザクション出力で満たされないようデフォルトのロギングレベルは **WARN** に設定されます。トランザクション処理の詳細を確認する必要がある場合は、トランザクション ID が表示されるよう **TRACE** ログレベルを使用します。

- 親ハンドラーの使用:
  親ハンドラーはロガーが出力を親ロガーに送信するかどうかを指定します。デフォルトの動作は **true** です。

3. **Save** をクリックして変更を保存します。

管理 CLI を使用したトランザクションロガーの設定
以下のコマンドを使用して管理 CLI からログレベルを設定します。スタンドアロンサーバーの場合は、コマンドから **/profile=default** を削除します。

```
/profile=default/subsystem=logging/logger=com.arjuna:write-
attribute(name=level,value=VALUE)
```

11.6.4. トランザクションの参照と管理

管理 CLI では、トランザクションレコードを参照および操作する機能がサポートされます。この機能は、TM と JBoss EAP の管理 API 間の対話によって提供されます。

TM は、待機中の各トランザクションとトランザクションに関連する参加者に関する情報を、オブジェクトストアと呼ばれる永続ストレージに格納します。管理 API は、オブジェクトストアを **log-store** と呼ばれるリソースとして公開します。**probe** と呼ばれる API 操作はトランザクションログを読み取り、各レコードに対してノードパスを作成します。**probe** コマンドは、**log-store** を更新する必要があるときに、いつでも手動で呼び出すことができます。トランザクションログが即座に表示され、非表示になるのは、正常な挙動です。

ログストアの更新
このコマンドは、管理対象ドメインでプロファイル **default** を使用するサーバーグループに対してログストアを更新します。スタンドアローンサーバーの場合は、コマンドから **profile=default** を削除します。

```
/profile=default/subsystem=transactions/log-store=log-store/:probe
```

準備済みトランザクションすべての表示
準備されたすべてのトランザクションを表示するには、最初にログストアを更新し (ログストアの更新 を参照)、ファイルシステムの **ls** コマンドに類似した機能を持つ次のコマンドを実行します。

```
ls /profile=default/subsystem=transactions/log-store=log-
store/transactions
```

または、

```
/host=master/server=server-one/subsystem=transactions/log-store=log-
store:read-children-names(child-type=transactions)
```

各トランザクションが一意の ID とともに表示されます。個々の操作は、各トランザクションに対して実行できます (トランザクションの管理 を参照)。

11.6.4.1. トランザクションの管理

トランザクションの属性の表示
JNDI 名、EIS 製品名およびバージョン、ステータスなどのトランザクションに関する情報を表示するには、**:read-resource** 管理 CLIコマンドを使用します。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:read-resource
```

トランザクションの参加者の表示
各トランザクションログには、**participants** (参加者) と呼ばれる子要素が含まれます。トランザクションの参加者を確認するには、この要素に対して **read-resource** 管理 CLI コマンドを使用します。参加者は JNDI 名によって識別されます。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9/participants=java\:\/JmsXA:read-resource
```

結果は以下のようになります。

```
{
   "outcome" => "success",
   "result" => {
       "eis-product-name" => "ActiveMQ",
       "eis-product-version" => "2.0",
       "jndi-name" => "java:/JmsXA",
       "status" => "HEURISTIC",
       "type" => "/StateManager/AbstractRecord/XAResourceRecord"
   }
}
```

ここで示された結果ステータスは **HEURISTIC** であり、リカバリーが可能です。詳細については、トランザクションのリカバリーを参照してください。

特別な場合では、ログに対応するトランザクションレコードがないオーファンレコード (XAResourceRecords) をオブジェクトストアに作成できます。たとえば、準備済みに XA リソースが TM が記録する前にクラッシュし、ドメイン管理 API ではアクセスできません。このようなレコードにアクセスするには、管理オプション **expose-all-logs** を **true** に設定する必要があります。このオプションは管理モデルには保存されず、サーバーが再起動されると **false** にリストアされます。

```
/profile=default/subsystem=transactions/log-store=log-store:write-
attribute(name=expose-all-logs, value=true)
```

代わりに以下のコマンドを実行すると、トラザクション参加者 ID が集約された形式で表示されます。

```
/host=master/server=server-one/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:read-children-
names(child-type=participants)
```

トランザクションの削除
各トランザクションログは、トランザクションを表すトランザクションログを削除する **:delete** 操作をサポートします。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:delete
```

トランザクションのリカバリー
トランザクションの各参加者は、**:recover** 管理 CLI コマンドを使用したリカバリーをサポートします。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9/participants=2:recover
```

- トランザクションの状態が **HEURISTIC** である場合は、リカバリー操作によって状態が **PREPARE** に変わり、リカバリーがトリガーされます。
- トランザクションの参加者の 1 つがヒューリスティックな場合、リカバリー操作は **commit** 操作を再実行しようとします。成功した場合、トランザクションログから参加者が削除されます。これを確認するには、**log-store** 上で **:probe** 操作を再実行し、参加者がリストされていないことを確認します。これが最後の参加者の場合は、トランザクションも削除されます。

リカバリーが必要なトランザクションの状態を更新します。
トランザクションをリカバリーする必要がある場合は、リカバリーを試行する前に **:refresh** 管理 CLI コマンドを使用して、トランザクションのリカバリーが必要であるかを確認できます。

```
/profile=default/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9/participants=2:refresh
```

11.6.5. トランザクション統計情報の表示

トランザクションマネージャーの統計が有効になっていると、トランザクションマネージャーにより処理されたトラザクションの統計を表示できます。トランザクションマネージャーの統計を有効にする方法については、トランザクションマネージャーの設定を参照してください。

管理コンソールまたは管理 CLI を使用して統計を表示できます。管理コンソールでは、トラザクションの統計は **Runtime** タブから **Transactions** サブシステムに移動することにより利用できます。管理コンソールではすべての統計を利用できるわけではありません。

以下の表は、利用できる統計とその説明を示しています。

表11.2 トランザクションサブシステムの統計

統計	説明
number-of-transactions	このサーバー上でトランザクションマネージャーにより処理されるトランザクションの合計数。
number-of-committed-transactions	このサーバー上でトランザクションマネージャーにより処理されるコミット済みトランザクションの数。
number-of-aborted-transactions	このサーバー上でトランザクションマネージャーにより処理されるアボートされたトランザクションの数。
number-of-timed-out-transactions	このサーバー上でトランザクションマネージャーにより処理されるタイムアウトのトランザクションの数。タイムアウトしたトランザクションの数はアボートされたトランザクションの数まで計算されます。
number-of-heuristics	ヒューリスティック状態のトランザクションの数。
number-of-inflight-transactions	開始した未完了のトランザクションの数。

統計	説明
number-of-application-rollbacks	障害の原因がアプリケーションであった失敗トランザクションの数。
number-of-resource-rollbacks	障害の原因がリソースであった失敗トランザクションの数。

11.7. 実際のトランザクションの使用

11.7.1. トランザクション使用の概要

次の手順は、アプリケーションでトランザクションを使用する必要がある場合に役に立ちます。

- トランザクションの制御
- トランザクションの開始
- トランザクションのコミット
- トランザクションのロールバック
- トランザクションでのヒューリスティックな結果の処理
- トランザクションマネージャーの設定
- トランザクションエラーの処理

11.7.2. トランザクションの制御

はじめに

この手順のリストでは、JTS API を使用するアプリケーションでトランザクションを制御するさまざまな方法を概説します。

- トランザクションの開始
- トランザクションのコミット
- トランザクションのロールバック
- JTA トランザクションの例

11.7.3. トランザクションの開始

この手順では、新しいトランザクションの開始方法を示します。実行するトランザクションマネージャー (TM) が JTA または JTS のいずれかで設定されていれば API は同じになります。

1. **UserTransaction のインスタンスを取得する**
   `@TransactionManagement(TransactionManagementType.BEAN)` アノテーションを用いると、JNDI、インジェクション、または EJB のコンテキストを使用してインスタンスを取得できます (EJB が Bean 管理のトランザクションを使用する場合)。

   a. **JNDI**

```
new InitialContext().lookup("java:comp/UserTransaction")
```

b. **インジェクション**

```
@Resource UserTransaction userTransaction;
```

c. **コンテキスト**

- ステートレス/ステートフル Bean の場合

```
@Resource SessionContext ctx;
ctx.getUserTransaction();
```

- メッセージ駆動型 Bean の場合

```
@Resource MessageDrivenContext ctx;
ctx.getUserTransaction()
```

2. データソースへの接続後に UserTransaction.begin() を呼び出す

```
try {
    System.out.println("\nCreating connection to database: "+url);
    stmt = conn.createStatement();  // non-tx statement
    try {
        System.out.println("Starting top-level transaction.");
        userTransaction.begin();
        stmtx = conn.createStatement(); // will be a tx-statement
        ...
    }
}
```

JTS 仕様を使用して既存のトランザクションに参加する

EJB (CMT または BMT のいずれかと使用) の利点の 1 つは、コンテナーがトランザクション処理の内部をすべて管理することです。そのため、ユーザーは JBoss EAP コンテナー間の XA トランザクションまたはトランザクションディストリビューションの一部であるトランザクションを処理する必要がありません。

結果

トランザクションが開始します。トランザクションをコミットまたはロールバックするまで、データソースのすべての使用でトランザクションに対応します。

完全な例については、JTA トランザクションの例 を参照してください。

11.7.4. ネストされたトランザクション

ネストされたトランザクションを用いると、アプリケーションは既存のトランザクションに埋め込まれるトランザクションを作成できます。このモデルでは、再帰的に複数のサブトランザクションをトランザクションに埋め込むことができます。親トランザクションをコミットまたはロールバックせずにサブトランザクションをコミットまたはロールバックできます。しかし、コミット操作の結果は、先祖のトランザクションがすべてコミットしたかどうかによって決まります。

実装固有の情報については、Narayana プロジェクトドキュメンテーションを参照してください。

ネストされたトランザクションは、JTS 仕様と使用した場合のみ利用できます。ネストされたトランザクションは JBoss EAP アプリケーションサーバーではサポートされない機能です。また、多くのデータベースベンダーがネストされたトランザクションをサポートしないため、ネストされたトランザクションをアプリケーションに追加する前にデータベースベンダーにお問い合わせください。

11.7.5. トランザクションのコミット

この手順では、Java Transaction API (JTA) を使用してトランザクションをコミットする方法を説明します。

前提条件

トランザクションは、コミットする前に開始する必要があります。トランザクションの開始方法については、トランザクションの開始を参照してください。

1. **UserTransaction の commit() メソッドを呼び出す**
   UserTransaction の commit() メソッドを呼び出す場合、TM はトランザクションをコミットしようとします。

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value) {
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin();
        <!-- Perform some data manipulation using entityManager -->
        ...
        // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
        try {
            userTransaction.rollback();
        } catch (SystemException se) {
            throw new RuntimeException(se);
        }
        throw new RuntimeException(ex);
    } finally {
        entityManager.close();
    }
}
```

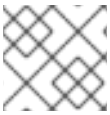
2. **Container Managed Transactions (CMT) を使用する場合は、手動でコミットする必要がない**
   Bean がコンテナー管理トランザクションを使用するよう設定すると、コンテナーはコードで設定したアノテーションに基づいてトランザクションライフサイクルを管理します。

```
@PersistenceContext
private EntityManager em;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void updateTable(String key, String value)
```

```
    <!-- Perform some data manipulation using entityManager -->
    ...
}
```

結果

データソースがコミットされ、トランザクションが終了します。そうでない場合は、例外が発生します。



注記

完全な例については、JTA トランザクションの例 を参照してください。

11.7.6. トランザクションのロールバック

この手順では、Java Transaction API (JTA) を使用してトランザクションをロールバックする方法を説明します。

前提条件

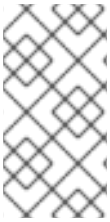
トランザクションは、ロールバックする前に開始する必要があります。トランザクションの開始方法については、トランザクションの開始を参照してください。

1. **UserTransaction の rollback() メソッドを呼び出す**
   `UserTransaction` の **`rollback()`** メソッドを呼び出す場合、TM はトランザクションをロールバックし、データを前の状態に戻そうとします。

```
@Inject
private UserTransaction userTransaction;

public void updateTable(String key, String value)
    EntityManager entityManager =
entityManagerFactory.createEntityManager();
    try {
        userTransaction.begin():
        <!-- Perform some data manipulation using entityManager -->
          ...
          // Commit the transaction
        userTransaction.commit();
    } catch (Exception ex) {
        <!-- Log message or notify Web page -->
        ...
        try {
            userTransaction.rollback();
        } catch (SystemException se) {
            throw new RuntimeException(se);
        }
        throw new RuntimeException(e);
    } finally {
        entityManager.close();
    }
}
```

2. **コンテナー管理トランザクション (CMT) を使用する場合は、手動でトランザクションをロールバックする必要がない**

Bean がコンテナー管理トランザクションを使用するよう設定すると、コンテナーはコードで設定したアノテーションに基づいてトランザクションライフサイクルを管理します。



注記

CMT のロールバックは RuntimeException が発生すると実行されます。setRollbackOnly メソッドを明示的に呼び出してロールバックを発生させることもできます。または、 アプリケーション例外の @ApplicationException(rollback=true) を使用してロールバックできます。

結果

トランザクションは TM によりロールバックされます。

注記

完全な例については、JTA トランザクションの例 を参照してください。

11.7.7. トランザクションにおけるヒューリスティックな結果の処理方法

ヒューリスティックなトランザクションの結果はよく発生するものではなく、通常は例外的な原因が存在します。ヒューリスティックという言葉は「手動」を意味し、こうした結果は通常手動で処理する必要があります。トランザクションのヒューリスティックな結果については、ヒューリスティックな結果を参照してください。

この手順では、Java Transaction API (JTA) を使用してトランザクションのヒューリスティックな結果を処理する方法を説明します。

1. 原因を調べる: トランザクションのヒューリスティックな結果の全体的な原因は、リソースマネージャーがコミットまたはロールバックの実行を約束したにも関わらず、約束を守らなかったことにあります。原因としては、サードパーティーコンポーネント、サードパーティーコンポーネントと JBoss EAP 6 間の統合レイヤー、または JBoss EAP 6 自体の問題が考えられます。
   ヒューリスティックなエラーの最も一般的な原因として圧倒的に多いのが、環境の一時的な障害とリソースマネージャーを扱うコードのコーディングエラーの 2 つです。

2. 環境の一時的な障害を修復する: 通常、環境内で一時的な障害が発生した場合は、ヒューリスティックなエラーを発見する前に気づくはずです。原因としては、ネットワークの停止、ハードウェア障害、データベース障害、電源異常などが考えられます。
   ストレステストの実施中にテスト環境でヒューリスティックな結果が発生した場合は、使用している環境の脆弱性に関する情報が提供されます。

警告

JBoss EAP は、障害発生時に非ヒューリスティックな状態にあるトランザクションの自動リカバリーを行いますが、ヒューリスティックなトランザクションのリカバリーを試行しません。

3. リソースマネージャーのベンダーに連絡する: 環境に明らかな障害がない場合や、ヒューリスティックな結果が容易に再現可能な場合は、コーディングエラーである可能性があります。

サードパーティーのベンダーに連絡して、解決方法の有無を確認してください。JBoss EAP の TM 自体に問題があると思われる場合は、Red Hat グローバルサポートサービスまでご連絡ください。

4. 管理 CLI から手動でトランザクションの回復を試行します。詳細については、トランザクションのリカバリーを参照してください。

5. テスト環境でログを削除し、JBoss EAP を再起動する: テスト環境である場合や、データの整合性を気にしない場合は、トランザクションログを削除して JBoss EAP を再起動すると、ヒューリスティックな結果はなくなります。デフォルトでは、トランザクションログの場所はスタンドアロンサーバーでは **`EAP_HOME/standalone/data/tx-object-store/`**、管理対象ドメインでは **`EAP_HOME/domain/servers/SERVER_NAME/data/tx-object-store`** になります。管理対象ドメインの SERVER_NAME は、サーバーグループに参加している個々のサーバー名を示しています。

注記

トランザクションログの場所は、使用中のオブジェクトストアや **`oject-store-relative-to`** パラメーターおよび **`object-store-path`** パラメーターの値セットによっても異なります。ファイルシステムログ (標準のシャドウや Apache ActiveMQ Artemis ログなど) では、デフォルトの方向の場所が使用されますが、JDBC オブジェクトストアを使用する場合、トランザクションログはデータベースに保存されます。

6. 手動で結果を解決する: トランザクションの結果を手動で解決するプロセスは、障害の厳密な状況によって大きく左右されます。通常は、以下の手順に従って、それぞれの状況に適用する必要があります。

   a. 関連するリソースマネージャーを特定する。

   b. TM の状態とリソースマネージャーを調べる。

   c. 関与する 1 つ以上のコンポーネント内でログのクリーンアップとデータ調整を手動で強制する。
   これらの手順を実行する方法の詳細は、本書の範囲外となります。

11.7.8. JTA トランザクションのエラー処理

11.7.8.1. トランザクションエラーの処理

トランザクションエラーは、多くの場合、タイミングに依存するため、解決するのが困難です。以下に、一部の一般的なエラーと、これらのエラーのトラブルシューティングに関するヒントを示します。

注記

これらのガイドラインはヒューリスティックエラーに適用されません。ヒューリスティックエラーが発生した場合は、トランザクションにおけるヒューリスティックな結果の処理方法 を参照し、Red Hat グローバルサポートサービスまでお問い合わせください。

トランザクションがタイムアウトになったが、ビジネスロジックスレッドが認識しませんでした。

多くの場合、このようなエラーは、Hibernate がレイジーロードのためにデータベース接続を取得できない場合に発生します。頻繁に発生する場合は、タイムアウト値を増加できます。トランザクションマネージャーの設定 を参照してください。

引き続き問題が解決されない場合は、パフォーマンスを向上させるために外部環境を調整するか、さらに効率的になるようコードを再構築できます。タイムアウトの問題が解消されない場合は、Red Hat グローバルサポートサービスにお問い合わせください。

トランザクションがすでにスレッドで実行されているか、NotSupportedException 例外が発生する

NotSupportedException 例外は、通常、JTA トランザクションをネストしようとし、ネストがサポートされていないことを示します。トランザクションをネストしようとしないときは、多くの場合、スレッドプールタスクで別のトランザクションが開始されますが、トランザクションを中断または終了せずにタスクが終了します。
通常、アプリケーションは、これを自動的に処理する **UserTransaction** を使用します。その場合は、フレームワークに問題があることがあります。

コードで **TransactionManager** メソッドまたは **Transaction** メソッドを直接使用する場合は、トランザクションをコミットまたはロールバックするときに次の動作に注意してください。コードで **TransactionManager** メソッドを使用してトランザクションを制御する場合は、トランザクションをコミットまたはロールバックすると、現在のスレッドからトランザクションの関連付けが解除されます。ただし、コードで **Transaction** メソッドを使用する場合は、トランザクションを、実行中のスレッドに関連付けることができず、スレッドプールにスレッドを返す前にスレッドからトランザクションの関連付けを手動で解除する必要があります。

2 番目のローカルリソースを登録することはできません。

このエラーは、2 番目の非 XA リソースをトランザクションに登録しようとした場合に、発生します。1 つのトランザクションで複数のリソースが必要な場合、それらのリソースは XA である必要があります。

11.8. トランザクションに関するリファレンス

11.8.1. JTA トランザクションの例

この例では、 JTA トランザクションを開始、コミット、およびロールバックする方法を示します。使用している環境に合わせて接続およびデータソースパラメーターを調整し、データベースで 2 つのテストテーブルをセットアップする必要があります。

```
public class JDBCExample {
    public static void main (String[] args) {
        Context ctx = new InitialContext();
        // Change these two lines to suit your environment.
        DataSource ds = (DataSource)ctx.lookup("jdbc/ExampleDS");
        Connection conn = ds.getConnection("testuser", "testpwd");
        Statement stmt = null; // Non-transactional statement
        Statement stmtx = null; // Transactional statement
        Properties dbProperties = new Properties();

        // Get a UserTransaction
        UserTransaction txn = new
InitialContext().lookup("java:comp/UserTransaction");

        try {
            stmt = conn.createStatement();  // non-tx statement

            // Check the database connection.
            try {
                stmt.executeUpdate("DROP TABLE test_table");
```

```
                stmt.executeUpdate("DROP TABLE test_table2");
            }
            catch (Exception e) {
                throw new RuntimeException(e);
                // assume not in database.
            }

            try {
                stmt.executeUpdate("CREATE TABLE test_table (a INTEGER,b
INTEGER)");
                stmt.executeUpdate("CREATE TABLE test_table2 (a INTEGER,b
INTEGER)");
            }
            catch (Exception e) {
                throw new RuntimeException(e);
            }

            try {
                System.out.println("Starting top-level transaction.");

                txn.begin();

                stmtx = conn.createStatement(); // will be a tx-statement

                // First, we try to roll back changes

                System.out.println("\nAdding entries to table 1.");

                stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES
(1,2)");

                ResultSet res1 = null;

                System.out.println("\nInspecting table 1.");

                res1 = stmtx.executeQuery("SELECT * FROM test_table");

                while (res1.next()) {
                    System.out.println("Column 1: "+res1.getInt(1));
                    System.out.println("Column 2: "+res1.getInt(2));
                }
                System.out.println("\nAdding entries to table 2.");

                stmtx.executeUpdate("INSERT INTO test_table2 (a, b) VALUES
(3,4)");
                res1 = stmtx.executeQuery("SELECT * FROM test_table2");

                System.out.println("\nInspecting table 2.");

                while (res1.next()) {
                    System.out.println("Column 1: "+res1.getInt(1));
                    System.out.println("Column 2: "+res1.getInt(2));
                }

                System.out.print("\nNow attempting to rollback
changes.");
```

```
                txn.rollback();

                // Next, we try to commit changes
                txn.begin();
                stmtx = conn.createStatement();
                System.out.println("\nAdding entries to table 1.");
                stmtx.executeUpdate("INSERT INTO test_table (a, b) VALUES
(1,2)");
                ResultSet res2 = null;

                System.out.println("\nNow checking state of table 1.");

                res2 = stmtx.executeQuery("SELECT * FROM test_table");

                while (res2.next()) {
                    System.out.println("Column 1: "+res2.getInt(1));
                    System.out.println("Column 2: "+res2.getInt(2));
                }

                System.out.println("\nNow checking state of table 2.");

                stmtx = conn.createStatement();

                res2 = stmtx.executeQuery("SELECT * FROM test_table2");

                while (res2.next()) {
                    System.out.println("Column 1: "+res2.getInt(1));
                    System.out.println("Column 2: "+res2.getInt(2));
                }

                txn.commit();
            }
            catch (Exception ex) {
                throw new RuntimeException(ex);

            }
        }
        catch (Exception sysEx) {
            sysEx.printStackTrace();
            System.exit(0);
        }
    }
}
```

11.8.2. トランザクション API ドキュメンテーション

トランザクション JTA API ドキュメンテーションは以下の場所で javadoc として利用できます。

- UserTransaction - http://docs.oracle.com/javaee/7/api/javax/transaction/UserTransaction.html

Red Hat JBoss Developer Studio を使用してアプリケーションを開発する場合は、 API ドキュメンテーションが **Help** メニューに含まれています。

第12章 JAVA 永続 API (JPA)

12.1. JAVA 永続 API (JPA) について

Java Persistence API (JPA) は、Java オブジェクトまたはクラスとリレーショナルデータベース間でデータのアクセス、永続化、および管理を行うための Java 仕様です。この JPA 仕様では、透過オブジェクトまたはリレーショナルマッピングのパラダイムが考慮されます。オブジェクトまたはリレーショナル永続化メカニズムに必要な基本的な API とメタデータが標準化されます。

注記

JPA 自体は製品ではなく仕様にすぎません。それ自体では永続化やその他の処理を実行できません。JPA はインターフェースセットにすぎず、実装を必要とします。

12.2. HIBERNATE CORE

Hibernate Core は、Java 言語のオブジェクトリレーショナルマッピングフレームワークです。これは、オブジェクト指向ドメインモデルをリレーショナルデータベースにマッピングするためのフレームワークを提供するため、アプリケーションはデータベースとの直接対話を回避できます。Hibernate では、直接的な永続データベースアクセスを高レベルオブジェクト処理関数に置き換えることにより、オブジェクトリレーショナルインピーダンスの不一致の問題が解決されます。

12.3. HIBERNATE ENTITYMANAGER

Hibernate EntityManager を使用すると、Java Persistence 2.1 仕様で定義されたように、プラグラミングインターフェースとライフサイクルルールが実装されます。このラッパーを Hibernate Annotations とともに使用することにより、 成熟した Hibernate Core の上に完全な (およびスタンドアロンの) JPA 永続化ソリューションが実装されます。プロジェクトのビジネス上のニーズまたは技術的なニーズに応じて、これら 3 つすべて、JPA プログラミングインターフェースなしのアノテーション、または純粋なネイティブ Hibernate Core の組み合わせを使用できます。いつでも Hibernate ネイティブ API、または必要な場合はネイティブ JDBC および SQL を使用できます。また、JBoss EAP が完全な Java 永続化ソリューションとともに提供されます。

JBoss EAP は Java Persistence 2.1 仕様に完全準拠しています。また、Hibernate はこの仕様に追加機能を提供します。JPA と JBoss EAP を使用するには、JBoss EAP に同梱されている **`bean-validation`**、**`greeter`**、および **`kitchensink`** クイックスタートを参照してください。クイックスタートのダウンロードおよび実行方法については、クイックスタートサンプルの使用を参照してください。

JPA の永続化は EJB 3 またはより新しい CDI、Java Context and Dependency Injection などのコンテナーと特定のコンテナーの外部で実行されるスタンドアロン Java SE アプリケーションで利用できます。両方の環境で以下のプログラミングインターフェースとアーティファクトを利用できます。

EntityManagerFactory

エンティティーマネージャーファクトリーはエンティティーマネージャーインスタンスを提供し、すべてのインスタンスは、特定の実装などで定義されたのと同じデフォルト設定を使用するために同じデータベースに接続するよう設定されます。複数のエンティティーマネージャーファクトリーを準備して複数のデータストアにアクセスできます。このインターフェースはネイティブ Hibernate の SessionFactory に似ています。

EntityManager

EntityManager API は、特定の作業単位でデータベースにアクセスするために使用されます。また、永続エンティティーインスタンスを作成および削除してプライマリーキー ID でエンティティーを見つけたり、すべてのエンティティーに対してクエリーを実行したりするためにも使用されます。こ

のインターフェースは、Hibernate のセッションに似ています。

永続コンテキスト

永続コンテキストは、永続エンティティー ID が一意のエンティティーインスタンスであるエンティティーインスタンスのセットです。永続コンテキスト内で、エンティティーインスタンスとそのライフサイクルは特定のエンティティーマネージャーによって管理されます。このコンテキストのスコープはトランザクションまたは拡張された作業単位のいずれかです。

永続ユニット

該当するエンティティーマネージャーで管理できるエンティティータイプのセットは、永続ユニットにより定義されます。永続ユニットは、アプリケーションで関連付けまたはグループ化され、単一データストアに対するマッピングで併置する必要があるすべてのクラスのセットを定義します。

コンテナー管理エンティティーマネージャー

ライフサイクルがコンテナーにより管理されるエンティティーマネージャー。

アプリケーション管理エンティティーマネージャー

ライフサイクルがアプリケーションにより管理されるエンティティーマネージャー。

JTA エンティティーマネージャー

JTA トランザクションに関与するエンティティーマネージャー。

リソースローカルエンティティーマネージャー

リソーストランザクション (JTA トランザクションではない) を使用するエンティティーマネージャー。

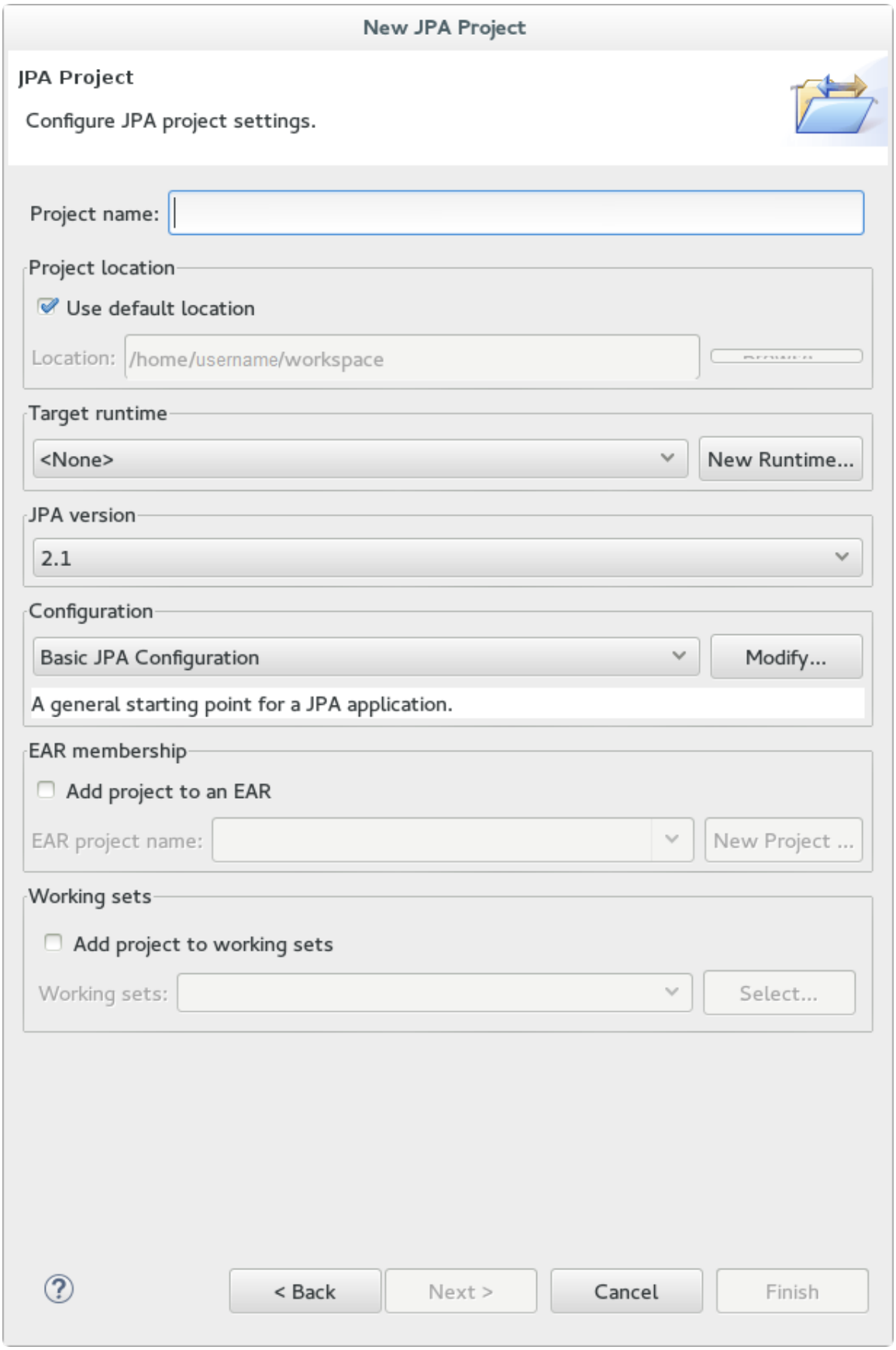
12.4. 単純な JPA アプリケーションの作成

Red Hat Developer Studio で単純な JPA アプリケーションを作成する場合は、以下の手順を実行します。

1. JBoss Developer Studio で JPA プロジェクトを作成します。

    a. Red Hat JBoss Developer Studio で、**File**-→ **New** -→ **Project** をクリックします。リストで **JPA** を見つけ、展開し、**JPA Project** を選択します。以下のダイアログが表示されます。

図12.1 新規 JPA プロジェクトダイアログ



b. **プロジェクト名**を入力します。

c. **ターゲットランタイム**を選択します。ターゲットランタイムがない場合は、Define New Server を使用した JBoss EAP Server の追加 に記載された手順に従って新しいサーバーとランタイムを定義します。

d. **JPA version (JPA バージョン)** de **2.1** が選択されていることを確認します。

e. **Configuration (設定)** で **Basic JPA Configuration (基本的な JPA 設定)** を選択します。

f. **Finish** をクリックします。

g. 要求されたら、このタイプのプロジェクトを JPA パースペクティブウインドウに関連付けるかどうかを選択します。

2. 新しい永続性設定ファイルを作成および設定します。

a. Red Hat JBoss Developer Studio で EJB 3.x プロジェクトを開きます。

b. **Project Explorer (プロジェクトエクスプローラー)** パネルでプロジェクトルートディレクトリーを右クリックします。

c. **New (新規)** → **Other (その他)....** を選択します。

d. XML フォルダーから **XML File (XML ファイル)** を選択し、**Next (次へ)** をクリックします。

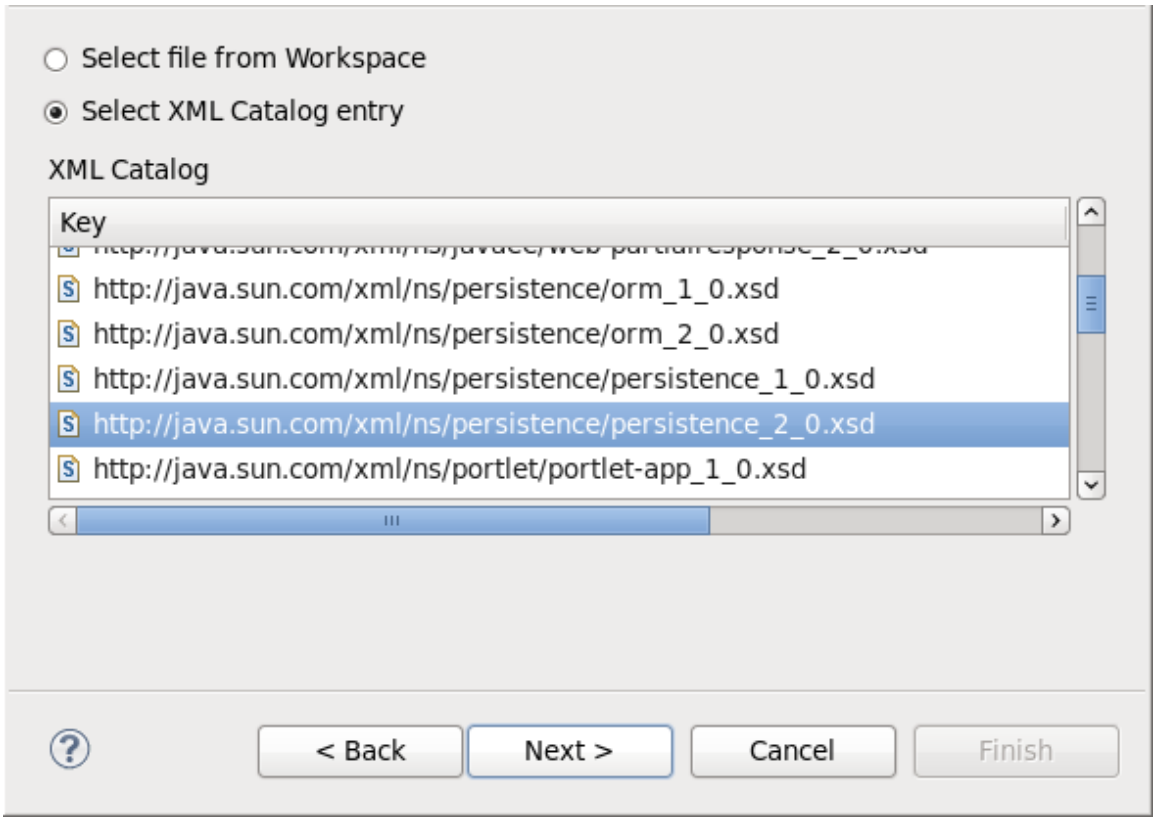
e. 親ディレクトリーとして **`ejbModule/META-INF/`** フォルダーを選択します。

f. ファイルの名前を **`persistence.xml`** と指定し、**Next (次へ)** をクリックします。

g. **Create XML file from an XML schema file (XML スキーマファイルから XML ファイルを作成)** を選択し、**Next (次へ)** をクリックします。

h. **Select XML Catalog entry (XML カタログエントリーの選択)** リストから **http://java.sun.com/xml/ns/persistence/persistence_2.0.xsd** を選択し、**Next (次へ)** をクリックします。

図12.2 永続 XML スキーマ



i. Finish (完了) をクリックしてファイルを作成します。**persistence.xml** が **META-INF/** フォルダーに作成され、設定可能な状態になります。

永続設定ファイルの例

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
   version="2.0">
   <persistence-unit name="example" transaction-type="JTA">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-
data-source>
      <mapping-file>ormap.xml</mapping-file>
      <jar-file>TestApp.jar</jar-file>
      <class>org.test.Test</class>
      <shared-cache-mode>NONE</shared-cache-mode>
      <validation-mode>CALLBACK</validation-mode>
      <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-
drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

12.5. CONFIGURATION (設定)

12.5.1. Hibernate 設定プロパティー

表12.1 Hibernate Java プロパティー

プロパティー名	説明
hibernate.dialect	Hibernate の **org.hibernate.dialect.Dialect** のクラス名。Hibernate で、特定のリレーショナルデータベースに最適化された SQL を生成できるようになります。 ほとんどのケースで、Hibernate は、JDBC ドライバーにより返された JDBC メタデータに基づいて正しい **org.hibernate.dialect.Dialect** 実装を選択できます。
hibernate.show_sql	ブール変数。SQL ステートメントをすべてコンソールに書き込みます。これは、ログカテゴリー **org.hibernate.SQL** を **debug** に設定することと同じです。
hibernate.format_sql	ブール変数。SQL をログとコンソールにプリティプリントします。

プロパティー名	説明
hibernate.default_schema	修飾されていないテーブル名を、生成された SQL の該当するスキーマ/テーブルスペースで修飾します。
hibernate.default_catalog	修飾されていないテーブル名を、生成された SQL の該当するカタログで修飾します。
hibernate.session_factory_name	org.hibernate.SessionFactory が、作成後に JNDI のこの名前に自動的にバインドされます。たとえば、**`jndi/composite/name`** のようになります。
hibernate.max_fetch_depth	シングルエンドのアソシエーション (1 対 1 や多対 1 など) に対して外部結合フェッチツリーの最大の深さを設定します。**`0`** を設定するとデフォルトの外部結合フェッチが無効になります。推奨値は、**`0`** から **`3`** までの値です。
hibernate.default_batch_fetch_size	関連付けの Hibernate 一括フェッチに対するデフォルトサイズを設定します。推奨値は、**`4`**、 **`8`**、および **`16`** です。
hibernate.default_entity_mode	この **`SessionFactory`** から開かれたすべてのセッションに対するエンティティー表現のデフォルトモードを設定します。値には **`dynamic-map`**、 **`dom4j`**、 **`pojo`** などがあります。
hibernate.order_updates	ブール変数。Hibernate で、更新されるアイテムの主キー値で SQL 更新の順番付けを行います。これにより、高度な並列システムにおけるトランザクションデッドロックが軽減されます。
hibernate.generate_statistics	ブール変数。有効にすると、Hibernate がパフォーマンスのチューニングに役に立つ統計情報を収集します。
hibernate.use_identifier_rollback	ブール変数。有効にすると、オブジェクトが削除されたときに、生成された識別子プロパティーがデフォルト値にリセットされます。
hibernate.use_sql_comments	ブール変数。有効にすると、デバッグを簡単にするために Hibernate が SQL 内にコメントを生成します。デフォルト値は **`false`** です。
hibernate.id.new_generator_mappings	ブール値。@GeneratedValue を使用する場合に関係するプロパティーです。新しい IdentifierGenerator 実装が javax.persistence.GenerationType.AUTO、javax.persistence.GenerationType.TABLE、または javax.persistence.GenerationType.SEQUENCE に対して使用されるかどうかを示します。デフォルト値は **`false`** です。

プロパティー名	説明
hibernate.ejb.naming_strategy	Hibernate EntityManager を使用している場合は、org.hibernate.cfg.NamingStrategy 実装を選択します。Hibernate 5.0 では **`hibernate.ejb.naming_strategy`** はサポートされなくなりました。これが使用された場合は、分割された ImplicitNamingStrategy と PhysicalNamingStrategy に置き換わり、サポートが停止され、削除されたことを示す廃止メッセージがログに記録されます。 アプリケーションが EntityManager を使用しない場合は、Hibernate Reference Documentation - Naming Strategies の手順に従って NamingStrategy を設定します。 MetadataBuilder を使用し、暗黙的なネーミングストラテジーを適用するネイティブブートストラップの例については、Hibernate 5.0 ドキュメンテーションの http://docs.jboss.org/hibernate/orm/5.0/userguide/html_single/Hibernate_User_Guide.html#bootstrap-native-metadata を参照してください。物理的なネーミングストラテジーは **`MetadataBuilder.applyPhysicalNamingStrategy()`** を使用して適用できます。**`org.hibernate.boot.MetadataBuilder`** の詳細については、https://docs.jboss.org/hibernate/orm/5.0/javadocs/ を参照してください。
hibernate.implicit_naming_strategy	使用する **`org.hibernate.boot.model.naming.ImplicitNamingStrategy`** クラスを指定します。また、**`hibernate.implicit_naming_strategy`** を使用して ImplicitNamingStrategy を実装するカスタムクラスを設定することもできます。この設定には以下の短い名前が定義されています。 • **`default`** - **`ImplicitNamingStrategyJpaCompliantImpl`** • **`jpa`** - **`ImplicitNamingStrategyJpaCompliantImpl`** • **`legacy-jpa`** - **`ImplicitNamingStrategyLegacyJpaImpl`** • **`legacy-hbm`** - **`ImplicitNamingStrategyLegacyHbmImpl`** • **`component-path`** - **`ImplicitNamingStrategyComponentPathImpl`** デフォルト設定は、**`default`** の短い名前の **`ImplicitNamingStrategy`** によって定義されます。デフォルト設定が空白の場合、フォールバックは **`ImplicitNamingStrategyJpaCompliantImpl`** を使用します。

プロパティー名	説明
hibernate.physical_naming_strategy	データベースオブジェクト名に物理的なネーミングルールを適用するプラグ可能なストラテジーコントラクト。使用する PhysicalNamingStrategy クラスを指定します。デフォルトでは **`PhysicalNamingStrategyStandardImpl`** が使用されます。**`hibernate.physical_naming_strategy`** を使用して、PhysicalNamingStrategy を実装するカスタムクラスを設定することもできます。

重要

新しいアプリケーションでは、**`hibernate.id.new_generator_mappings`** のデフォルト値を **`true`** のままにする必要があります。Hibernate 3.3.x を使用した既存のアプリケーションが継続してシーケンスオブジェクトやテーブルベースのジェネレーターを使用し、後方互換性を維持する場合は、デフォルト値を **`false`** に変更する必要がある場合があります。

12.5.2. Hibernate JDBC と接続プロパティー

表12.2 プロパティー

プロパティー名	説明
hibernate.jdbc.fetch_size	JDBC のフェッチサイズを判断するゼロでない値です (**`Statement.setFetchSize()`** を呼び出します)。
hibernate.jdbc.batch_size	Hibernate による JDBC2 バッチ更新の使用を有効にするゼロでない値です。推奨値は、**`5`**～**`30`** です。
hibernate.jdbc.batch_versioned_data	ブール変数。JDBC ドライバーが **`executeBatch()`** から正しい行数を返す場合は、このプロパティーを **`true`** に設定します。Hibernate は自動的にバージョン化されたデータにバッチ処理された DML を使用します。デフォルト値は **`false`** です。
hibernate.jdbc.factory_class	カスタム org.hibernate.jdbc.Batcher を選択します。ほとんどのアプリケーションにはこの設定プロパティーは必要ありません。
hibernate.jdbc.use_scrollable_resultset	ブール変数。Hibernate による JDBC2 のスクロール可能な結果セットの使用を有効にします。このプロパティーはユーザーが提供した JDBC 接続を使用する場合にのみ必要です。その他の場合、Hibernate は接続メタデータを使用します。

プロパティー名	説明
hibernate.jdbc.use_streams_for_binary	ブール変数。システムレベルのプロパティーです。**`binary`** または **`serializable`** 型を JDBC へ読み書きしたり、JDBC から読み書きしたりする場合にストリームを使用します。
hibernate.jdbc.use_get_generated_keys	ブール変数。JDBC3 **`PreparedStatement.getGeneratedKeys()`** を使用して、挿入後にネイティブで生成された鍵を取得できるようにします。JDBC3+ ドライバーと JRE1.4+ が必要です。JDBC ドライバーに Hibernate 識別子ジェネレーターの問題がある場合は false に設定します。デフォルトでは、接続メタデータを使用してドライバーの機能を判断しようとします。
hibernate.connection.provider_class	JDBC 接続を Hibernate に提供するカスタム org.hibernate.connection.ConnectionProvider のクラス名です。
hibernate.connection.isolation	JDBC トランザクションの分離レベルを設定します。java.sql.Connection で意味のある値をチェックしますが、ほとんどのデータベースはすべての分離レベルをサポートするとは限らず、一部のデータベースは標準的でない分離を追加的に定義します。標準的な値は **`1, 2, 4, 8`** です。
hibernate.connection.autocommit	ブール変数。このプロパティーの使用は推奨されません。JDBC でプールされた接続に対して自動コミットを有効にします。
hibernate.connection.release_mode	Hibernate が JDBC 接続を解放するタイミングを指定します。デフォルトでは、セッションが明示的に閉じられるか切断されるまで JDBC 接続が保持されます。デフォルト値である auto では、JTA および CMT トランザクションストラテジーに対して **`after_statement`** が選択され、JDBC トランザクションストラテジーに対して **`after_transaction`** が選択されます。 利用可能な値は auto (デフォルト)、on_close、**`after_transaction`**、**`after_statement`** です。 この設定により、**`SessionFactory.openSession`** から返されたセッションのみが影響を受けます。**`SessionFactory.getCurrentSession`** から取得されたセッションの場合は、使用のために設定された **`CurrentSessionContext`** 実装がこれらのセッションの接続リリースモードを制御します。

プロパティー名	説明
hibernate.connection.**<propertyName>**	JDBC プロパティー **<propertyName>** を **`DriverManager.getConnection()`** に渡します。
hibernate.jndi.**<propertyName>**	プロパティー **<propertyName>** を JNDI **`InitialContextFactory`** に渡します。

12.5.3. Hibernate キャッシュプロパティー

表12.3 プロパティー

プロパティー名	説明
`hibernate.cache.region.factory_class`	カスタム **`CacheProvider`** のクラス名。
`hibernate.cache.use_minimal_puts`	ブール変数です。2 次キャッシュの操作を最適化し、読み取りを増やして書き込みを最小限にします。これはクラスター化されたキャッシュで最も便利な設定であり、Hibernate 3 ではクラスター化されたキャッシュの実装に対してデフォルトで有効になっています。
`hibernate.cache.use_query_cache`	ブール変数です。クエリーキャッシュを有効にします。各クエリーをキャッシュ可能に設定する必要があります。
`hibernate.cache.use_second_level_cache`	ブール変数です。**<cache>** マッピングを指定するクラスに対してデフォルトで有効になっている 2 次キャッシュを完全に無効にするため使用されます。
`hibernate.cache.query_cache_factory`	カスタム **`QueryCache`** インターフェースのクラス名です。デフォルト値は組み込みの **`StandardQueryCache`** です。
`hibernate.cache.region_prefix`	2 次キャッシュのリージョン名に使用する接頭辞です。
`hibernate.cache.use_structured_entries`	ブール変数です。人間が解読可能な形式でデータを 2 次キャッシュに保存するよう Hibernate を設定します。
`hibernate.cache.default_cache_concurrency_strategy`	@Cacheable または @Cache が使用される場合に使用するデフォルトの org.hibernate.annotations.CacheConcurrencyStrategy の名前を付与するため使用される設定です。このデフォルト値を上書するには、**`@Cache(strategy="..")`** を使用します。

12.5.4. Hibernate トランザクションプロパティー

表12.4 プロパティー

プロパティー名	説明
hibernate.transaction.factory_class	Hibernate **Transaction** API と使用する **TransactionFactory** のクラス名です。デフォルト値は **JDBCTransactionFactory** です。
jta.UserTransaction	アプリケーションサーバーから JTA **UserTransaction** を取得するために **JTATransactionFactory** により使用される JNDI 名。
hibernate.transaction.manager_lookup_class	**TransactionManagerLookup** のクラス名。JVM レベルのキャッシングが有効になっている場合や、JTA 環境の hilo ジェネレーターを使用する場合に必要です。
hibernate.transaction.flush_before_completion	ブール変数。有効な場合、トランザクションの完了前フェーズの間にセッションが自動的にフラッシュされます。ビルトインおよび自動セッションコンテキスト管理が推奨されます。
hibernate.transaction.auto_close_session	ブール変数。有効な場合、トランザクションの完了後フェーズの間にセッションが自動的に閉じられます。ビルトインおよび自動セッションコンテキスト管理が推奨されます。

12.5.5. その他の Hibernate プロパティー

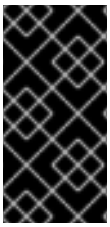
表12.5 プロパティー

プロパティー名	説明
hibernate.current_session_context_class	「現在」の **Session** のスコープに対するカスタムストラテジーを提供します。値には **jta**、**thread**、**managed**、**custom.Class** があります。
hibernate.query.factory_class	**org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory** または **org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory** の HQL パーサー実装を選択します。

プロパティー名	説明
hibernate.query.substitutions	Hibernate クエリーのトークンと SQL トークンとのマッピングに使用します (トークンは関数名またはリテラル名である場合があります)。たとえば、**hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC** のようになります。
hibernate.hbm2ddl.auto	**SessionFactory** が作成されると、スキーマ DDL を自動的に検証し、データベースにエクスポートします。**create-drop** を使用すると、**SessionFactory** が明示的に閉じられたときにデータベーススキーマが破棄されます。プロパティー値のオプションは、**validate**、**update**、**create**、**create-drop** です。
hibernate.hbm2ddl.import_files	SessionFactory の作成中に実行される SQL DML ステートメントが含まれる任意のファイルの名前 (コンマ区切り)。テストやデモに便利です。たとえば、INSERT ステートメントを追加すると、デプロイ時に最小限のデータセットがデータベースに入力されます。値の例としては、**/humans.sql,/dogs.sql** のようになります。 特定ファイルのステートメントは後続ファイルのステートメントの前に実行されるため、ファイルの順番に注意する必要があります。これらのステートメントはスキーマが作成された場合のみ実行されます (**hibernate.hbm2ddl.auto** が **create** または **create-drop** に設定された場合など)。
hibernate.hbm2ddl.import_files_sql_extractor	カスタム ImportSqlCommandExtractor のクラス名。デフォルト値は組み込みの SingleLineSqlCommandExtractor です。各インポートファイルから単一の SQL ステートメントを抽出する専用のパーサーを実装する時に便利です。Hibernate は、複数行にまたがる命令/コメントおよび引用符で囲まれた文字列をサポートする MultipleLinesSqlCommandExtractor も提供します (各ステートメントの最後にセミコロンが必要です)。
hibernate.bytecode.use_reflection_optimizer	ブール値。**hibernate.cfg.xml** ファイルで設定できないシステムレベルのプロパティーです。ランタイムリフレクションの代わりにバイトコード操作の使用を有効にします。リフレクションは、トラブルシューティングを行うときに役に立つ場合があります。オプティマイザーが無効の場合でも Hibernate には cglib または javassist が常に必要です。

プロパティー名	説明
hibernate.bytecode.provider	javassist または cglib をバイト操作エンジンとして使用することができます。デフォルトでは **javassist** が使用されます。プロパティー値は **javassist** または **cglib** のいずれかです。

12.5.6. Hibernate SQL 方言



重要

hibernate.dialect プロパティーをアプリケーションデータベースの適切な **org.hibernate.dialect.Dialect** サブクラスに設定する必要があります。方言が指定されている場合、Hibernate は他のプロパティーの一部に実用的なデフォルトを使用します。そのため、これらのプロパティーを手作業で指定する必要はありません。

表12.6 SQL 方言 (hibernate.dialect)

RDBMS	方言
DB2	**org.hibernate.dialect.DB2Dialect**
DB2 AS/400	**org.hibernate.dialect.DB2400Dialect**
DB2 OS390	**org.hibernate.dialect.DB2390Dialect**
Firebird	**org.hibernate.dialect.FirebirdDialect**
FrontBase	**org.hibernate.dialect.FrontbaseDialect**
H2 Database	**org.hibernate.dialect.H2Dialect**
HypersonicSQL	**org.hibernate.dialect.HSQLDialect**
Informix	**org.hibernate.dialect.InformixDialect**
Ingres	**org.hibernate.dialect.IngresDialect**
Interbase	**org.hibernate.dialect.InterbaseDialect**
MariaDB 10	**org.hibernate.dialect.MySQL57InnoDBDialect**
Mckoi SQL	**org.hibernate.dialect.MckoiDialect**
Microsoft SQL Server 2000	**org.hibernate.dialect.SQLServerDialect**
Microsoft SQL Server 2005	**org.hibernate.dialect.SQLServer2005Dialect**

RDBMS	方言
Microsoft SQL Server 2008	**org.hibernate.dialect.SQLServer2008Dialect**
Microsoft SQL Server 2012	**org.hibernate.dialect.SQLServer2012Dialect**
Microsoft SQL Server 2014	**org.hibernate.dialect.SQLServer2012Dialect**
MySQL5	**org.hibernate.dialect.MySQL5Dialect**
MySQL5.7	**org.hibernate.dialect.MySQL57InnoDBDialect**
InnoDB を用いる MYSQL5	**org.hibernate.dialect.MySQL5InnoDBDialect**
MyISAM を用いる MySQL	**org.hibernate.dialect.MySQLMyISAMDialect**
Oracle (全バージョン)	**org.hibernate.dialect.OracleDialect**
Oracle 9i	**org.hibernate.dialect.Oracle9iDialect**
Oracle 10g	**org.hibernate.dialect.Oracle10gDialect**
Oracle 11g	**org.hibernate.dialect.Oracle10gDialect**
Oracle 12c	**org.hibernate.dialect.Oracle12cDialect**
Pointbase	**org.hibernate.dialect.PointbaseDialect**
PostgreSQL	**org.hibernate.dialect.PostgreSQLDialect**
PostgreSQL 9.2	**org.hibernate.dialect.PostgreSQL9Dialect**
PostgreSQL 9.3	**org.hibernate.dialect.PostgreSQL9Dialect**
PostgreSQL 9.4	**org.hibernate.dialect.PostgreSQL94Dialect**
Postgres Plus Advanced Server	**org.hibernate.dialect.PostgresPlusDialect**
Progress	**org.hibernate.dialect.ProgressDialect**
SAP DB	**org.hibernate.dialect.SAPDBDialect**
Sybase	**org.hibernate.dialect.SybaseASE15Dialect**
Sybase 15.7	**org.hibernate.dialect.SybaseASE157Dialect**
Sybase Anywhere	**org.hibernate.dialect.SybaseAnywhereDialect**

12.6. 2 次キャッシュ

12.6.1. 2 次キャッシュ

2 次キャッシュとは、アプリケーションセッション以外で永続的に情報を保持するローカルのデータストアのことです。このキャッシュは永続プロバイダーにより管理されており、アプリケーションとデータを分けることでランタイム効率の改善をはかることができます。

JBoss EAP では、以下の目的のためにキャッシュがサポートされます。

- Web セッションのクラスタリング
- ステートフルセッション Bean のクラスタリング
- SSO クラスタリング
- Hibernate 2 次キャッシュ

各キャッシュコンテナーは 「repl」 と 「dist」 キャッシュを定義します。これらのキャッシュは、ユーザーアプリケーションで直接使用しないでください。

12.6.2. Hibernate 用 2 次キャッシュの設定

Hibernate 向けの 2 次レベルキャッシュとして動作する Infinispan の設定は、以下の 2 つの方法で行なえます。

- Hibernate ネイティブアプリケーション経由 (**`hibernate.cfg.xml`** を使用)
- JPA アプリケーション経由 (**`persistence.xml`** を使用)

Hibernate ネイティブアプリケーションを使用した Hibernate 用 2 次キャッシュの設定

1. デプロイメントのクラスパスに **`hibernate.cfg.xml`** を作成します。
2. 以下の XML を **`hibernate.cfg.xml`** に追加します。XML は **`<session-factory>`** タグ内に存在する必要があります。

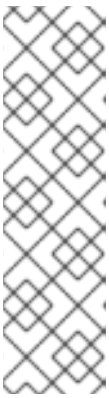
```
<property
name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property
name="hibernate.cache.region.factory_class">org.jboss.as.jpa.hiberna
te5.infinispan.InfinispanRegionFactory</property>
```

JPA ネイティブアプリケーションを使用した Hibernate 用 2 次キャッシュの設定

1. Red Hat JBoss Developer Studio で Hibernate 設定ファイルを作成する方法については、単純な JPA アプリケーションの作成を参照してください。
2. 以下の内容を **`persistence.xml`** ファイルに追加します。

```
<persistence-unit name="...">
  (...) <!-- other configuration -->
  <shared-cache-mode>$SHARED_CACHE_MODE</shared-cache-mode>
  <properties>
    <property name="hibernate.cache.use_second_level_cache"
```

```
value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
  </properties>
</persistence-unit>
```



注記

$SHARED_CACHE_MODE の値には以下のものがあります。

- ALL - すべてのエンティティーがキャッシュ可能である見なされます。
- ENABLE_SELECTIVE - キャッシュ可能とマークされたエンティティーのみがキャッシュ可能であると見なされます。
- DISABLE_SELECTIVE - キャッシュ不可と明示的に示されたものを除くすべてのエンティティーはキャッシュ可能と見なされます。

12.7. HIBERNATE アノテーション

org.hibernate.annotations パッケージには、標準的な JPA アノテーション以外に Hibernate により提供される一部のアノテーションが含まれます。

表12.7 一般的なアノテーション

アノテーション	説明
Check	クラス、プロパティー、コレクションのいずれかのレベルで定義できる任意の SQL チェック制約です。
Immutable	エンティティーまたはコレクションを不変としてマーク付けします。アノテーションがない場合、要素は可変となります。 不変のエンティティーはアプリケーションによって更新されないことがあります。不変エンティティーへの更新は無視されますが、例外は発生しません。 **@Immutable** をコレクションに付けるとコレクションは不変になるため、コレクションからの追加や削除およびコレクションへの追加や削除は許可されません。この結果、**HibernateException** が発生します。

表12.8 キャッシュエンティティー

アノテーション	説明
Cache	ルートエンティティーまたはコレクションにキャッシングストラテジーを追加します。

表12.9 コレクション関連のアノテーション

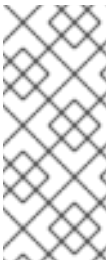
アノテーション	説明
MapKeyType	永続マップのキータイプを定義します。
ManyToAny	異なるエンティティータイプを参照する **ToMany** アソシエーションを定義します。 メタデータ識別子カラムを介してエンティティータイプの照合が行われます。このようなマッピングはできるだけ行わないでください。
OrderBy	SQL の順序付け (HQL の順序付けではない) を使用してコレクションの順序を付けます。
OnDelete	コレクションやアレイ、結合されたサブクラスの削除に使用されるストラテジーです。**OnDelete** の 2 次テーブルはサポートされていません。
Persister	カスタムパーシスターを指定します。
Sort	コレクションのソート (Java レベルのソート)。
Where	要素エンティティーまたはコレクションのターゲットエンティティーへ追加する where 節。この節は SQL で書かれます。
WhereJoinTable	コレクション結合テーブルへ追加する where 節。この節は SQL で書かれます。

表12.10 CRUD 操作用のカスタム SQL

アノテーション	説明
Loader	Hibernate のデフォルトである **FIND** メソッドを上書きします。
SQLDelete	Hibernate のデフォルトである **DELETE** メソッドを上書きします。
SQLDeleteAll	Hibernate のデフォルトである **DELETE ALL** メソッドを上書きします。
SQLInsert	Hibernate のデフォルトである **INSERT INTO** メソッドを上書きします。
SQLUpdate	Hibernate のデフォルトである **UPDATE** メソッドを上書きします。
Subselect	不変の読み取り専用エンティティーを指定のサブセレクト表現へマッピングします。

アノテーション	説明
Synchronize	自動フラッシュが適切に行われ、派生したエンティティーに対するクエリーが古いデータを返さないようにします。ほとんどの場合、**Subselect** と共に使用されます。

表12.11 Entity

アノテーション	説明
Cascade	アソシエーションにカスケードストラテジーを適用します。
Entity	標準的な **@Entity** で定義されたもの以外に必要になることがあるメタデータを追加します。 • **mutable**: このエンティティーが変更可能であるかどうか • **dynamicInsert**: 挿入に動的 SQL を許可する • **dynamicUpdate**: 更新に動的 SQL を許可する • **selectBeforeUpdate**: オブジェクトが実際に変更されない限り、Hibernate が SQL UPDATE を実行しないことを指定します。 • **polymorphism**: エンティティーのポリモーフィズムが PolymorphismType.IMPLICIT (default) であるか、または PolymorphismType.EXPLICIT であるか • **optimisticLock**: オプティミスティックロッキングストラテジー (OptimisticLockType.VERSION、OptimisticLockType.NONE、OptimisticLockType.DIRTY、または OptimisticLockType.ALL)  **注記** アノテーション「Entity」は非推奨になり、今後のリリースで削除される予定です。個々の属性または値はアノテーションにする必要があります。
Polymorphism	Hibernate がエンティティーの階層に適用する多様性タイプを定義するために使用されます。

アノテーション	説明
Proxy	特定クラスのレイジーおよびプロキシ設定。
Table	1 次または 2 次テーブルへの補足情報。
Tables	Table の複数アノテーション。
Target	明示的なターゲットを定義し、リフレクションやジェネリクスで解決しないようにします。
Tuplizer	1 つのエンティティーまたはコンポーネントに対して単一の tuplizer を定義します。
Tuplizers	1 つのエンティティーまたはコンポーネントに対して tuplizer のセットを定義します。

表12.12 Fetching

アノテーション	説明
BatchSize	SQL ローディングのバッチサイズ。
FetchProfile	フェッチングストラテジープロファイルを定義します。
FetchProfiles	**@FetchProfile** の複数アノテーション。

表12.13 Filters

アノテーション	説明
Filter	エンティティーまたはコレクションのターゲットエンティティーにフィルターを追加します。
FilterDef	フィルター定義。
FilterDefs	フィルター定義のアレイ。
FilterJoinTable	結合テーブルのコレクションへフィルターを追加します。
FilterJoinTables	複数の **@FilterJoinTable** をコレクションへ追加します。
Filters	複数の **@Filter** を追加します。

アノテーション	説明
ParamDef	パラメーターの定義。

表12.14 主キー

アノテーション	説明
Generated	このアノテーション付けされたプロパティーはデータベースによって生成されます。
GenericGenerator	Hibernate ジェネレーターをデタイプ (detyped) で記述するジェネレーターアノテーションです。
GenericGenerators	汎用ジェネレーター定義のアレイ。
NaturalId	プロパティーがエンティティーのナチュラル ID の一部であることを指定します。
Parameter	キーと値のパターン。
RowId	Hibernate の **ROWID** マッピング機能をサポートします。

表12.15 継承

アノテーション	説明
DiscriminatorFormula	ルートエントリーに置かれる識別子の公式です。
DiscriminatorOptions	Hibernate 固有の識別子プロパティーを表現する任意のアノテーションです。
MetaValue	該当する識別子の値を対応するエンティティータイプにマッピングします。

表12.16 JP-QL/HQL クエリーのマッピング

アノテーション	説明
NamedNativeQueries	Hibernate NamedNativeQuery オブジェクトを保持するよう **NamedNativeQueries** を拡張します。
NamedNativeQuery	**NamedNativeQuery** を Hibernate の機能で拡張します。

アノテーション	説明
NamedQueries	**NamedQuery** オブジェクトを保持するよう **NamedQueries** を拡張します。
NamedQuery	Hibernate の機能で **NamedQuery** を拡張します。

表12.17 単純なプロパティーのマッピング

アノテーション	説明
AccessType	プロパティーのアクセスタイプ。
Columns	カラムのアレイをサポートします。コンポーネントユーザータイプのマッピングに便利です。
ColumnTransformer	カラムからの値の読み取りやカラムへの値の書き込みに使用されるカスタム SQL 表現です。直接的なオブジェクトのロードや保存、クエリーに使用されます。write 表現には必ず値に対して 1 つの 「?」 プレースホルダーが含まれなければなりません。
ColumnTransformers	**@ColumnTransformer** の複数アノテーションです。複数のカラムがこの挙動を使用する場合に便利です。

表12.18 プロパティー

アノテーション	説明
Formula	ほとんどの場所で **@Column** の代替として使用されます。公式は有効な SQL フラグメントである必要があります。
Index	データベースのインデックスを定義します。
JoinFormula	ほとんどの場所で **@JoinColumn** の代替として使用されます。公式は有効な SQL フラグメントである必要があります。
Parent	所有者 (通常は所有するエンティティー) へのポインターとしてプロパティーを参照します。
Type	Hibernate のタイプ。
TypeDef	Hibernate タイプの定義。
TypeDefs	Hibernate タイプ定義のアレイ。

表12.19 単一アソシエーション関連のアノテーション

アノテーション	説明
Any	複数のエンティティータイプを参照する **ToOne** を定義します。対応するエンティティータイプの照合は、メタデータ識別子カラムを介して行われます。このようなマッピングは最低限行う必要があります。
AnyMetaDef	**@Any** および **@ManyToAny** メタデータを定義します。
AnyMetaDefs	メタデータの **@Any** および **@ManyToAny** セットを定義します。エンティティーまたはパッケージレベルで定義できます。
Fetch	特定のアソシエーションに使用されるフェッチングストラテジーを定義します。
LazyCollection	コレクションのレイジー状態を定義します。
LazyToOne	ToOne アソシエーション (つまり、**OneToOne** または **ManyToOne**) のレイジーステータスを定義します。
NotFound	アソシエーション上で要素が見つからなかった時に実行するアクションです。

表12.20 オプティミスティックロッキング

アノテーション	説明
OptimisticLock	アノテーション付けされたプロパティーの変更によってエンティティーのバージョン番号が増加するかどうか。アノテーション付けされていない場合、プロパティーは楽観的ロックストラテジー (デフォルト) に関与します。
OptimisticLocking	エンティティーに適用される楽観的ロックのスタイルを定義するために使用されます。階層ではルートエンティティーのみに有効です。
Source	バージョンおよびタイムスタンプバージョンプロパティーと併用するのに最適なアノテーションです。アノテーション値はタイムスタンプが生成される場所を決定します。

12.8. HIBERNATE クエリー言語

12.8.1. Hibernate クエリー言語

JPQL の概要
Java Persistence Query Language (JPQL) は、Java Persistence API (JPA) 仕様の一部として定義されたプラットフォーム非依存のオブジェクト指向クエリー言語です。JPQL は、リレーショナルデータベースに格納されたエンティティーに対してクエリーを実行するために使用されます。この言語は SQL から大きな影響を受けており、クエリーの構文は SQL クエリーに類似しますが、データベーステーブルと直接動作するのではなく JPA エンティティーオブジェクトに対して動作します。

HQL の概要
Hibernate Query Language (HQL) は、強力なクエリー言語であり、見た目は SQL に似ています。ただし、SQL と比較して、HQL は完全なオブジェクト指向であり、継承、ポリモーフィズム、アソシエーションなどの概念を理解します。

HQL は JPQL のスーパーセットです。HQL クエリーは有効な JPQL クエリーでないこともありますが、JPQL クエリーは常に有効な HQL クエリーになります。

HQL と JPQL は共にタイプセーフでないクエリー操作を実行します。基準 (criteria) クエリーがタイプセーフなクエリーを提供します。

12.8.2. HQL ステートメントについて

HQL と JPQL では、**SELECT**、**UPDATE**、および **DELETE** ステートメントを使用できます。HQL では、SQL の **INSERT-SELECT** に似た形式で **INSERT** ステートメントも使用できます。

重要

更新操作または削除操作を一括で実行する場合は、アクティブな永続コンテキストでデータベースとエンティティーとの間に不整合が発生することがあるため、注意が必要です。一般的に、一括の更新操作または削除操作は、新しい永続コンテキストのトランザクション内、またはこのような操作により状態が影響を受ける可能性があるエンティティーを取得する前またはアクセスする前にのみ実行します。

表12.21 HQL ステートメント

ステートメント	説明
SELECT	HQL での SELECT ステートメントの BNF は以下のとおりです。 `select_statement :: = [select_clause] from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]`
UDPATE	HQL の UPDATE ステートメントの BNF は JPQL と同じです。
DELETE	HQL の DELETE ステートメントの BNF は JPQL と同じです。

12.8.3. INSERT ステートメント

HQL は **INSERT** ステートメントを定義する機能を追加します。これに相当するステートメントは JPQL にはありません。HQL の **INSERT** ステートメントの BNF は次のとおりです。

```
insert_statement ::= insert_clause select_statement

insert_clause ::= INSERT INTO entity_name (attribute_list)

attribute_list ::= state_field[, state_field ]*
```

attribute_list は、SQL **INSERT** ステートメントの **column specification** と似ています。マップされた継承に関係するエンティティーでは、名前付きエンティティー上で直接定義された属性のみを **attribute_list** で使用することが可能です。スーパークラスプロパティーは許可されず、サブクラスプロパティーは意味がありません。よって、 **INSERT** ステートメントは本質的に非多形となります。

警告

The **select_statement** can be any valid HQL select query, with the caveat that the return types must match the types expected by the insert. Currently, this is checked during query compilation rather than allowing the check to relegate to the database. This can cause problems with Hibernate Types that are **equivalent** as opposed to **equal**. For example, this might cause mismatch issues between an attribute mapped as an **org.hibernate.type.DateType** and an attribute defined as a **org.hibernate.type.TimestampType**, even though the database might not make a distinction or might be able to handle the conversion.

insert ステートメントは **id** 属性に対して 2 つのオプションを提供します。1 つ目は、**id** 属性を **attribute_list** に明示的に指定するオプションです。この場合、値は対応する select 式から取得されます。2 つ目は **attribute_list** に指定しないオプションであり、この場合生成された値が使用されます。2 つ目のオプションは、「データベース内」で動作する **id** ジェネレーターを使用する場合のみ選択可能です。このオプションを「インメモリ」タイプのジェネレーターで使用しようとすると、構文解析中に例外が生じます。

insert ステートメントは楽観的ロックの属性に対しても 2 つのオプションを提供します。1 つ目は **attribute_list** に属性を指定するオプションです。この場合、値は対応する select 式から取得されます。2 つ目は **attribute_list** に指定しないオプションです。この場合、対応する org.hibernate.type.VersionType によって定義される **seed value** が使用されます。

例: INSERT クエリーステートメント

```
String hqlInsert = "insert into DelinquentAccount (id, name) select c.id,
c.name from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert ).executeUpdate();
```

12.8.4. FROM 節

FROM 節の役割は、他のクエリーが使用できるオブジェクトモデルタイプの範囲を定義することです。また、他のクエリーが使用できる「ID 変数」もすべて定義します。

12.8.5. WITH 節

HQL は **WITH** 節を定義し、結合条件を限定します。これは HQL に固有の機能で、JPQL はこの機能を定義しません。

例: With 句

```
select distinct c
from Customer c
    left join c.orders o
        with o.value > 5000.00
```

生成された SQL では、**with clause** の条件が生成された SQL の **on clause** の一部となりますが、本項の他のクエリーでは HQL/JPQL の条件が生成された SQL の **where clause** の一部となることが重要な違いです。この例に特有の違いは重要ではないでしょう。さらに複雑なクエリーでは、**with clause** が必要になることがあります。

明示的な結合は、アソシエーションまたはコンポーネント/埋め込み属性を参照することがあります。コンポーネント/埋め込み属性では、結合は論理的であり、物理 (SQL) 結合に関連しません。

12.8.6. HQL の順序付け

クエリーの結果を順序付けすることも可能です。**ORDER BY** 節を使用して、結果を順序付けするために使用される選択値を指定します。order-by 節の一部として有効な式タイプには以下が含まれます。

- ステートフィールド
- コンポーネント/埋め込み可能属性
- 算術演算や関数などのスカラー式
- 前述の式タイプのいずれかに対する select 節に宣言された ID 変数

HQL は、order-by 節で参照されたすべての値が select 節で名付けされることを強制しませんが、JPQL では必要となります。データベースの移植性を要求するアプリケーションは、select 節で参照されない order-by 節の参照値をサポートしないデータベースがあることを認識する必要があります。

order-by の各式は、**ASC** (昇順) または **DESC** (降順) で希望の順序を示すよう修飾することができます。

例: Order-by

```
// legal because p.name is implicitly part of p
select p
from Person p
order by p.name

select c.id, sum( o.total ) as t
from Order o
    inner join o.customer c
group by c.id
order by t
```

12.8.7. 一括更新、一括送信、および一括削除

Hibernate では、Data Manipulation Language (DML) を使用して、マップ済みデータベースのデータを直接、一括挿入、一括更新、および一括削除できます (Hibernate Query Language を使用)。

警告

DML を使用すると、オブジェクト/リレーショナルマッピングに違反し、オブジェクトの状態に影響が出ることがあります。オブジェクトの状態はメモリーでは変わりません。DML を使用することにより、基礎となるデータベースで実行された操作に応じて、メモリー内オブジェクトの状態は影響を受けません。DML を使用する場合、メモリー内データは注意を払って使用する必要があります。

UPDATE ステートメントと DELETE ステートメントの擬似構文:

`( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?`.

注記

`FROM` キーワードと **`WHERE Clause`** はオプションです。

UPDATE ステートメントまたは DELETE ステートメントの実行結果は、実際に影響 (更新または削除) を受けた行の数です。

例: 一括更新ステートメント

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Company set name = :newName where name =
:oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

例: 一括削除ステートメント

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Company where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Query.executeUpdate() メソッドにより返された **int** 値は、操作で影響を受けたデータベース内のエンティティー数を示します。

内部的に、データベースは複数の SQL ステートメントを使用して DML 更新または削除の要求に対する操作を実行することがあります。多くの場合、これは、更新または削除する必要があるテーブルと結合テーブル間に存在する関係のためです。

たとえば、上記の例のように削除ステートメントを発行すると、**oldName** で指定された会社用の **Company** テーブルだけでなく、結合テーブルに対しても削除が実行されることがあります。したがって、Employee テーブルとの関係が BiDirectional ManyToMany である Company テーブルで、以前の例の正常な実行結果として、対応する結合テーブル **Company_Employee** から複数の行が失われます。

上記の **int deletedEntries** 値には、この操作により影響を受けたすべての行 (結合テーブルの行を含む) の数が含まれます。

INSERT ステートメントの擬似構文は **INSERT INTO EntityName properties_list select_statement** です。

注記

INSERT INTO … SELECT … という形式のみサポートされ、INSERT INTO … VALUES … という形式はサポートされません。

例: 一括挿入ステートメント

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Account (id, name) select c.id, c.name
from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

SELECT ステートメントを介して **id** 属性の値を提供しない場合は、基礎となるデータベースが自動生成されたキーをサポートする限り、ユーザーに対して ID が生成されます。この一括挿入操作の戻り値は、データベースで実際に作成されたエントリーの数です。

12.8.8. コレクションメンバーの参照

コレクション値 (collection-valued) アソシエーションへの参照は、実際はコレクションの**値**を参照します。

例: コレクションの参照

```
select c
from Customer c
    join c.orders o
    join o.lineItems l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'
```

```
// alternate syntax
select c
from Customer c,
    in(c.orders) o,
    in(o.lineItems) l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'
```

この例では、ID 変数 **o** が、Customer#orders アソシエーションの要素の型であるオブジェクトモデル型 Order を実際に参照します。

更にこの例には、**IN** 構文を使用してコレクションアソシエーション結合を指定する代替の構文があります。構文は両方同等です。アプリケーションが使用する構文は任意に選択できます。

12.8.9. 修飾パス式

コレクション値 (collection-valued) のアソシエーションは、実際にはそのコレクションの**値**を参照すると前項で説明しました。コレクションのタイプを基に、明示的な修飾式のセットも使用可能です。

表12.22 修飾パス式

表現	説明
VALUE	コレクション値を参照します。修飾子を指定しないことと同じです。目的を明示的に表す場合に便利です。コレクション値 (collection-valued) の参照のすべてのタイプに対して有効です。
INDEX	HQL ルールに基づき、マップキーまたはリストの場所 (OrderColumn の値) を参照するよう javax.persistence.OrderColumn アノテーションを指定するマップとリストに対して有効です。ただし、JPQL ではリストでの使用に対して確保され、マップに対して **KEY** が追加されます。JPA プロバイダーの移植性に関係するアプリケーションは、この違いを認識する必要があります。
KEY	マップに対してのみ有効です。マップのキーを参照します。キー自体がエンティティーである場合は、さらにナビゲートすることが可能です。
ENTRY	マップに対してのみ有効です。マップの論理 java.util.Map.Entry タプル (キーと値の組み合わせ) を参照します。**ENTRY** は終端パスとしてのみ有効であり、select 句のみで有効になります。

例: 修飾されたコレクションの参照

```
// Product.images is a Map<String,String> : key = a name, value = file
path

// select all the image file paths (the map value) for Product#123
```

```
select i
from Product p
    join p.images i
where p.id = 123

// same as above
select value(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names (the map key) for Product#123
select key(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names and file paths (the 'Map.Entry') for
Product#123
select entry(i)
from Product p
    join p.images i
where p.id = 123

// total the value of the initial line items for all orders for a customer
select sum( li.amount )
from Customer c
        join c.orders o
        join o.lineItems li
where c.id = 123
  and index(li) = 1
```

12.8.10. スカラー関数

HQL は、使用される基盤のデータに関係なく使用できる一部の標準的な機能を定義します。また、HQL は方言やアプリケーションによって定義された追加の機能も理解することができます。

12.8.11. HQL の標準化された関数について

使用される基盤のデータベースに関係なく HQL で使用できる関数は次のとおりです。

表12.23 HQL の標準化された関数

機能	説明
BIT_LENGTH	バイナリーデータの長さを返します。
CAST	SQL キャストを実行します。キャストターゲットは使用する Hibernate マッピングタイプを命名する必要があります。

機能	説明
EXTRACT	datetime 値で SQL の抽出を実行します。抽出により、datetime 値の一部が抽出されます (年など)。以下の省略形を参照してください。
SECOND	秒を抽出する抽出の省略形。
MINUTE	分を抽出する抽出の省略形。
HOUR	時間を抽出する抽出の省略形。
DAY	日を抽出する抽出の省略形。
MONTH	月を抽出する抽出の省略形。
YEAR	年を抽出する抽出の省略形。
STR	値を文字データとしてキャストする省略形。

アプリケーション開発者は独自の関数セットを提供することもできます。通常、カスタム SQL 関数か SQL スニペットのエイリアスで表します。このような関数は、**org.hibernate.cfg.Configuration** の addSqlFunction メソッドを使用して宣言します。

12.8.12. 連結演算

HQL は、連結 (**CONCAT**) 関数をサポートするだけでなく、連結演算子も定義します。連結演算子は JPQL によっては定義されないため、移植可能なアプリケーションでは使用しないでください。連結演算子は SQL の連結演算子である **||** を使用します。

例: 連結操作

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

12.8.13. 動的インスタンス化

select 節でのみ有効な特別な式タイプがありますが、Hibernate では「動的インスタンス化」と呼びます。JPQL はこの機能の一部をサポートし、「コンストラクター式」と呼びます。

例: 動的インスタンス化 - コンストラクター

```
select new Family( mother, mate, offspr )
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

Object[] に対処せずに、クエリーの結果として返されるタイプセーフの Java オブジェクトで値をラッピングします。クラス参照は完全修飾する必要があり、一致するコンストラクターがなければなりません。

ここでは、クラスをマッピングする必要はありません。エンティティーを表す場合、結果となるインスタンスは NEW 状態で返されます (管理されません)。

この部分は JPQL もサポートします。HQL は他の「動的インスタンス化」もサポートします。最初に、スカラーの結果に対して Object[] ではなくリストを返すよう、クエリーで指定できます。

例: 動的インスタンス化 - リスト

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

このクエリーの結果は List<Object[]> ではなく List<List>

また、HQL はマップにおけるスカラーの結果のラッピングもサポートします。

例: 動的インスタンス化 - マップ

```
select new map( mother as mother, offspr as offspr, mate as mate )
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr

select new map( max(c.bodyWeight) as max, min(c.bodyWeight) as min,
count(*) as n )
from Cat cxt
```

このクエリーの結果は List<Object[]> ではなく List<Map<String,Object>> になります。マップのキーは select 式へ提供されたエイリアスによって定義されます。

12.8.14. HQL 述語

述語は **where** 句、**having** 句、および検索 case 式の基盤を形成します。これらは通常は **TRUE** または **FALSE** の真理値に解決される式ですが、一般的に NULL 値が関係するブール値の比較は **UNKNOWN** に解決されます。

HQL 述語

- Null 述語
  NULL の値をチェックします。基本的な属性参照、エンティティー参照、およびパラメーターへ適用できます。HQL では、コンポーネント/埋め込み可能タイプにも適用できます。

  Null チェックの例

  ```
  // select everyone with an associated address
  select p
  from Person p
  where p.address is not null

  // select everyone without an associated address
  ```

```
select p
from Person p
  where p.address is null
```

- LIKE 述語
  文字列値で LIKE 比較を実行します。構文は次のとおりです。

```
like_expression ::=
       string_expression
       [NOT] LIKE pattern_value
       [ESCAPE escape_character]
```

  セマンティックは SQL の LIKE 式に従います。**pattern_value** は、**string_expression** で一致を試みるパターンです。SQL と同様に、**pattern_value** には _ (アンダースコア) と % (パーセント) をワイルドカードとして使用できます。意味も同じであり、_ はあらゆる 1 つの文字と一致し、% はあらゆる数の文字と一致します。

  任意の **escape_character** は、**pattern_value** の _ や % をエスケープするために使用するエスケープ文字を指定するために使用されます。これは _ や % が含まれるパターンを検索する必要がある場合に役立ちます。

  Like 述語の例

```
select p
from Person p
where p.name like '%Schmidt'

select p
from Person p
where p.name not like 'Jingleheimmer%'

// find any with name starting with "sp_"
select sp
from StoredProcedureMetadata sp
where sp.name like 'sp|_%' escape '|'
```

- BETWEEN 述語
  SQL の **BETWEEN** 式と同様です。値が他の 2 つの値の間にあることを評価するために実行します。演算対象はすべて比較可能な型を持つ必要があります。

  Between 述語の例

```
select p
from Customer c
    join c.paymentHistory p
where c.id = 123
  and index(p) between 0 and 9

select c
from Customer c
where c.president.dateOfBirth
        between {d '1945-01-01'}
            and {d '1965-01-01'}
```

```
select o
from Order o
where o.total between 500 and 5000

select p
from Person p
where p.name between 'A' and 'E'
```

- IN 述語
  IN 述語は、値のリストに特定の値があることを確認するチェックを行います。構文は次のとおりです。

```
in_expression ::= single_valued_expression
            [NOT] IN single_valued_list

single_valued_list ::= constructor_expression |
            (subquery) |
            collection_valued_input_parameter

constructor_expression ::= (expression[, expression]*)
```

  single_valued_expression のタイプと **single_valued_list** の各値は一致しなければなりません。JPQL は有効なタイプを文字列、数字、日付、時間、タイムスタンプ、列挙型に限定します。JPQL では、 **single_valued_expression** は下記のみを参照できます。

  - 簡単な属性を表す「ステートフィールド」。アソシエーションとコンポーネント/埋め込み属性を明確に除外します。
  - エンティティータイプの式。
    HQL では、**single_valued_expression** はさらに広範囲の式タイプを参照することが可能です。単一値のアソシエーションは許可されます。コンポーネント/埋め込み属性も許可されますが、この機能は、基礎となるデータベースのタプルまたは「行値コンストラクター構文」へのサポートのレベルに依存します。また、HQL は値タイプを制限しませんが、基礎となるデータベースのベンダーによってはサポートが制限されるタイプがあることをアプリケーション開発者は認識しておいたほうがよいでしょう。これが JPQL の制限の主な原因となります。

    値のリストは複数の異なるソースより取得することが可能です。**constructor_expression** と **collection_valued_input_parameter** では、空の値のリストは許可されず、最低でも 1 つの値が含まれなければなりません。

    In 述語の例

```
select p
from Payment p
where type(p) in (CreditCardPayment, WireTransferPayment)

select c
from Customer c
where c.hqAddress.state in ('TX', 'OK', 'LA', 'NM')

select c
from Customer c
where c.hqAddress.state in ?
```

```
select c
from Customer c
where c.hqAddress.state in (
    select dm.state
    from DeliveryMetadata dm
    where dm.salesTax is not null
)

// Not JPQL compliant!
select c
from Customer c
where c.name in (
    ('John','Doe'),
    ('Jane','Doe')
)

// Not JPQL compliant!
select c
from Customer c
where c.chiefExecutive in (
    select p
    from Person p
    where ...
)
```

12.8.15. 関係比較

比較には比較演算子 (=、>、>=、<、⇐、<>) の 1 つが関与します。また、HQL によって != は <>. と同義の比較演算子として定義されます。オペランドは同じ型でなければなりません。

例: 相対比較

```
// numeric comparison
select c
from Customer c
where c.chiefExecutive.age < 30

// string comparison
select c
from Customer c
where c.name = 'Acme'

// datetime comparison
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

// enum comparison
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

// boolean comparison
select c
from Customer c
```

```
where c.sendEmail = true

// entity type comparison
select p
from Payment p
where type(p) = WireTransferPayment

// entity value comparison
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

比較には、サブクエリー修飾子である **ALL**、 **ANY**、 **SOME** も関与します。**SOME** と **ANY** は同義です。

サブクエリーの結果にあるすべての値に対して比較が true である場合、**ALL** 修飾子は true に解決されます。サブクエリーの結果が空の場合は false に解決されます。

例: ALL サブクエリー比較修飾子

```
// select all players that scored at least 3 points
// in every game.
select p
from Player p
where 3 > all (
   select spg.points
   from StatsPerGame spg
   where spg.player = p
)
```

サブクエリーの結果にある値の一部 (最低でも 1 つ) に対して比較が true の場合、**ANY** または **SOME** 修飾子は true に解決されます。サブクエリーの結果が空である場合、false に解決されます。

12.9. HIBERNATE サービス

12.9.1. Hibernate サービス

サービスは、さまざまな機能タイプのプラグ可能な実装を Hibernate に提供するクラスです。サービスは特定のサービスコントラクトインターフェースの実装です。インターフェースはサービスロールとして知られ、実装クラスはサービス実装として知られています。通常、ユーザーはすべての標準的なサービスロールの代替実装へプラグインできます (オーバーライド)。また、サービスロールのベースセットを越えた追加サービスを定義できます (拡張)。

12.9.2. サービスコントラクト

マーカーインターフェース org.hibernate.service.Service を実装することがサービスの基本的な要件になります。Hibernate は基本的なタイプセーフのために内部でこのインターフェースを使用します。

起動と停止の通知を受け取るために、サービスは org.hibernate.service.spi.Startable および org.hibernate.service.spi.Stoppable インターフェースを任意で実装することもできます。その他に、JMX 統合が有効になっている場合に JMX でサービスを管理可能としてマーク付けする org.hibernate.service.spi.Manageable という任意のサービスコントラクトがあります。

12.9.3. サービス依存関係のタイプ

サービスは、以下の 2 つの方法のいずれかを使用して、他のサービスに依存関係を宣言できます。

@org.hibernate.service.spi.InjectService

単一のパラメーターを受け取るサービス実装クラスのメソッドと **@InjectService** アノテーションが付けられているメソッドは、他のサービスのインジェクションを要求していると見なされます。

デフォルトでは、メソッドパラメーターのタイプは、インジェクトされるサービスロールであると想定されます。パラメータータイプがサービスロールではない場合は、**InjectService** の **serviceRole** 属性を使用してロールを明示的に指定する必要があります。

デフォルトでは、インジェクトされたサービスは必須のサービスであると見なされます。そのため、名前付けされた依存サービスがない場合は、起動に失敗します。インジェクトされるサービスが任意のサービスである場合は、**InjectService** の required 属性を **false** (デフォルト値は **true**) として宣言する必要があります。

org.hibernate.service.spi.ServiceRegistryAwareService

2 つ目の方法は、単一の **injectServices** メソッドを宣言する任意のサービスインターフェース **org.hibernate.service.spi.ServiceRegistryAwareService** をサービスが実装する方法です。

起動中、Hibernate は **org.hibernate.service.ServiceRegistry** 自体をこのインターフェースを実装するサービスにインジェクトします。その後、サービスは **ServiceRegistry** 参照を使用して、必要な他のサービスを見つけることができます。

12.9.4. サービスレジストリー

12.9.4.1. ServiceRegistry

サービス自体以外の中央サービス API は org.hibernate.service.ServiceRegistry インターフェースです。サービスレジストリーの主な目的は、サービスを保持および管理し、サービスへのアクセスを提供することです。

サービスレジストリーは階層的です。レジストリーのサービスは、同じレジストリーおよび親レジストリーにあるサービスへの依存や利用が可能です。

org.hibernate.service.ServiceRegistryBuilder を使用して org.hibernate.service.ServiceRegistry インスタンスをビルドします。

例: ServiceRegistryBuilder を使用した ServiceRegistry の作成

```
ServiceRegistryBuilder registryBuilder = new ServiceRegistryBuilder(
bootstrapServiceRegistry );
    ServiceRegistry serviceRegistry =
registryBuilder.buildServiceRegistry();
```

12.9.5. カスタムサービス

12.9.5.1. カスタムサービス

org.hibernate.service.ServiceRegistry がビルドされると、不変であると見なされます。サービス自体は再設定を許可することもありますが、ここで言う不変とはサービスの追加や置換を意味します。そのため org.hibernate.service.ServiceRegistryBuilder によって提供される別のロールにより、生成された org.hibernate.service.ServiceRegistry に格納されるサービスを微調整できるようになります。

カスタムサービスについて org.hibernate.service.ServiceRegistryBuilder に通知する方法は 2 つあります。

- org.hibernate.service.spi.BasicServiceInitiator クラスを実装してサービスクラスの要求に応じた構築を制御し、addInitiator メソッドを介して org.hibernate.service.ServiceRegistryBuilder へ追加します。
- サービスクラスをインスタンス化し、addService メソッドを介して org.hibernate.service.ServiceRegistryBuilder へ追加します。

サービスを追加する方法とイニシエーターを追加する方法はいずれも、レジストリーの拡張 (新しいサービスロールの追加) やサービスのオーバーライド (サービス実装の置換) に対して有効です。

例: ServiceRegistryBuilder を用いた既存サービスのカスタマーサービスへの置き換え

```
   ServiceRegistryBuilder registryBuilder = new ServiceRegistryBuilder(
bootstrapServiceRegistry );
   registryBuilder.addService( JdbcServices.class, new FakeJdbcService()
);
   ServiceRegistry serviceRegistry =
registryBuilder.buildServiceRegistry();


   public class FakeJdbcService implements JdbcServices{

       @Override
       public ConnectionProvider getConnectionProvider() {
           return null;
       }

       @Override
       public Dialect getDialect() {
           return null;
       }

       @Override
       public SqlStatementLogger getSqlStatementLogger() {
           return null;
       }

       @Override
       public SqlExceptionHelper getSqlExceptionHelper() {
           return null;
       }

       @Override
       public ExtractedDatabaseMetaData getExtractedMetaDataSupport() {
           return null;
       }

       @Override
       public LobCreator getLobCreator(LobCreationContext
lobCreationContext) {
           return null;
       }
```

```
        @Override
        public ResultSetWrapper getResultSetWrapper() {
            return null;
        }
    }
```

12.9.6. ブートストラップレジストリー

12.9.6.1. ブートストラップレジストリー

ブートストラップレジストリーは、ほとんどの操作を行うために必ず必要になるサービスを保持します。主なサービスは ClassLoaderService (代表的な例) です。設定ファイルの解決にもクラスローディングサービス (リソースのルックアップ) へのアクセスが必要になります。通常の使用では、これがルートレジストリー (親なし) になります。

ブートストラップレジストリーのインスタンスは **`org.hibernate.service.BootstrapServiceRegistryBuilder`** クラスを使用してビルドされます。

12.9.6.2. BootstrapServiceRegistryBuilder の使用

例: BootstrapServiceRegistryBuilder の使用

```
BootstrapServiceRegistry bootstrapServiceRegistry = new
BootstrapServiceRegistryBuilder()
        // pass in org.hibernate.integrator.spi.Integrator instances which
are not
        // auto-discovered (for whatever reason) but which should be
included
        .with( anExplicitIntegrator )
        // pass in a class loader that Hibernate should use to load
application classes
        .with( anExplicitClassLoaderForApplicationClasses )
        // pass in a class loader that Hibernate should use to load
resources
        .with( anExplicitClassLoaderForResources )
        // see BootstrapServiceRegistryBuilder for rest of available
methods
        ...
        // finally, build the bootstrap registry with all the above
options
        .build();
```

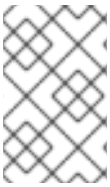
12.9.6.3. BootstrapRegistry サービス

`org.hibernate.service.classloading.spi.ClassLoaderService`

Hibernate はクラスローダーと対話する必要がありますが、Hibernate (またはライブラリー) がクラスローダーと対話する方法は、アプリケーションをホストするランタイム環境によって異なります。クラスローディングの要件は、アプリケーションサーバー、OSGi コンテナー、およびその他のモジュラークラスローディングシステムによって限定されます。このサービスは、このような環境の複雑性の抽象化を Hibernate に提供しますが、単一のスワップ可能なコンポーネントを用いることも重要な点になります。

クラスローダーとの対話では、Hibernate に以下の機能が必要になります。

- アプリケーションクラスを見つける機能
- 統合クラスを見つける機能
- リソース (プロパティーファイル、xml ファイルなど) を見つける機能
- java.util.ServiceLoader をロードする機能



注記

現在、アプリケーションクラスをロードする機能と統合クラスをロードする機能は、サービス上の 1 つの「ロードクラス」機能として組み合わされていますが、今後のリリースで変更になる可能性があります。

`org.hibernate.integrator.spi.IntegratorService`

アプリケーション、アドオン、およびその他のモジュールは Hibernate と統合する必要があります。以前の方法では、各モジュールの登録を調整するためにコンポーネント (通常はアプリケーション) が必要でした。この登録は各モジュールのインタグレーターの代わりに実行されました。

このサービスはディスカバリーに重点を置きます。
org.hibernate.service.classloading.spi.ClassLoaderService によって提供される標準の Java java.util.ServiceLoader 機能を使用して、org.hibernate.integrator.spi.Integrator コントラクトの実装を検出します。

インテグレーターは **`/META-INF/services/org.hibernate.integrator.spi.Integrator`** という名前のファイルを定義し、クラスパス上で使用できるようにします。

このファイルは **`java.util.ServiceLoader`** メカニズムによって使用され、**`org.hibernate.integrator.spi.Integrator`** インターフェースを実装するクラスの完全修飾名を 1 行に 1 つずつリストします。

12.9.7. SessionFactory レジストリー

12.9.7.1. SessionFactory レジストリー

すべてのレジストリータイプのインスタンスを指定の **`org.hibernate.SessionFactory`** のターゲットとして扱うことが最良の方法ですが、このグループのサービスのインスタンスは明示的に 1 つの **`org.hibernate.SessionFactory`** に属します。

違いは開始する必要があるタイミングになります。一般的に開始する **`org.hibernate.SessionFactory`** にアクセスする必要があります。この特別なレジストリーは **`org.hibernate.service.spi.SessionFactoryServiceRegistry`** です。

12.9.7.2. SessionFactory サービス

`org.hibernate.event.service.spi.EventListenerRegistry`

説明

イベントリスナーを管理するサービス。

イニシエーター

org.hibernate.event.service.internal.EventListenerServiceInitiator

実装

org.hibernate.event.service.internal.EventListenerRegistryImpl

12.9.8. インテグレーター

12.9.8.1. インテグレーター

org.hibernate.integrator.spi.Integrator の目的は、機能する SessionFactory のビルドプロセスに開発者がフックできるようにする簡単な手段を提供することです。**org.hibernate.integrator.spi.Integrator** インターフェースは、ビルドプロセスにフックできるようにする **integrate** と、終了する SessionFactory にフックできるようにする **disintegrate** の 2 つのメソッドを定義します。

注記

org.hibernate.cfg.Configuration の代わりに **org.hibernate.metamodel.source.MetadataImplementor** を受け入れるオーバーロード形式の integrate は、**org.hibernate.integrator.spi.Integrator** で定義される 3 つ目のメソッドになります。この形式は 5.0 で完了予定であった新しいメタモデルコードでの使用向けです。

IntegratorService によって提供されるディスカバリー以外に、BootstrapServiceRegistry のビルド時にアプリケーションはインテグレーターを手動で登録することができます。

12.9.8.2. インテグレーターのユースケース

現在、**org.hibernate.integrator.spi.Integrator** の主なユースケースは、イベントリスナーの登録とサービスの提供です (**org.hibernate.integrator.spi.ServiceContributingIntegrator** を参照)。5.0 では、オブジェクトとリレーショナルモデルとの間のマッピングを定義するメタモデルを変更できるようにするための拡張を計画しています。

例: イベントリスナーの登録

```
public class MyIntegrator implements
org.hibernate.integrator.spi.Integrator {

    public void integrate(
            Configuration configuration,
            SessionFactoryImplementor sessionFactory,
            SessionFactoryServiceRegistry serviceRegistry) {
        // As you might expect, an EventListenerRegistry is the thing with
which event listeners are registered  It is a
        // service so we look it up using the service registry
        final EventListenerRegistry eventListenerRegistry =
serviceRegistry.getService( EventListenerRegistry.class );

        // If you wish to have custom determination and handling of
"duplicate" listeners, you would have to add an
        // implementation of the
org.hibernate.event.service.spi.DuplicationStrategy contract like this
        eventListenerRegistry.addDuplicationStrategy(
```

```
myDuplicationStrategy );

        // EventListenerRegistry defines 3 ways to register listeners:
        //     1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners( EventType.AUTO_FLUSH,
myCompleteSetOfListeners );
        //     2) This form adds the specified listener(s) to the
beginning of the listener chain
        eventListenerRegistry.prependListeners( EventType.AUTO_FLUSH,
myListenersToBeCalledFirst );
        //     3) This form adds the specified listener(s) to the end of
the listener chain
        eventListenerRegistry.appendListeners( EventType.AUTO_FLUSH,
myListenersToBeCalledLast );
    }
}
```

12.10. ENVERS

12.10.1. Hibernate Envers

Hibernate Envers は監査およびバージョニングシステムであり、永続クラスへの変更履歴を追跡する方法を JBoss EAP に提供します。エンティティーに対する変更履歴を保存する **@Audited** アノテーションが付けられたエンティティーに対して監査テーブルが作成されます。その後、データの取得と問い合わせが可能になります。

Envers により開発者は次の作業を行うことが可能になります。

- JPA 仕様によって定義されるすべてのマッピングの監査
- JPA 仕様を拡張するすべての Hibernate マッピングの監査
- ネイティブ Hibernate API によりマッピングされた監査エンティティー
- リビジョンエンティティーを用いて各リビジョンのデータをログに記録
- 履歴データのクエリー

12.10.2. 永続クラスの監査

JBoss EAP では、Hibernate Envers と **@Audited** アノテーションを使用して永続クラスの監査を行います。アノテーションがクラスに適用されると、エンティティーのリビジョン履歴が保存されるテーブルが作成されます。

クラスに変更が加えられるたびに監査テーブルにエントリーが追加されます。エントリーにはクラスへの変更が含まれ、リビジョン番号が付けられます。そのため、変更をロールバックしたり、以前のリビジョンを表示したりすることが可能です。

12.10.3. 監査ストラテジー

12.10.3.1. 監査ストラテジー

監査ストラテジーは、監査情報の永続化、クエリー、および格納の方法を定義します。Hibernate Envers には、現在 2 つの監査ストラテジーが存在します。

デフォルトの監査ストラテジー

- このストラテジーは監査データと開始リビジョンを共に永続化します。監査テーブルで挿入、更新、削除された各行については、開始リビジョンの有効性と合わせて、1 つ以上の行が監査テーブルに挿入されます。
- 監査テーブルの行は挿入後には更新されません。監査情報のクエリーはサブクエリーを使い監査テーブルの該当行を選択します (これは時間がかかり、インデックス化が困難です)。

妥当性監査ストラテジー

- このストラテジーは監査情報の開始リビジョンと最終リビジョンの両方を格納します。監査テーブルで挿入、更新、または削除された各行については、開始リビジョンの有効性とあわせて、1 つ以上の行が監査テーブルに挿入されます。
- 同時に、以前の監査行 (利用可能な場合) の最終リビジョンフィールドがこのリビジョンに設定されます。監査情報に対するクエリーは、サブクエリーの代わりに**開始と最終リビジョンのいずれかを**使用します。つまり、更新の数が増えるため監査情報の永続化には今までより少し時間がかかりますが、監査情報の取得は非常に早くなります。
- インデックスを増やすことで改善することも可能です。

監査の詳細については、永続クラスの監査を参照してください。アプリケーションの監査ストラテジーを設定するには、 監査ストラテジーの設定を参照してください。

12.10.3.2. 監査ストラテジーの設定

JBoss EAP では、2 つの監査ストラテジーがサポートされます。

- デフォルトの監査ストラテジー
- 妥当性監査ストラテジー

監査ストラテジーの定義
アプリケーションの **`persistence.xml`** ファイルで **`org.hibernate.envers.audit_strategy`** プロパティーを設定します。このプロパティーが **`persistence.xml`** ファイルで設定されていない場合は、デフォルトの監査ストラテジーが使用されます。

デフォルトの監査ストラテジーの設定

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.DefaultAuditStrategy"/>
```

妥当性監査ストラテジーの設定

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.ValidityAuditStrategy"/>
```

12.10.4. JPA エンティティーへの監査サポートの追加

JBoss EAP は、エンティティーの監査を使用して (Hibernate Envers を参照)、永続クラスの変更履歴を追跡します。本トピックでは、JPA エンティティーに対する監査サポートを追加する方法について説明します。

JPA エンティティーへの監査サポートの追加

1. デプロイメントに適した使用可能な監査パラメーターを設定します (Envers パラメーターの設定を参照)。
2. 監査対象となる JPA エンティティーを開きます。
3. **`org.hibernate.envers.Audited`** インターフェースをインポートします。
4. 監査対象となる各フィールドまたはプロパティーに **`@Audited`** アノテーションを付けます。または、1 度にクラス全体へアノテーションを付けます。

例: 2 つのフィールドの監査

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
public class Person {
    @Id
    @GeneratedValue
    private int id;

    @Audited
    private String name;

    private String surname;

    @ManyToOne
    @Audited
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

例: クラス全体の監査

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
@Audited
public class Person {
    @Id
    @GeneratedValue
    private int id;

    private String name;
```

```
    private String surname;

    @ManyToOne
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

JPA エンティティーの監査が設定されると、変更履歴を保存するために **_AUD** という名前のテーブルが作成されます。

12.10.5. Configuration (設定)

12.10.5.1. Envers パラメーターの設定

JBoss EAP は、Hibernate Envers からエンティティーの監査を使用して、永続クラスの変更履歴を追跡します。

利用可能な Envers パラメーターの設定

1. アプリケーションの **persistence.xml** ファイルを開きます。
2. 必要に応じて Envers プロパティーを追加、削除、または設定します。使用可能なプロパティーの一覧については、Envers の設定プロパティーを参照してください。

   例: Envers パラメーター

```
<persistence-unit name="mypc">
  <description>Persistence Unit.</description>
  <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-
source>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create-drop" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.cache.use_second_level_cache"
value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
    <property name="hibernate.generate_statistics" value="true" />
    <property name="org.hibernate.envers.versionsTableSuffix"
value="_V" />
    <property name="org.hibernate.envers.revisionFieldName"
value="ver_rev" />
  </properties>
</persistence-unit>
```

12.10.5.2. ランタイム時に監査を有効または無効にする

ランタイム時にエンティティーバージョン監査を有効または無効にする

1. **AuditEventListener** クラスをサブクラス化します。

2. Hibernate イベント上で呼び出される次のメソッドを上書きします。
   - **onPostInsert**
   - **onPostUpdate**
   - **onPostDelete**
   - **onPreUpdateCollection**
   - **onPreRemoveCollection**
   - **onPostRecreateCollection**
3. イベントのリスナーとしてサブクラスを指定します。
4. 変更を監査すべきであるか判断します。
5. 変更を監査する必要がある場合は、呼び出しをスーパークラスへ渡します。

12.10.5.3. 条件付き監査の設定

Hibernate Envers は一連のイベントリスナーを使用して、さまざまな Hibernate イベントに対して監査データを永続化します。Envers jar がクラスパスにある場合、これらのリスナーは自動的に登録されます。

条件付き監査の実装

1. **persistence.xml** ファイルで **hibernate.listeners.envers.autoRegister** の Hibernate プロパティーを false に設定します。
2. 上書きする各イベントリスナーをサブクラス化します。条件付き監査の論理をサブクラスに置き、監査の実行が必要な場合はスーパーメソッドを呼び出します。
3. **org.hibernate.envers.event.EnversIntegrator** と似ている **org.hibernate.integrator.spi.Integrator** のカスタム実装を作成します。デフォルトのクラスではなく、手順 2 で作成したイベントリスナーサブクラスを使用します。
4. jar に **META-INF/services/org.hibernate.integrator.spi.Integrator** ファイルを追加します。このファイルにはインターフェースを実装するクラスの完全修飾名を含める必要があります。

12.10.5.4. Envers の設定プロパティー

表12.24 エンティティーデータのバージョニング設定パラメーター

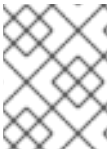
プロパティー名	デフォルト値	説明
org.hibernate.envers.audit_table_prefix	デフォルト値なし	監査エンティティーの名前の前に付けられた文字列。監査情報を保持するエンティティーの名前を作成します。

プロパティー名	デフォルト値	説明
org.hibernate.envers.audit_table_suffix	_AUD	監査情報を保持するエンティティーの名前を作成する監査エンティティーの名前に追加された文字列。たとえば、**Person** のテーブル名を持つエンティティーが監査される場合は、Envers により履歴データを格納する **Person_AUD** と呼ばれるテーブルが生成されます。
org.hibernate.envers.revision_field_name	REV	改訂番号を保持する監査エンティティーのフィールド名。
org.hibernate.envers.revision_type_field_name	REVTYPE	リビジョンタイプを保持する監査エンティティーのフィールド名。挿入、変更、または削除のための現在のリビジョンタイプは、それぞれ **add**、**mod**、および **del** です。
org.hibernate.envers.revision_on_collection_change	true	このプロパティーは、所有されていない関係フィールドが変更された場合にリビジョンを生成するかどうかを決定します。これは、一対多関係のコレクションまたは一対一関係の **mappedBy** 属性を使用したフィールドのいずれかです。
org.hibernate.envers.do_not_audit_optimistic_locking_field	true	true の場合、オプティミスティックロッキングに使用したプロパティー (**@Version** アノテーションが付いたもの) は自動的に監査から除外されます。
org.hibernate.envers.store_data_at_delete	false	このプロパティーは、ID のみではなく、他の全プロパティーが null とマークされたエンティティーが削除される場合にエンティティーデータをリビジョンに保存すべきかどうかを定義します。このデータは最終リビジョンに存在するため、これは通常必要ありません。最終リビジョンのデータにアクセスする方が簡単で効率的ですが、この場合、削除前にエンティティーに含まれたデータが 2 回保存されることになります。
org.hibernate.envers.default_schema	null (通常のテーブルと同じ)	監査テーブルに使用されるデフォルトのスキーマ名。**@AuditTable(schema="…")** アノテーションを使用してオーバーライドできます。このスキーマがない場合、スキーマは通常のテーブルのスキーマと同じです。

プロパティー名	デフォルト値	説明
org.hibernate.envers.default_catalog	null (通常のテーブルと同じ)	監査テーブルに使用するデフォルトのカタログ名。**@AuditTable(catalog="…")** アノテーションを使用してオーバーライドできます。このカタログがない場合、カタログは通常のテーブルのカタログと同じです。
org.hibernate.envers.audit_strategy	**org.hibernate.envers.strategy.DefaultAuditStrategy**	このプロパティーは、監査データを永続化する際に使用する監査ストラテジーを定義します。デフォルトでは、エンティティーが変更されたリビジョンのみが保存されます。あるいは、**org.hibernate.envers.strategy.ValidityAuditStrategy** が、開始リビジョンと最終リビジョンの両方を保存します。これらは、監査行が有効である場合に定義されます。
org.hibernate.envers.audit_strategy_validity_end_rev_field_name	REVEND	監査エンティティーのリビジョン番号を保持するカラムの名前。このプロパティーは、妥当性監査ストラテジーが使用されている場合のみ有効です。
org.hibernate.envers.audit_strategy_validity_store_revend_timestamp	false	このプロパティーは、データが最後に有効だった最終リビジョンのタイムスタンプを最終リビジョンとともに格納するかどうかを定義します。これは、テーブルパーティショニングを使用することにより、関係データベースから以前の監査レコードを削除する場合に役に立ちます。パーティショニングには、テーブル内に存在する列が必要です。このプロパティーは、**ValidityAuditStrategy** が使用される場合にのみ評価されます。
org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name	REVEND_TSTMP	データが有効であった最終リビジョンのタイムスタンプの列名。**ValidityAuditStrategy** が使用され、**org.hibernate.envers.audit_strategy_validity_store_revend_timestamp** が true と評価された場合のみ使用されます。

12.10.6. クエリーを介した監査情報の読み出し

Hibernate Envers は、クエリーから監査情報を読み出しする機能を提供します。



注記

監査されたデータのクエリーは相関サブセレクトが関与するため、多くの場合で **live** データの対応するクエリーよりも大幅に処理が遅くなります。

特定のリビジョンでクラスのエンティティーをクエリーする

このようなクエリーのエントリーポイントは次のとおりです。

```
AuditQuery query = getAuditReader()
    .createQuery()
    .forEntitiesAtRevision(MyEntity.class, revisionNumber);
```

AuditEntity ファクトリクラスを使用して制約を指定することができます。以下のクエリーは、**name** プロパティーが **John** と同等である場合のみエンティティーを選択します。

```
query.add(AuditEntity.property("name").eq("John"));
```

以下のクエリーは特定のエンティティーと関連するエンティティーのみを選択します。

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

結果を順序付けや制限付けしたり、凝集 (aggregations) および射影 (projections) のセット (グループ化を除く) を持つことが可能です。以下はフルクエリーの例になります。

```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision(Person.class, 12)
    .addOrder(AuditEntity.property("surname").desc())
    .add(AuditEntity.relatedId("address").eq(addressId))
    .setFirstResult(4)
    .setMaxResults(2)
    .getResultList();
```

特定クラスのエンティティーが変更された場合のクエリーリビジョン

このようなクエリーのエントリーポイントは次のとおりです。

```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true);
```

前の例と同様に、このクエリーへ制約を追加することが可能です。このクエリーに以下を追加することも可能です。

AuditEntity.revisionNumber()

監査されたエンティティーが修正されたリビジョン番号の制約や射影、順序付けを指定します。

AuditEntity.revisionProperty(propertyName)

監査されたエンティティーが修正されたリビジョンに対応するリビジョンエンティティーのプロパティーの制約や射影、順序付けを指定します。

AuditEntity.revisionType()

リビジョンのタイプ (ADD、MOD、DEL) へのアクセスを提供します。

クエリー結果を必要に応じて調整することが可能です。次のクエリーは、リビジョン番号 42 の後に **entityId** ID を持つ **MyEntity** クラスのエンティティーが変更された最小のリビジョン番号を選択します。

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.id().eq(entityId))
    .add(AuditEntity.revisionNumber().gt(42))
    .getSingleResult();
```

リビジョンのクエリーはプロパティーを最小化および最大化することも可能です。次のクエリーは、特定エンティティーの **actualDate** 値が指定の値よりは大きく、可能な限り小さいリビジョンを選択します。

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    // We are only interested in the first revision
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.property("actualDate").minimize()
        .add(AuditEntity.property("actualDate").ge(givenDate))
        .add(AuditEntity.id().eq(givenEntityId)))
    .getSingleResult();
```

minimize() および **maximize()** メソッドは制約を追加できる基準を返します。最大化または最小化されたプロパティーを持つエンティティーはこの基準を満たさなければなりません。

クエリー作成時に渡されるブール変数パラメーターは 2 つあります。

selectEntitiesOnly

このパラメーターは、明示的なプロジェクションが設定されていない場合のみ有効です。
true の場合、クエリーの結果は指定された制約を満たすリビジョンで変更されたエンティティーのリストです。
false の場合、結果は 3 つの要素アレイのリストです。最初の要素は変更されたエンティティーインスタンスです。2 つ目の要素はリビジョンデータを含むエンティティーです。カスタムエンティティーが使用されない場合、これは **DefaultRevisionEntity** のインスタンスです。3 つ目の要素アレイはリビジョンの種類 (ADD、MOD、DEL) です。

selectDeletedEntities

このパラメーターは、エンティティーが削除されたリビジョンが結果に含まれなければならない場合に指定されます。true の場合、エンティティーのリビジョンタイプが **DEL** になり、id 以外のすべてのフィールドの値が **null** になります。

特定のプロパティーを修正したエンティティーのクエリーリビジョン

下記のクエリーは、**actualDate** プロパティーが変更された、指定の ID を持つ **MyEntity** のすべてのリビジョンを返します。

```
AuditQuery query = getAuditReader().createQuery()
  .forRevisionsOfEntity(MyEntity.class, false, true)
  .add(AuditEntity.id().eq(id));
  .add(AuditEntity.property("actualDate").hasChanged())
```

hasChanged 条件は他の基準と組み合わせることができます。次のクエリーは、revisionNumber の生成時に **MyEntity** の水平スライスを返します。これは、**prop2** ではなく **prop1** を変更したリビジョンに限定されます。

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

結果セットには revisionNumber よりも小さい番号のリビジョンも含まれます。これは、このクエリーが「**prop1** が変更され、**prop2** が変更されない revisionNumber で変更された **MyEntities** をすべて返す」とは読み取られないことを意味します。

次のクエリーは **forEntitiesModifiedAtRevision** クエリーを使用してこの結果をどのように返すことができるかを示します。

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesModifiedAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

特定のリビジョンで修正されたクエリーエンティティー

次の例は、特定のリビジョンで変更されたエンティティーに対する基本的なクエリーになります。読み出される特定のリビジョンで、エンティティー名と対応する Java クラスを変更できます。

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()
    .getCrossTypeRevisionChangesReader().findEntityTypes(revisionNumber);
```

org.hibernate.envers.CrossTypeRevisionChangesReader からもアクセスできる他のクエリーは以下のとおりです。

List<Object> findEntities(Number)

特定のリビジョンで変更 (追加、更新、削除）されたすべての監査済みエンティティーのスナップショットを返します。**n+1** 個の SQL クエリを実行します (**n** は指定のリビジョン内で変更された異なるエンティティークラスの数になります)。

List<Object> findEntities(Number, RevisionType)

変更タイプによってフィルターされた特定のリビジョンで変更 (追加、更新、削除) されたすべての監査済みエンティティーのスナップショットを返します。**n+1** 個の SQL クエリーを実行します (**n** は指定のリビジョン内で変更された異なるエンティティークラスの数です)。Map<RevisionType, List<Object>>

findEntitiesGroupByRevisionType(Number)

修正操作 (追加、更新、削除など) によってグループ化されたエンティティースナップショットの一覧が含まれるマップを返します。**3n+1** 個の SQL クエリーを実行します (**n** は指定のリビジョン内で変更された異なるエンティティークラスの数になります)。

12.11. パフォーマンスチューニング

12.11.1. 代替のバッチローディングアルゴリズム

Hibernate では、4 つのフェッチングストラテジー (join、select、subselect、および batch) のいずれかを使用してアソシエーションのデータをロードできます。batch ローディングは select フェッチングの

最適化ストラテジーであるため、パフォーマンスを最大化できます。このストラテジーでは、主キーまたは外部キーのリストを指定することで、Hibernate が単一の SELECT ステートメントでエンティティーインスタンスまたはコレクションのバッチを読み出します。batch フェッチングは、レイジー select フェッチングストラテジーの最適化です。

batch フェッチングを設定する方法には、クラスごとのレベルと、コレクションごとのレベルの 2 つの方法があります。

- クラスごとのレベル
  Hibernate がクラスごとのレベルでデータをロードする場合、クエリー時に事前ロードするアソシエーションのバッチサイズが必要になります。たとえば、起動時に **car** オブジェクトの 30 個のインスタンスがセッションでロードされるとします。各 **car** オブジェクトは **owner** オブジェクトに属します。**lazy** ローディングで、すべての **car** オブジェクトを繰り返し、これらの owner (所有者) を要求する場合、Hibernate は owner ごとに 1 つ、合計 30 個の select ステートメントを発行します。これは、パフォーマンス上のボトルネックになります。

  この代わりに、クエリーによって要求される前に次の owner のバッチに対してデータを事前ロードするよう Hibernate を指示できます。**owner** オブジェクトがクエリーされると、Hibernate は同じ SELECT ステートメントでこれらのオブジェクトをさらに多くクエリーします。

  事前にクエリーされる **owner** オブジェクトの数は、設定時に指定された **batch-size** パラメーターによって異なります。

  ```
  <class name="owner" batch-size="10"></class>
  ```

  これは、今後必要になると見込まれる最低 10 個の **owner** オブジェクトをクエリーするよう Hibernate に指示します。ユーザーが **car A** の **owner** をクエリーする場合、**car B** の **owner** はすでにバッチローディングの一部としてロードされていることがあります。ユーザーが実際に **car B** の **owner** を必要とする場合、データーベースにアクセスして SELECT ステートメントを発行する代わりに、現在のセッションから値を読み出すことができます。

  Hibernate 4.2.0 には、**batch-size** パラメーターの他に、バッチローディングのパフォーマンスを向上する新しい設定項目が追加されました。この設定項目は **Batch Fetch Style** 設定と呼ばれ、**hibernate.batch_fetch_style** パラメーターによって指定されます。

  LEGACY、PADDED、および DYNAMIC の 3 つのバッチフェッチスタイルがサポートされます。使用するスタイルを指定するには、**org.hibernate.cfg.AvailableSettings#BATCH_FETCH_STYLE** を使用します。

  - LEGACY: LEGACY スタイルのローディングでは、**ArrayHelper.getBatchSizes(int)** に基づいた一連の事前ビルド済みバッチサイズが使用されます。バッチは、既存のバッチ可能な識別子の数から、次に小さい事前ビルド済みバッチサイズを使用してロードされます。
    前述の例を用いた場合、**batch-size** の設定が 30 であると、事前ビルドされたバッチサイズは [30, 15, 10, 9, 8, 7, .., 1] になります。29 個の識別子をバッチロードしようとすると、バッチは 15、10、および 4 になります。対応する 3 つの SQL クエリーが生成され、各クエリーはデータベースより 15、10、および 4 人の owner をロードします。

  - PADDED - PADDED は LEGACY スタイルのバッチローディングと似ています。PADDED も事前ビルドされたバッチサイズを使用しますが、次に大きなバッチサイズを使用し、余分な識別子プレースホルダーを埋め込みます。
    上記の例で、30 個の owner オブジェクトが初期化される場合は、データベースに対して 1 つのクエリーのみが実行されます。

29 個の owner オブジェクトが初期化される場合、Hibernate は同様にバッチサイズが 30 の SQL select ステートメントを 1 つ実行しますが、余分なスペースが繰り替えされる識別子で埋め込みされます。

- DYNAMIC - DYNAMIC スタイルのバッチローディングはバッチサイズの制限に準拠しますが、実際にロードされるオブジェクトの数を使用して SQL SELECT ステートメントを動的にビルドします。
  たとえば、30 個の owner オブジェクトで最大バッチサイズが 30 の場合、 30 個の owner オブジェクトの読み出しは 1 つの SQL SELECT ステートメントによって実行されます。35 個の owner オブジェクトを読み出す場合は、SQL ステートメントが 2 つになり、それぞれのバッチサイズが 30 と 5 になります。Hibernate は、制限どおりにバッチサイズを 30 以下とし、2 つ目の SQL ステートメントを動的に変更して、必要数である 5 にします。PADDED とは異なり、2 つ目の SQL は埋め込みされません。また、2 つ目の SQL ステートメントは動的に作成され、固定サイズでないことが LEGACY とは異なります。

  30 個未満のクエリーでは、このスタイルは要求された識別子の数のみを動的にロードします。

- コレクションごとのレベル
  Hibernate は、前項の「クラスごとのレベル」で説明したバッチフェッチサイズとスタイルを維持してコレクションをバッチロードすることもできます。

  前項の例を逆にして、各 **owner** オブジェクトによって所有されるすべての **car** オブジェクトをロードする必要があるとします。10 個の **owner** オブジェクトがすべての owner を繰り返し、現セッションにロードされた場合、**getCars()** メソッドの呼び出しごとに 1 つの SELECT ステートメントが生成されるため、合計で 10 個の SELECT ステートメントが生成されます。owner のマッピングでの car コレクションのバッチフェッチングを有効にすると、Hibernate は下記のようにこれらのコレクションを事前フェッチできます。

  ```
  <class name="Owner"><set name="cars" batch-size="5"></set></class>
  ```

  よって、バッチサイズが 5 でレガシーバッチスタイルを使用して 10 個のコレクションをロードする場合、Hibernate は 2 つの SELECT ステートメントを実行し、各ステートメントは 5 つのコレクションを読み出します。

12.11.2. 不変データのオブジェクト参照の 2 次キャッシング

Hibernate はパフォーマンスを向上するため、自動的にデータをメモリー内にキャッシュします。これは、データベースのルックアップが必要となる回数を削減する (特にほとんど変更されないデータに対し) インメモリーキャッシュによって実現されます。

Hibernate は 2 つのタイプのキャッシュを保持します。1 次キャッシュ (プライマリーキャッシュ) は必須のキャッシュです。このキャッシュは現在のセッションと関連し、すべてのリクエストが通過する必要があります。2 次キャッシュ (セカンダリーキャッシュ) は任意のキャッシュで、1 次キャッシュがアクセスされた後でのみアクセスされます。

データは、最初にステートアレイに逆アセンブルされ、2 次キャッシュに保存されます。このアレイはディープコピーされ、ディープコピーがキャッシュに格納されます。キャッシュからデータを読み取る場合は、この逆のプロセスが発生します。この仕組みは、変化するデータ (可変データ) ではうまく機能しますが、不変データでは不十分です。

データのディープコピーは、メモリーの使用と処理速度に負荷のかかる操作です。大きなデータセットでは、メモリーおよび処理速度がパフォーマンスを制限する要素になります。Hibernate では、不変データがコピーされずに参照されるよう指定できます。Hibernate はデータセット全体をコピーする代わりに、データへの参照をキャッシュに保存できます。

これは、設定 **hibernate.cache.use_reference_entries** の値を **true** に変更することによって行えます。デフォルトでは、**hibernate.cache.use_reference_entries** は **false** に設定されます。

hibernate.cache.use_reference_entries が **true** に設定されると、アソシエーションを持たない不変データオブジェクトは 2 次キャッシュにコピーされず、不変データオブジェクトへの参照のみが保存されます。

警告

hibernate.cache.use_reference_entries が **true** に設定されても、アソシエーションを持つ不変データオブジェクトは 2 次キャッシュにディープコピーされます。

第13章 HIBERNATE SEARCH

13.1. HIBERNATE SEARCH を初めて使う場合

13.1.1. Hibernate Search について

Hibernate Search は、Hibernate アプリケーションに全文検索機能を提供します。SQL ベースのソリューションが適切でない検索アプリケーションに特に適しています (全文、あいまい、および位置情報検索を含む)。Hibernate Search は全文検索エンジンとして Apache Lucene を使用しますが、メンテナンスオーバーヘッドを最小化するよう設計されています。設定後は、インデックス作成、クラスタリング、およびデータ同期化が透過的にメンテナンスされ、ユーザーはビジネス要件を満たすことに集中できます。

注記

JBoss EAP の以前のリリースには Hibernate 4.2 と Hibernate Search 4.6 が含まれていました。JBoss EAP 7 には、Hibernate 5 と Hibernate Search 5.5 が含まれます。

Hibernate Search 5.5 は Java 7 と連携し、Lucene 5.3.x に基づいて構築されています。ネイティブ Lucene API を使用している場合は、このバージョンを使用してください。

13.1.2. 概要

Hibernate Search は、Apache Lucene によりサポートされるインデックス作成コンポーネントとインデックス検索コンポーネントから構成されます。データベースに対してエンティティーが挿入、更新、または削除されるたびに、Hibernate Search はこのイベントを (Hibernate イベントシステム経由で) 追跡し、インデックスアップデートをスケジュールします。これらすべてのアップデートは、Apache Lucene API を直接使用せずに処理されます。代わりに、基礎となる Lucene インデックスとの対話は **`IndexManager`** 経由で処理されます。デフォルトでは、IndexManager と Lucene インデックス間には 1 対 1 の関係があります。IndexManager は選択された **back end**、**reader strategy**、および **DirectoryProvider** を含む特定のインデックス設定を抽象化します。

インデックスが作成されると、基礎となる Lucene インフラストラクチャーを使用する代わりに、エンティティーを検索し、管理対象エンティティーのリストを返すことができます。同じ永続化コンテキストが Hibernate と Hibernate Search 間で共有されます。**`FullTextSession`** クラスは、アプリケーションコードが統一された **`org.hibernate.Query`** または **`javax.persistence.Query`** API を HQL、JPA-QL、またはネイティブクエリーとまったく同じように使用できるように Hibernate **`Session`** クラス上に構築されます。

JDBC ベースであるかどうかに関係なく、すべての操作にはトランザクション形式のバッチモードが推奨されます。

注記

データベースと Hibernate Search の両方において、JDBC または JTA に関係なくトランザクションで操作を実行することが推奨されます。

注記

Hibernate Search は、Hibernate または EntityManager の長い会話パターン (アトミック会話) で完全に動作します。

13.1.3. Directory Provider について

Hibernate Search インフラストラクチャーの一部である Apache Lucene には、インデックスを格納するディレクトリーという概念があります。Hibernate Search は、**ディレクトリープロバイダー**経由で Lucene ディレクトリーの初期化と設定を処理します。

`directory_provider` プロパティーは、インデックスを格納するために使用するディレクトリープロバイダーを指定します。デフォルトのファイルシステムディレクトリープロバイダーは、**`filesystem`** であり、インデックスを格納するローカルファイルシステムを使用します。

13.1.4. ワーカーについて

Lucene インデックスに対するアップデートは、Hibernate Search **ワーカー**によって処理されます。ワーカーはすべてのエンティティーの変更を受け取り、それらをコンテキスト別にキューに格納し、コンテキストが終了したら適用します。最も一般的なコンテキストはトランザクションですが、エンティティーの変更または他のアプリケーションイベントの数によって異なることがあります。

効率を向上させるために、対話はバッチ処理され、一般的にコンテキストの終了時に適用されます。トランザクション外部では、インデックスアップデート操作は、実際のデータベース操作の直後に実行されます。実行中のトランザクションの場合は、トランザクションコミットフェーズに対してインデックスアップデート操作がスケジュールされ、トランザクションロールバックの場合は、破棄されます。ワーカーには特定のバッチサイズ制限を設定できます。この制限を超えると、コンテキストに関係なくインデックス作成が実行されます。

インデックスアップデートのこの処理方法は以下の 2 つの利点があります。

- パフォーマンス: Lucene インデックス作成のパフォーマンスは、操作をバッチで実行した場合に向上します。
- ACIDity: 実行したワークは、データベーストランザクションにより実行されたワークと同じスコープを持ち、トランザクションがコミットされた場合にのみ実行されます。これは、厳密には ACID ではありませんが、ACID の動作は全文検索インデックスにほとんど役に立ちません。インデックスはいつでもソースから再構築できます。

2 つのバッチモード (スコープなしと従来のもの) の関係は、オートコミットと従来の動作の関係に似ています。パフォーマンスの観点からは、**トランザクション**モードが推奨されます。スコープの選択は透過的に行われます。Hibernate Search は、トランザクションの存在を検出し、スコープを調整します。

13.1.5. バックエンドセットアップおよび操作

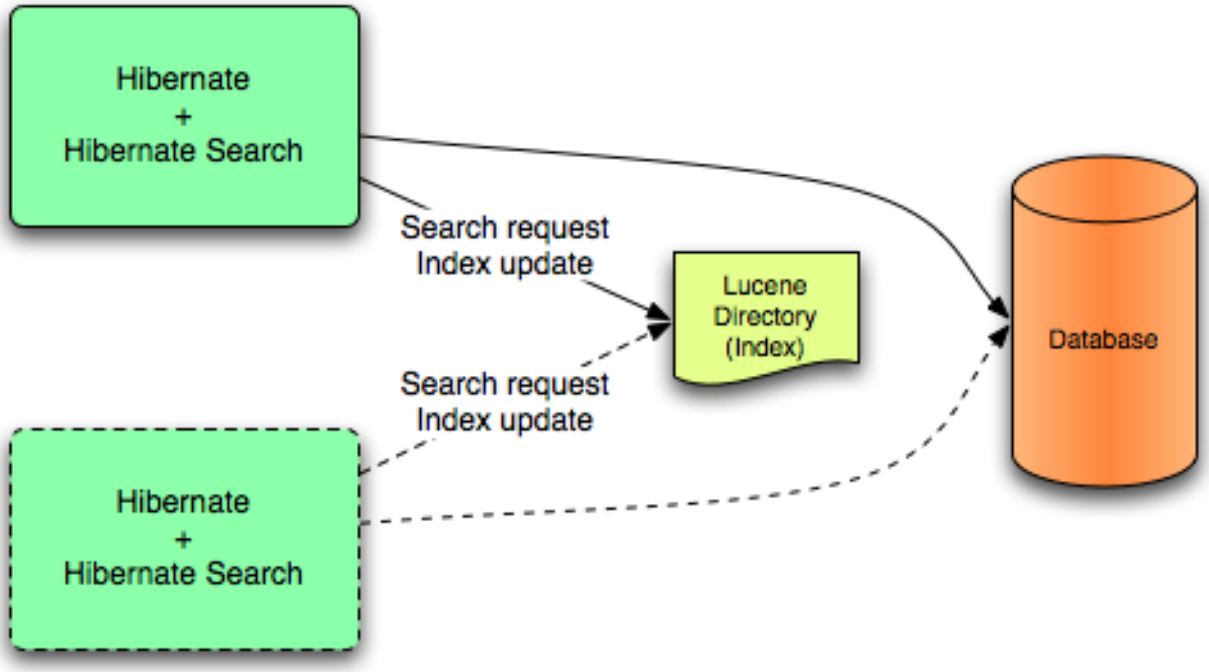
13.1.5.1. バックエンド

Hibernate Search は、さまざまなバックエンドを使用してワークのバッチを処理します。バックエンドは設定オプション **`default.worker.backend`** に制限されません。このプロパティーは、バックエンド設定の一部である **`BackendQueueProcessor`** インターフェースの実装を指定します。バックエンドをセットアップするには、JMS バックエンドなどの追加の設定が必要です。

13.1.5.2. Lucene

Lucene モードでは、ノードのすべてのインデックスアップデートが、ディレクトリープロバイダーを使用した Lucene ディレクトリーに対する同じノードにより実行されます。このモードは、非クラスター環境または共有ディレクトリーストアがあるクラスター環境で使用します。

図13.1 Lucene バックエンド設定

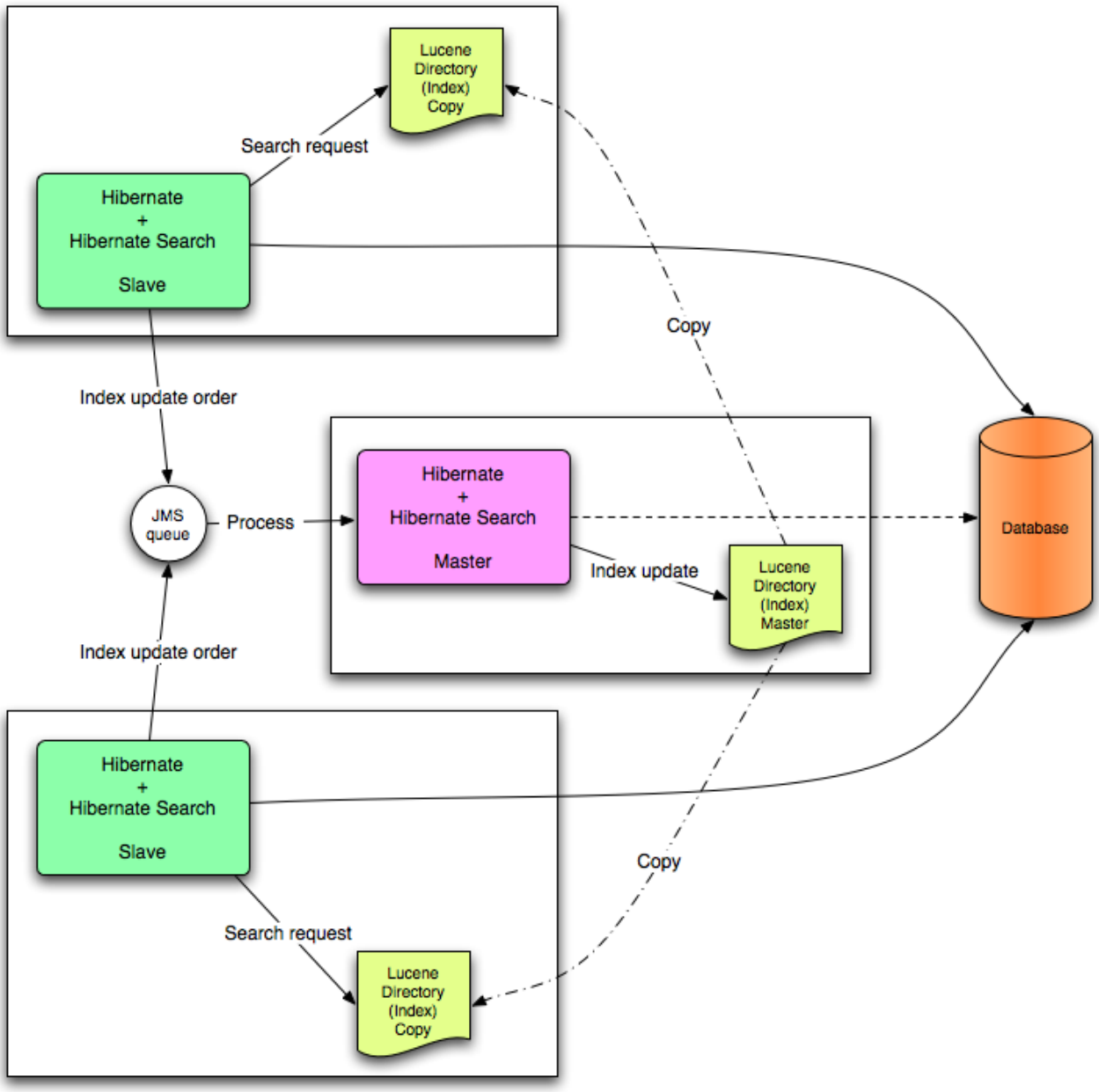


Lucene モードは、ディレクトリーでロックストラテジーを管理する非クラスターアプリケーションまたはクラスターアプリケーションを対象とします。Lucene モードの第一の利点は、Lucene クエリーの変更の単純化と即時的な可視性です。Near Real Time (NRT) バックエンドは非クラスターおよび非共有インデックス設定向けの代替バックエンドです。

13.1.5.3. JMS

ノードのインデックスアップデートは JMS キューに送信されます。特別なリーダーがキューを処理し、マスターインデックスをアップデートします。マスターおよびスレーブパターンを確立するために、マスターインデックスはスレーブコピーに定期的に複製されます。マスターは Lucene インデックスアップデートを行います。スレーブは読み書き操作を受け取りますが、読み取り操作をローカルインデックスコピーで処理します。Lucene インデックスを更新するのはマスターだけです。また、マスターのみがアップデート操作でローカル変更を適応できます。

図13.2 JMS バックエンド設定



このモードは、スループットが重要であり、インデックスアップデートの遅延が許容されるクラスタ環境を対象とします。JMS プロバイダーにより、信頼性が保証され、ローカルインデックスコピーを変更するためにスレーブが使用されます。

13.1.6. リーダーストラテジー

クエリーを実行する場合は、Hibernate Search がリーダーストラテジーを使用して Apache Lucene インデックスを処理します。頻繁なアップデート、読み取りが大部分、非同期インデックスアップデートなどのアプリケーションのプロファイルに基づいてリーダーストラテジーを選択します。

13.1.6.1. shared ストラテジー

`shared` ストラテジーを使用して、Hibernate Search は複数のクエリーおよびスレッドで該当する Lucene インデックスに対して同じ **`IndexReader`** を共有します (**`IndexReader`** がアップデートされたままの場合)。**`IndexReader`** がアップデートされない場合は、新しいものがオープンし、提供されま

す。各 **IndexReader** は複数の **SegmentReaders** から構成されます。shared ストラテジーは、最後のオープン後に変更または作成されたセグメントを再びオープンし、以前のインスタンスからすでにロードされたセグメントを共有します。これはデフォルトストラテジーです。

13.1.6.2. not-shared ストラテジー

not-shared ストラテジーを使用すると、クエリーが実行されるたびに Lucene **IndexReader** がオープンします。**IndexReader** のオープンおよび起動はコストがかかる操作です。したがって、各クエリー実行で **IndexReader** をオープンすることは効率的なストラテジーではありません。

13.1.6.3. カスタムリーダーストラテジー

カスタムリーダーストラテジーは、**org.hibernate.search.reader.ReaderProvider** の実装を使用して記述できます。実装はスレッドセーフである必要があります。

13.2. CONFIGURATION (設定)

13.2.1. 最小設定

Hibernate Search は、設定と操作に柔軟性を提供するよう設計されており、デフォルト値は大部分のユースケースに適合するよう慎重に選択されます。最低でも、**Directory Provider** とプロパティーを設定する必要があります。デフォルトの Directory Provider は **filesystem** であり、インデックスストレージにローカルファイルシステムが使用されます。利用可能な Directory Provider とその設定の詳細については、DirectoryProvider Configuration を参照してください。

Hibernate を直接使用している場合は、DirectoryProvider などの設定を設定ファイルの **hibernate.properties** または **hibernate.cfg.xml** で指定する必要があります。Hibernate を JPA で使用している場合、設定ファイルは **persistence.xml** になります。

13.2.2. IndexManager の設定

Hibernate Search は、このインターフェースに対して複数の実装を提供します。

- **directory-based**: Lucene **Directory** 抽象化を使用してインデックスファイルを管理するデフォルトの実装。
- **near-real-time**: 各コミット時にディスクへの書き込みのフラッシュを回避します。また、このインデックスマネージャーは **Directory** ベースですが、Lucene の Near Real-Time (NRT) 機能を使用します。

デフォルト値以外の IndexManager を指定するには、以下のプロパティーを指定します。

```
hibernate.search.[default|<indexname>].indexmanager = near-real-time
```

13.2.2.1. Directory-based

Directory-based 実装は、デフォルトの **IndexManager** 実装です。高度な設定が可能であり、リーダーストラテジー、バックエンド、およびディレクトリープロバイダーに対して個別の設定を指定できます。

13.2.2.2. Near Real Time

NRTIndexManager は、デフォルトの **IndexManager** の拡張であり、レイテンシーの低いインデックス書き込みに Lucene NRT (Near Real Time) 機能を使用します。ただし、**lucene** 以外の別のバックエンドに対する設定は無視され、**Directory** に対する排他的な書き込みロックが取得されます。

IndexWriter は、低いレイテンシーを提供するために各変更をディスクにフラッシュしません。クエリーは、フラッシュされていないインデックスライターバッファーからアップデートされた状態を読み取ることができます。ただし、**IndexWriter** が終了した場合やアプリケーションがクラッシュした場合は、アップデートが失われ、インデックスを再構築する必要があることがあります。

記載された欠点が原因でデータが制限された非クラスタ Web サイトには Near Real Time 設定が推奨されます (パフォーマンスを向上させるためにマスターノードを個別に設定できます)。

13.2.2.3. Custom

カスタマイズ **IndexManager** をセットアップするにはカスタム実装に完全修飾クラス名を指定します。以下のように、実装に対して引数のないコンストラクターをセットアップします。

```
[default|<indexname>].indexmanager = my.corp.myapp.CustomIndexManager
```

カスタムインデックスマネージャー実装では、デフォルト実装と同じコンポーネントが必要ありません。たとえば、**Directory** インターフェースを公開しないリモートインデックスサービスに委任します。

13.2.3. DirectoryProvider 設定

DirectoryProvider は、**Directory** に対する Hibernate Search の抽象化であり、基礎となる Lucene リソースの設定と初期化を処理します。ディレクトリープロバイダーおよびプロパティーには、Hibernate Search で利用可能なディレクトリープロバイダーと対応するオプションが示されます。

各インデックスエンティティーは、Lucene インデックスに関連付けられます (複数のエンティティーが同じインデックスを共有する場合を除く)。インデックスの名前は、**@Indexed** アノテーションの **index** プロパティーにより提供されます。**index** プロパティーが指定されない場合は、インデックスクラスの完全修飾名が名前として使用されます (推奨)。

DirectoryProvider と追加のオプションは、接頭辞 **hibernate.search.<indexname>** を使用して設定できます。名前 **default** (**hibernate.search.default**) は予約され、すべてのインデックスに適用するプロパティーを定義するために使用できます。ディレクトリープロバイダーの設定には、**hibernate.search.default.directory_provider** を使用してデフォルトディレクトリープロバイダーをファイルシステムのものにどのように設定するかが示されています。**hibernate.search.default.indexBase** により、インデックス用のデフォルトベースディレクトリーが設定されます。結果として、エンティティー **Status** のインデックスが **/usr/lucene/indexes/org.hibernate.example.Status** に作成されます。

ただし、**Rule** エンティティーのインデックスはメモリ内ディレクトリーを使用します。これは、このエンティティーのデフォルトディレクトリープロバイダーがプロパティー **hibernate.search.Rules.directory_provider** によりオーバーライドされるためです。

最後に、**Action** エンティティーは、**hibernate.search.Actions.directory_provider** 経由で指定されたカスタムディレクトリープロバイダー **CustomDirectoryProvider** を使用します。

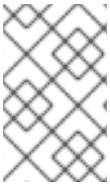
インデックス名の指定

```
package org.hibernate.example;
```

```
@Indexed
public class Status { ... }

@Indexed(index="Rules")
public class Rule { ... }

@Indexed(index="Actions")
public class Action { ... }
```

ディレクトリープロバイダーの設定

```
hibernate.search.default.directory_provider = filesystem
hibernate.search.default.indexBase=/usr/lucene/indexes
hibernate.search.Rules.directory_provider = ram
hibernate.search.Actions.directory_provider =
com.acme.hibernate.CustomDirectoryProvider
```



注記

定義された設定スキームを使用して、ディレクトリープロバイダーやベースディレクトリーなどの共通のルールを簡単に定義したり、これらのデフォルト値をインデックスごとに後でオーバーライドしたりできます。

ディレクトリープロバイダーおよびプロパティー

ram

なし

filesystem

ファイルシステムベースのディレクトリー。使用されるディレクトリーは <indexBase>/< indexName >

- **indexBase** : ベースディレクトリー
- **indexName**: @Indexed.index をオーバーライドする (共有インデックスの場合に有用)
- **locking_strategy** : オプション (LockFactory 設定を参照)
- **filesystem_access_type**: この **DirectoryProvider** により使用される **FSDirectory** 実装のタイプを決定できます。許可される値は、**auto** (デフォルト値。Windows 以外のシステムでは **NIOFSDirectory**、Windows では **SimpleFSDirectory** を選択)、**simple (SimpleFSDirectory)**、**nio (NIOFSDirectory)**、**mmap (MMapDirectory)** です。この設定を変更する前にこれらの Directory 実装の Javadocs を参照してください。**NIOFSDirectory** または **MMapDirectory** を使用するとパフォーマンスが大幅に向上することがありますが、問題も発生します。

filesystem-master

ファイルシステムベースのディレクトリー (**filesystem** など)。通常は、インデックスもソースディレクトリー (コピーディレクトリー) にコピーされます。
更新期間の推奨値は、情報をコピーする時間よりも 50% (以上) 大きい値です (デフォルトでは 3600 秒 - 60 分)。

コピーは増分コピーメカニズムに基づき、平均コピー時間が短縮されることに注意してください。

DirectoryProvider は、通常 JMS バックエンドクラスターのマスターノートで使用されます。

buffer_size_on_copy は、オペレーティングシステムと利用可能な RAM に依存します。最良の結果を得るには、16〜64MB の値を使用することが推奨されます。

- **indexBase**: ベースディレクトリー
- **indexName**: @Indexed.index をオーバーライドする (共有インデックスの場合に有用)
- **sourceBase**: ソース (コピー) ベースディレクトリー
- **source**: ソースディレクトリー接尾辞 (デフォルト値は **@Indexed.index**)。実際のソースディレクトリー名は **<sourceBase>/<source>**
- **refresh**: 秒単位の更新期間 (コピーは **refresh** 秒ごとに実行されます)。以下の **refresh** 期間が経過した時にまだコピーが実行中である場合は、2 番目のコピー操作が省略されます。
- **buffer_size_on_copy**: 単一の低レベルコピー命令で移動する MegaBytes のサイズ。デフォルト値は 16MB です。
- **locking_strategy** : オプション (LockFactory 設定を参照)
- **filesystem_access_type**: この **DirectoryProvider** により使用される **FSDirectory** 実装のタイプを決定できます。許可される値は、**auto** (デフォルト値。Windows 以外のシステムでは **NIOFSDirectory**、Windows では **SimpleFSDirectory** を選択)、**simple (SimpleFSDirectory)**、**nio (NIOFSDirectory)**、**mmap (MMapDirectory)** です。この設定を変更する前にこれらの Directory 実装の Javadocs を参照してください。**NIOFSDirectory** または **MMapDirectory** を使用するとパフォーマンスが大幅に向上することがありますが、問題も発生します。

filesystem-slave

ファイルシステムベースのディレクトリー (**filesystem** など)。通常は、マスターバージョン (ソース) が取得されます。ロックや不整合な検索結果を避けるために、2 つのローカルコピーが保持されます。
更新期間の推奨値は、情報をコピーする時間よりも 50% (以上) 大きい値です (デフォルトでは 3600 秒 - 60 分)。

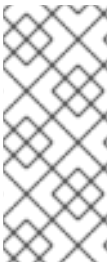
コピーは増分コピーメカニズムに基づき、平均コピー時間が短縮されることに注意してください。**refresh** 期間が経過した時にまだコピーが実行中である場合は、2 番目のコピー操作が省略されます。

JMS バックエンドを使用してスレーブノードで一般的に使用される DirectoryProvider。

buffer_size_on_copy は、オペレーティングシステムと利用可能な RAM に依存します。最良の結果を得るには、16〜64MB の値を使用することが推奨されます。

- **indexBase**: ベースディレクトリー
- **indexName**: @Indexed.index をオーバーライドする (共有インデックスの場合に有用)
- **sourceBase**: ソース (コピー) ベースディレクトリー
- **source**: ソースディレクトリー接尾辞 (デフォルト値は **@Indexed.index**)。実際のソースディレクトリー名は **<sourceBase>/<source>**
- **refresh**: 秒単位の更新期間 (コピーは refresh 秒ごとに実行されます)。

- **buffer_size_on_copy**: 単一の低レベルコピー命令で移動する MegaBytes のサイズ。デフォルト値は 16MB です。
- **locking_strategy** : オプション (LockFactory 設定を参照)
- **retry_marker_lookup**: オプションであり、デフォルト値は 0 です。Hibernate Search がソースディレクトリーのマーカーファイルをチェックする回数を定義します。試行の間隔は 5 秒です。
- **retry_initialize_period**: オプション。再試行初期化機能を有効にするために秒を整数値で設定します。スレーブがマスターインデックスを見つけることができない場合は、見つかるまでバックグラウンドで再試行され、アプリケーションの起動は阻止されません。インデックスが初期化される前に実行された fullText クエリーはブロックされませんが、空の結果を返します。オプションを有効にしない場合や、明示的にゼロに設定しない場合は、再試行タイマーのスケジュールの代わりに、失敗して例外が発生します。無効なインデックスなしでアプリケーションが起動しないようにし、初期化タイムアウトを制御する場合は、代わりに **retry_marker_lookup** を参照してください。
- **filesystem_access_type**: この **DirectoryProvider** により使用される **FSDirectory** 実装のタイプを決定できます。許可される値は、auto (デフォルト値。Windows 以外のシステムでは **NIOFSDirectory**、Windows では **SimpleFSDirectory** を選択)、**simple (SimpleFSDirectory)**、**nio (NIOFSDirectory)**、**mmap (MMapDirectory)** です。この設定を変更する前にこれらの Directory 実装の Javadocs を参照してください。**NIOFSDirectory** または **MMapDirectory** を使用するとパフォーマンスが大幅に向上することがありますが、問題も発生します。



注記

組み込みのディレクトリープロバイダーがニーズを満たさない場合は、**org.hibernate.store.DirectoryProvider** インターフェースを実装して独自のディレクトリープロバイダーを記述できます。この場合は、プロバイダーの完全修飾クラス名を **directory_provider** プロパティーに渡します。追加のプロパティーは、接頭辞 **hibernate.search.<indexname>** を使用して渡すことができます。

13.2.4. ワーカー設定

ワーカー設定を介して Hibernate Search がどのように Lucene と対話するかを定義できます。複数のアーキテクチャーコンポーネントと拡張が存在します。

ワーカー設定を使用して Infinispan クエリーが Lucene と対話する方法を定義します。この設定では、複数のアーキテクチャーコンポーネントと拡張が存在します。

最初に、**Worker** について説明します。**Worker** インターフェースの実装は、すべてのエンティティーの変更を受け取り、コンテキスト別にキューに格納し、コンテキストが終了したら適用します。最も直感的なコンテキスト (特に ORM との接続) はトランザクションです。このため、Hibernate Search は、トランザクションごとにすべての変更をスコープ指定するためにデフォルトで **TransactionalWorker** を使用します。ただし、コンテキストがエンティティーの変更や他のアプリケーション (ライフサイクル) イベントの数などに依存するシナリオが考えられます。

表13.1 スコープ設定

プロパティー	説明
hibernate.search.worker.scope	使用する **Worker** 実装の完全修飾クラス名。このプロパティーが設定されていないか、空白または **transaction** である場合は、デフォルトの **TransactionalWorker** が使用されます。
hibernate.search.worker.*	接頭辞が **hibernate.search.worker** であるすべての設定プロパティーは、初期化時にワーカーに渡されます。これにより、カスタムのワーカー固有パラメーターを追加できるようになります。
hibernate.search.worker.batch_size	コンテキストごとにバッチ化されたインデックス作成操作の最大数を定義します。この制限に到達すると、コンテキストがまだ終了していなくてもインデックス作成がトリガーされます。このプロパティーは、**Worker** 実装がキューに格納されたワークを BatchedQueueingProcessor に委任する場合のみ機能します (これは **TransactionalWorker** が行う操作です)。

コンテキストが終了すると、インデックスの変更を準備し、適用します。これは、新しいスレッド内で同期的または非同期的に行えます。同期のアップデートの利点は、インデックスが常にデータベースと同期されることです。その一方で、非同期アップデートでは、ユーザー応答時間を最小化できます。非同期アップデートの欠点は、データベースとインデックスの状態が一致しない場合があることです。

注記

以下のオプションはインデックスごとに異なることがあります。実際には、indexName 接頭辞または **default** を使用してすべてのインデックスのデフォルト値を設定します。

表13.2 実行設定

プロパティー	説明
hibernate.search.<indexName>.worker.execution	**sync**: 同期実行 (デフォルト) **async**: 非同期実行
hibernate.search.<indexName>.worker.thread_pool.size	バックエンドは、スレッドプールを使用して、同じトランザクションコンテキスト (またはバッチ) からアップデートを同時に適用できます。デフォルト値は 1 です。トランザクションごとに多くの操作がある場合は、大きい値を試行できます。
hibernate.search.<indexName>.worker.buffer_queue.max	スレッドプールが枯渇した場合のワークキューの最大数を定義します。非同期実行の場合のみ役に立ちます。デフォルト値は無制限です。制限に達すると、メインスレッドによりワークが実行されます。

これまで、どの実行モードであるかに関係なく、すべてのワークは同じ仮想マシン (VM) 内で実行されました。1 つの VM に対してワークの合計量は変わりませんでした。問題に対処するために、委任という優れた方法があります。**hibernate.search.default.worker.backend** を設定することにより、インデックス作成ワークを別のサーバーに送信できます。繰り返しますが、このオプションは、各インデックスに対して個別に設定できます。

表13.3 バックエンド設定

プロパティー	説明
hibernate.search.<indexName>.worker.backend	**lucene**: 同じ VM でインデックスのアップデートを実行するデフォルトバックエンド。プロパティーが未定義または空白の場合にも使用されます。 **jms**: JMS バックエンド。インデックスアップデートは、インデックス作成マスターにより処理される JMS キューに送信されます。追加の設定オプションとこのセットアップの詳細については、JMS バックエンド設定を参照してください。 **blackhole**: すべてのインデックス作成ワークを無視する主にテスト / 開発者向けの設定 また、**BackendQueueProcessor** を実装するクラスの完全修飾名を指定することもできます。これにより、ユーザーは独自の通信レイヤーを実装できます。実装は、実行時インデックスワークを処理する **Runnable** を返します。

表13.4 JMS バックエンド設定

プロパティー	説明
hibernate.search.<indexName>.worker.jndi.*	InitialContext を初期化するために JNDI プロパティーを定義します (必要な場合)。JNDI は JMS バックエンドによってのみ使用されます。
hibernate.search.<indexName>.worker.jms.connection_factory	JMS バックエンドに必要です。JMS 接続ファクトリーをルックアップする JNDI 名を定義します (Red Hat JBoss Enterprise Application Platform ではデフォルトで **/ConnectionFactory**)。
hibernate.search.<indexName>.worker.jms.queue	JMS バックエンドに必要です。JMS キューをルックアップする JNDI 名を定義します。キューはワークメッセージをポストするために使用されます。

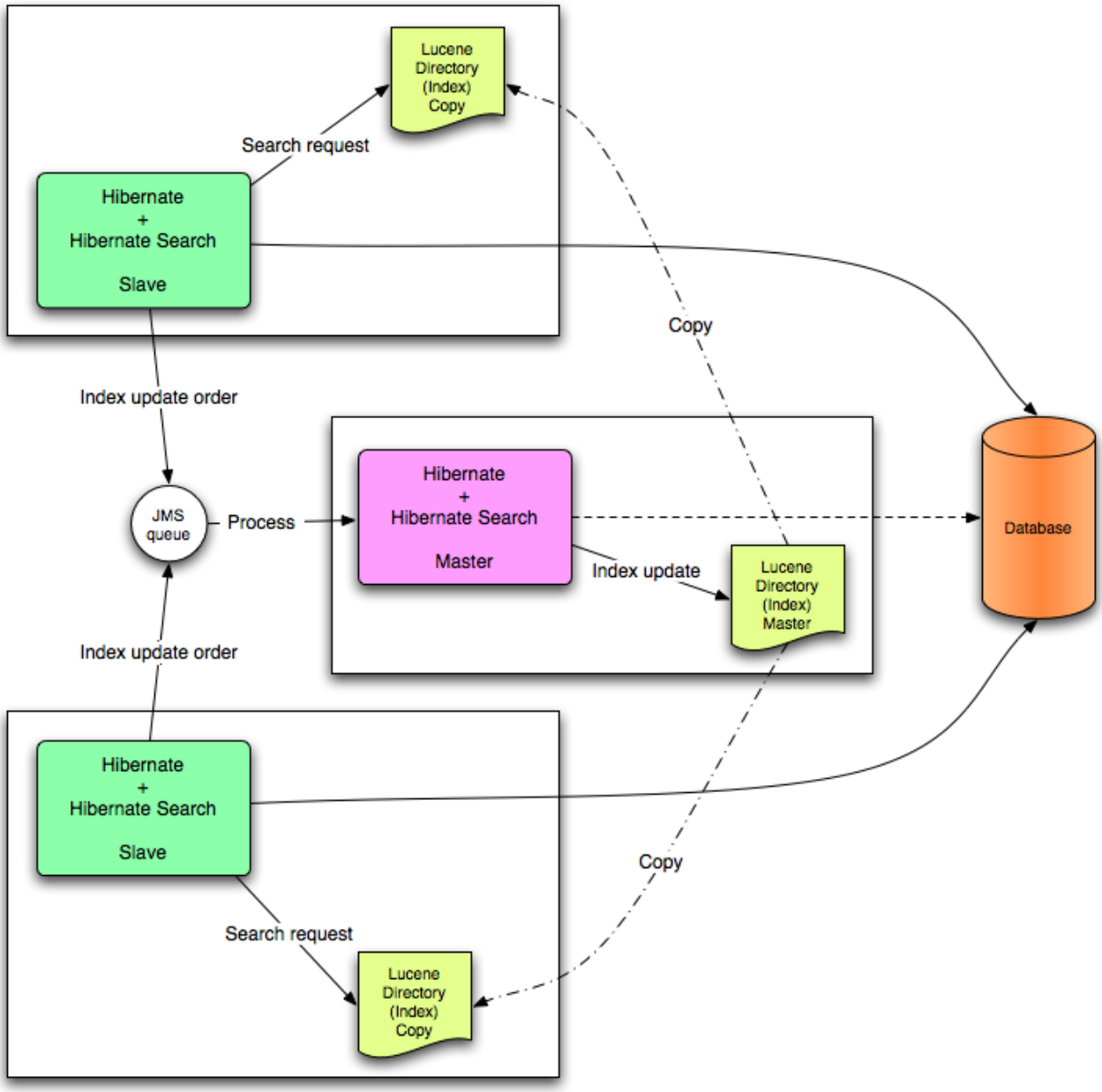
警告

お気づきになったかもしれませんが、示された一部のプロパティーは相互に影響を及ぼします。つまり、すべてのプロパティー値の組み合わせが有効であるとは限りません。実際には、機能しない設定を使用することがあります。これは、特に、示された一部のインターフェースの独自の実装を提供する場合に当てはまります。独自の `Worker` または `BackendQueueProcessor` 実装を記述する前に既存のコードを調べてください。

13.2.4.1. JMS マスター/スレーブバックエンド

本セクションでは、マスター/スレーブ Hibernate Search アーキテクチャーの設定方法について詳しく説明します。

図13.3 JMS バックエンド設定



13.2.4.2. スレーブノード

各インデックスアップデート操作は、JMS キューに送信されます。インデックスクエリー操作はローカルインデックスコピーで実行されます。

JMS スレーブ設定

```
### slave configuration

## DirectoryProvider
# (remote) master location
hibernate.search.default.sourceBase =
/mnt/mastervolume/lucenedirs/mastercopy

# local copy location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider = filesystem-slave

## Back-end configuration
hibernate.search.default.worker.backend = jms
hibernate.search.default.worker.jms.connection_factory =
/ConnectionFactory
hibernate.search.default.worker.jms.queue = queue/hibernatesearch
#optional jndi configuration (check your JMS provider for more information)

## Optional asynchronous execution strategy
# hibernate.search.default.worker.execution = async
# hibernate.search.default.worker.thread_pool.size = 2
# hibernate.search.default.worker.buffer_queue.max = 50
```

注記

検索結果を高速化するには、ファイルシステムローカルコピーが推奨されます。

13.2.4.3. マスターノード

各インデックスアップデート操作は、JMS キューから取得され、実行されます。マスターインデックスは定期的にコピーされます。

JMS キュー内のインデックスアップデート操作が実行され、マスターインデックスが定期的にコピーされます。

JMS マスター設定

```
### master configuration

## DirectoryProvider
# (remote) master location where information is copied to
hibernate.search.default.sourceBase =
/mnt/mastervolume/lucenedirs/mastercopy
```

```
# local master location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider = filesystem-master

## Back-end configuration
#Back-end is the default for Lucene
```

JMS からインデックスワークキューを処理するために、Hibernate Search フレームワーク設定以外に、メッセージ駆動 Bean を記述して、セットアップする必要があります。

インデックス作成キューを処理するメッセージ駆動 Bean

```
@MessageDriven(activationConfig = {
      @ActivationConfigProperty(propertyName="destinationType",
                                propertyValue="javax.jms.Queue"),
      @ActivationConfigProperty(propertyName="destination",
                                propertyValue="queue/hibernatesearch"),
      @ActivationConfigProperty(propertyName="DLQMaxResent",
propertyValue="1")
   } )
public class MDBSearchController extends
AbstractJMSHibernateSearchController
                                 implements MessageListener {
    @PersistenceContext EntityManager em;

    //method retrieving the appropriate session
    protected Session getSession() {
        return (Session) em.getDelegate();
    }

    //potentially close the session opened in #getSession(), not needed
here
    protected void cleanSessionIfNeeded(Session session)
    }
}
```

このサンプルは、Hibernate Search ソースコードで利用可能な抽象 JMS コントローラークラスから継承され、JavaEE MDB を実装します。この実装はサンプルとして提供され、Java EE メッセージ駆動 Bean 以外のものを使用するよう調整できます。

13.2.5. Lucene インデックス作成のチューニング

13.2.5.1. Lucene インデックス作成パフォーマンスのチューニング

Hibernate Search では、基礎となる Lucene **`IndexWriter`** に渡される **`mergeFactor`**、**`maxMergeDocs`**、**`maxBufferedDocs`** などのパラメーターセットを指定して、Lucene インデックス作成パフォーマンスをチューニングできます。これらのパラメーターは、インデックスまたはシャードごとにすべてのインデックスに適用するデフォルト値として指定します。

さまざまなユースケースに対してチューニングできる複数の低レベル **IndexWriter** 設定があります。これらのパラメーターは、以下のように **indexwriter** キーワードでグループ化されます。

```
hibernate.search.[default|<indexname>].indexwriter.<parameter_name>
```

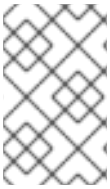
特定のシャード設定の **indexwriter** 値に値が設定されていない場合は、Hibernate Search がインデックスセクションをチェックし、その後でデフォルトセクションをチェックします。

以下の表の設定では、次のように **Animal** インデックスの 2 番目のシャードで適用される設定になります。

- **max_merge_docs** = 10
- **merge_factor** = 20
- **ram_buffer_size** = 64MB
- **term_index_interval** = Lucene default

他のすべての値は Lucene で定義されたデフォルト値を使用します。

すべての値のデフォルトでは、Lucene の独自のデフォルトになります。このため、インデックス作成パフォーマンスと動作プロパティーのリストにリストされた値は使用している Lucene のバージョンによって異なります。示された値はバージョン **2.4** に対して相対的になります。



注記

Hibernate Search の以前のバージョンでは **batch** プロパティーと **transaction** プロパティーが使用されていました。これは、バックエンドが常に同じ設定を使用してワークを実行するため、現在該当しません。

表13.5 インデックス作成パフォーマンスと動作プロパティーのリスト

プロパティー	説明	デフォルト値
hibernate.search.[default|<indexname>].exclusive_index_use	同じインデックスに書き込む必要がある他のプロセスがない場合は、**true** に設定されます。これにより、Hibernate Search がインデックスについて排他モードで動作することが可能になり、インデックスへ変更を書き込むときにパフォーマンスが向上します。	**true** (パフォーマンスが向上し、シャットダウン時にのみロックがリリースされる)
hibernate.search.[default|<indexname>].max_queue_length	各インデックスには、インデックスに適用するアップデートを含む「パイプライン」が含まれます。このキューがいっぱいである場合は、キューに操作を追加すると、ブロック操作になります。**worker.execution** が **async** として設定されていない限り、この設定の指定にはあまり意味がありません。	**1000**

プロパティー	説明	デフォルト値
hibernate.search.[default|<indexname>].indexwriter.max_buffered_delete_terms	バッファされたメモリ内削除用語を適用およびフラッシュする前に必要な削除用語の最小数を決定します。メモリ内にバッファされたドキュメントがある場合、ドキュメントはマージされ、新しいセグメントが作成されます。	無効 (RAM 使用によりフラッシュ)
hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs	インデックス作成中にメモリ内にバッファされるドキュメントの量を制御します。値が大きくなると、より多くの RAM が消費されます。	無効 (RAM 使用によりフラッシュ)
hibernate.search.[default|<indexname>].indexwriter.max_merge_docs	セグメントで許可されたドキュメントの最大数を定義します。頻繁にインデックスが変わる場合は、小さい値が適切です。インデックスが頻繁に変わらない場合は、値が大きいと、検索パフォーマンスが向上します。	無制限 (Integer.MAX_VALUE)
hibernate.search.[default|<indexname>].indexwriter.merge_factor	セグメントマージの頻度とサイズを制御します。 挿入の実行時にセグメントインデックスをマージする頻度を決定します。値が小さいと、インデックス作成中の RAM の使用量が小さくなり、最適化されていないインデックスでの検索が高速になりますが、インデックス作成時間が長くなります。値が大きいと、インデックス作成中の RAM の使用量が大きくなり、最適化されていないインデックスでの検索が低速になりますが、インデックス作成時間は短くなります。したがって、大きい値 (> 10) はバッチインデックス作成に最適であり、小さい値 (< 10) は対話的に維持するインデックスに最適です。値は 2 よりも小さくしないでください。	10
hibernate.search.[default|<indexname>].indexwriter.merge_min_size	セグメントマージの頻度とサイズを制御します。次のセグメントマージ操作でこのサイズ (MB 単位) よりも小さいセグメントが常に考慮されます。この値が大きすぎると、頻度が少ない場合であってもマージ操作のコストが高くなることがあります。**org.apache.lucene.index.LogDocMergePolicy.minMergeSize** も参照してください。	0 MB (実際には 1K 以下)

プロパティー	説明	デフォルト値
hibernate.search.[default|<indexname>].indexwriter.merge_max_size	セグメントマージの頻度とサイズを制御します。 このサイズ (MB 単位) よりも大きいセグメントは大きいセグメントでマージされません。 これにより、メモリーの要件が少なくなり、最適な検索速度が失われる一部のマージ操作を回避できます。インデックスの最適化時に、この値は無視されます。 **org.apache.lucene.index.LogDocMergePolicy.maxMergeSize** も参照してください。	無制限
hibernate.search.[default|<indexname>].indexwriter.merge_max_optimize_size	セグメントマージの頻度とサイズを制御します。 インデックスの最適化時でも、このサイズ (MB 単位) よりも大きいセグメントは大きいセグメントでマージされません (**merge_max_size** 設定も参照)。 **org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptimize** に適用されます。	無制限
hibernate.search.[default|<indexname>].indexwriter.merge_calibrate_by_deletes	セグメントマージの頻度とサイズを制御します。 マージポリシーを推測する場合に削除されたドキュメントを考慮しない場合は、**false** に設定されます。 **org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes** に適用されます。	**true**
hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size	ドキュメントバッファー専用の RAM の量 (MB 単位) を制御します。max_buffered_docs とともに使用すると、最初に発生したイベントに対してフラッシュが実行されます。 一般的に、インデックス作成パフォーマンスを向上させるには、ドキュメント数の代わりに RAM 使用量によりフラッシュし、できるだけ大きい RAM バッファーを使用することが推奨されます。	16 MB

プロパティー	説明	デフォルト値
hibernate.search.[default|<indexname>].indexwriter.term_index_interval	エキスパート: インデックス用語間の間隔を設定します。 値が大きいと、IndexReader が使用するメモリーが少なくなりますが、用語へのランダムアクセスは遅くなります。値が小さいと、IndexReader が使用するメモリーが多くなり、用語へのランダムアクセスが速くなります。詳細については、Lucene ドキュメンテーションを参照してください。	128
hibernate.search.[default|<indexname>].indexwriter.use_compound_file	複合ファイル形式を使用する利点は、使用するファイル記述子が少なくなることです。欠点は、インデックスの作成時間が長くなり、一時ディスク容量が大きくなることです。インデックス作成時間を短縮するためにこのパラメーターを **false** に設定できますが、**mergeFactor** も大きい場合はファイル記述子が足りなくなることがあります。 ブール値パラメーター。“true” または “false” を使用します。このオプションのデフォルト値は **true** です。	true
hibernate.search.enable_dirty_check	すべてのエンティティーの変更で Lucene インデックスのアップデートが必要になるわけではありません。アップデートされたすべてのエンティティープロパティー (ダーティープロパティー) のインデックスが作成されていない場合は、Hibernate Search によりインデックスの再作成プロセスが省略されます。 各アップデートイベントで呼び出す必要があるカスタム **FieldBridges** を使用する場合は、このオプションを無効にします (フィールドブリッジが設定されたプロパティーが変更されない場合であっても)。 この最適化は、**@ClassBridge** または **@DynamicBoost** を使用したクラスに適用されません。 ブール値パラメーター。“true” または “false” を使用します。このオプションのデフォルト値は **true** です。	true

警告

blackhole バックエンドは、本番稼働で使用することを目的としません。インデックス作成のボトルネックを特定するツールとしてのみ使用してください。

13.2.5.2. Lucene IndexWriter

さまざまなユースケースに対してチューニングできる複数の低レベル **IndexWriter** 設定があります。これらのパラメーターは、以下のように **indexwriter** キーワードでグループ化されます。

```
default.<indexname>.indexwriter.<parameter_name>
```

シャード設定の **indexwriter** に値が設定されていない場合は、Hibernate Search がインデックスセクションをチェックし、その後でデフォルトセクションをチェックします。

13.2.5.3. パフォーマンスオプション設定

以下の設定では、次のように **Animal** インデックスの 2 番目のシャードで適用される設定になります。

パフォーマンスオプション設定例

```
default.Animals.2.indexwriter.max_merge_docs = 10
default.Animals.2.indexwriter.merge_factor = 20
default.Animals.2.indexwriter.term_index_interval = default
default.indexwriter.max_merge_docs = 100
default.indexwriter.ram_buffer_size = 64
```

- **max_merge_docs** = 10
- **merge_factor** = 20
- **ram_buffer_size** = 64MB
- **term_index_interval** = Lucene default

他のすべての値は Lucene で定義されたデフォルト値を使用します。

Lucene デフォルト値は、Hibernate Search のデフォルト設定です。したがって、以下の表にリストされた値は使用している Lucene のバージョンによって異なります。示された値はバージョン **2.4** に対して相対的になります。Lucene インデックス作成パフォーマンスの詳細については、Lucene ドキュメンテーションを参照してください。

注記

バックエンドでは常に同じ設定を使用してワークが実行されます。

表13.6 インデックス作成パフォーマンスと動作プロパティーのリスト

プロパティー	説明	デフォルト値
default.<indexname>.exclusive_index_use	同じインデックスに書き込む必要がある他のプロセスがない場合は、**true** に設定されます。これにより、Hibernate Search がインデックスについて排他モードで動作することが可能になり、インデックスへ変更を書き込むときにパフォーマンスが向上します。	**true** (パフォーマンスが向上し、シャットダウン時にのみロックがリリースされる)

プロパティー	説明	デフォルト値
default.<indexname>.max_queue_length	各インデックスには、インデックスに適用するアップデートを含む「パイプライン」が含まれます。このキューがいっぱいである場合は、キューに操作を追加すると、ブロック操作になります。**worker.execution** が **async** として設定されていない限り、この設定の指定にはあまり意味がありません。	**1000**
default.<indexname>.indexwriter.max_buffered_delete_terms	バッファされたメモリ内削除用語を適用およびフラッシュする前に必要な削除用語の最小数を決定します。メモリ内にバッファされたドキュメントがある場合、ドキュメントはマージされ、新しいセグメントが作成されます。	無効 (RAM 使用によりフラッシュ)
default.<indexname>.indexwriter.max_buffered_docs	インデックス作成中にメモリ内にバッファされるドキュメントの量を制御します。値が大きくなると、より多くの RAM が消費されます。	無効 (RAM 使用によりフラッシュ)
default.<indexname>.indexwriter.max_merge_docs	セグメントで許可されたドキュメントの最大数を定義します。頻繁にインデックスが変わる場合は、小さい値が適切です。インデックスが頻繁に変わらない場合は、値が大きいと、検索パフォーマンスが向上します。	無制限 (Integer.MAX_VALUE)
default.<indexname>.indexwriter.merge_factor	セグメントマージの頻度とサイズを制御します。 挿入の実行時にセグメントインデックスをマージする頻度を決定します。値が小さいと、インデックス作成中の RAM の使用量が小さくなり、最適化されていないインデックスでの検索が高速になりますが、インデックス作成時間が長くなります。値が大きいと、インデックス作成中の RAM の使用量が大きくなり、最適化されていないインデックスでの検索が低速になりますが、インデックス作成時間は短くなります。したがって、大きい値 (> 10) はバッチインデックス作成に最適であり、小さい値 (< 10) は対話的に維持するインデックスに最適です。値は 2 よりも小さくしないでください。	10

プロパティー	説明	デフォルト値
default.<indexname>.indexwriter.merge_min_size	セグメントマージの頻度とサイズを制御します。 次のセグメントマージ操作でこのサイズ (MB 単位) よりも小さいセグメントが常に考慮されます。 この設定値が大きすぎると、コストがかかるマージ操作になる場合があります (頻度は少なくなります)。 **org.apache.lucene.index.LogDocMergePolicy.minMergeSize** も参照してください。	0 MB (実際には 1K 以下)
default.<indexname>.indexwriter.merge_max_size	セグメントマージの頻度とサイズを制御します。 このサイズ (MB 単位) よりも大きいセグメントは大きいセグメントでマージされません。 これにより、メモリーの要件が少なくなり、最適な検索速度が失われる一部のマージ操作を回避できます。インデックスの最適化時に、この値は無視されます。 **org.apache.lucene.index.LogDocMergePolicy.maxMergeSize** も参照してください。	無制限
default.<indexname>.indexwriter.merge_max_optimize_size	セグメントマージの頻度とサイズを制御します。 インデックスの最適化時でも、このサイズ (MB 単位) よりも大きいセグメントは大きいセグメントでマージされません (**merge_max_size** 設定も参照)。 **org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptimize** に適用されます。	無制限
default.<indexname>.indexwriter.merge_calibrate_by_deletes	セグメントマージの頻度とサイズを制御します。 マージポリシーを推測する場合に削除されたドキュメントを考慮しない場合は、**false** に設定されます。 **org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes** に適用されます。	**true**

プロパティー	説明	デフォルト値
default.<indexname>.indexwriter.ram_buffer_size	ドキュメントバッファー専用の RAM の量 (MB 単位) を制御します。max_buffered_docs とともに使用すると、最初に発生したイベントに対してフラッシュが実行されます。 一般的に、インデックス作成パフォーマンスを向上させるには、ドキュメント数の代わりに RAM 使用量によりフラッシュし、できるだけ大きい RAM バッファーを使用することが推奨されます。	16 MB
default.<indexname>.indexwriter.term_index_interval	エキスパート: インデックス用語間の間隔を設定します。 値が大きいと、IndexReader が使用するメモリーが少なくなりますが、用語へのランダムアクセスは遅くなります。値が小さいと、IndexReader が使用するメモリーが多くなり、用語へのランダムアクセスが速くなります。詳細については、Lucene ドキュメンテーションを参照してください。	128
default.<indexname>.indexwriter.use_compound_file	複合ファイル形式を使用する利点は、使用するファイル記述子が少なくなることです。欠点は、インデックスの作成時間が長くなり、一時ディスク容量が大きくなることです。インデックス作成時間を短縮するためにこのパラメーターを **false** に設定できますが、**mergeFactor** も大きい場合はファイル記述子が足りなくなることがあります。 ブール値パラメーター。“true” または “false” を使用します。このオプションのデフォルト値は **true** です。	true
default.enable_dirty_check	すべてのエンティティーの変更で Lucene インデックスのアップデートが必要になるわけではありません。アップデートされたすべてのエンティティープロパティー (ダーティープロパティー) のインデックスが作成されていない場合は、Hibernate Search によりインデックスの再作成プロセスが省略されます。 各アップデートイベントで呼び出す必要があるカスタム **FieldBridges** を使用する場合は、このオプションを無効にします (フィールドブリッジが設定されたプロパティーが変更されない場合であっても)。 この最適化は、**@ClassBridge** または **@DynamicBoost** を使用したクラスに適用されません。 ブール値パラメーター。“true” または “false” を使用します。このオプションのデフォルト値は **true** です。	true

13.2.5.4. インデックス作成速度のチューニング

アーキテクチャーで許可される場合は、インデックス作成の効率を向上させるために **default.exclusive_index_use=true** を保持します。

インデックス作成速度をチューニングする場合は、最初にオブジェクトのロードを最適化し、インデックス作成プロセスをチューニングするベースラインとして実現するタイミングを使用することが推奨されます。ワーカーバックエンドとして **blackhole** を設定し、インデックス作成ルーチンを開始します。このバックエンドは Hibernate Search を無効にしません。インデックスに対して必要な変更セットが生成されますが、それらの変更セットはインデックスに対してフラッシュされるのではなく破棄されます。**hibernate.search.indexing_strategy** を **manual** に設定するのとは異なり、**blackhole** を使用すると、データベースからより多くのデータがロードされることがあります (関連付けられたエンティティーのインデックスが再び作成されるため)。

```
hibernate.search.[default|<indexname>].worker.backend blackhole
```

警告

blackhole バックエンドは、本番稼働で使用することを目的としません。インデックス作成のボトルネックを特定する診断ツールとしてのみ使用してください。

13.2.5.5. セグメントサイズの制御

以下のオプションを使用すると、作成するセグメントの最大サイズを設定できます。

- **merge_max_size**
- **merge_max_optimize_size**
- **merge_calibrate_by_deletes**

セグメントサイズの制御

```
//to be fairly confident no files grow above 15MB, use:
hibernate.search.default.indexwriter.ram_buffer_size = 10
hibernate.search.default.indexwriter.merge_max_optimize_size = 7
hibernate.search.default.indexwriter.merge_max_size = 7
```

マージ操作に対して **max_size** をハード制限セグメントサイズの半分未満に設定します (セグメントのマージでは 2 つのセグメントが結合されるため)。

新しいセグメントは最初に期待したものよりも大きい場合がありますが、作成されるセグメントは **ram_buffer_size** よりも大幅に大きくなりません。このしきい値は推定値としてチェックされます。

13.2.6. LockFactory 設定

Lucene ディレクトリーは、Hibernate Search で管理された各インデックス用の **LockingFactory** 経由でカスタムロックストラテジーを使用して設定できます。

一部のロックストラテジーは、ファイルシステムレベルのロックを必要とし、RAM ベースのインデックスで使用できます。このストラテジーを使用する場合は、ロックマーカーファイルを格納するファイルシステムの場所を参照するよう **IndexBase** 設定オプションを指定する必要があります。

ロックファクトリーを選択するには、**hibernate.search.<index>.locking_strategy** オプションを以下のいずれかのオプションに設定します。

- **simple**
- **native**
- **single**
- **none**

表13.7 利用可能な LockFactory 実装のリスト

名前	クラス	説明
LockFactory 設定 **simple**	org.apache.lucene.store.SimpleFSLockFactory	Java のファイル API に基づいた安全な実装。マーカーファイルを作成することによりインデックスの使用をマークします。 何らかの理由でアプリケーションを終了する必要がある場合はアプリケーションを再び起動する前にこのファイルを削除する必要があります。
native	org.apache.lucene.store.NativeFSLockFactory	**simple** と同様に、マーカーファイルを作成することによりインデックスの使用をマークしますが、JVM が終了した場合であってもロックがクリーンアップされるようにネイティブの OS ファイルロックを使用します。 この実装には NFS の既知の問題があります。ネットワーク共有で使用しないでください。 **native** は、**filesystem**、**filesystem-master**、および **filesystem-slave** ディレクトリープロバイダーのデフォルトの実装です。
single	org.apache.lucene.store.SingleInstanceLockFactory	この LockFactory はファイルマーカーを使用しませんが、メモリーに保持される Java オブジェクトロックです。したがって、これは、インデックスが他のプロセスで共有されない場合のみ使用できます。 これは、**ram** ディレクトリープロバイダーのデフォルト実装です。
none	org.apache.lucene.store.NoLockFactory	このインデックスへの変更はロックにより調整されません。

ロックストラテジー設定の例は以下のとおりです。

```
hibernate.search.default.locking_strategy = simple
```

```
hibernate.search.Animals.locking_strategy = native
hibernate.search.Books.locking_strategy =
org.custom.components.MyLockingFactory
```

13.2.7. インデックス形式の互換性

Hibernate Search は、アプリケーションを新しいバージョンに移植するための後方互換性がある API またはツールを現在提供していません。API はインデックスの書き込みと検索に Apache Lucene を使用します。場合によっては、インデックス形式のアップデートが必要になることがあります。この場合は、データのインデックスを再び作成する必要があることがあります (Lucene が古い形式を読み取れないとき)。

警告

インデックス形式をアップデートする前にインデックスをバックアップします。

Hibernate Search は、**`hibernate.search.lucene_version`** 設定プロパティーを公開します。このプロパティーによって、Analyzers と他の Lucene クラスが古いバージョンの Lucene で定義された動作に準拠するよう指示されます。**`lucene-core.jar`** に含まれる **`org.apache.lucene.util.Version`** も参照してください。オプションが指定されていない場合は、Hibernate Search によって、Lucene がデフォルトのバージョンを使用するよう指示されます。アップグレードの実行時に変更が自動的に行われないように、使用するバージョンを設定で明示的に定義することが推奨されます。アップグレード後に、必要に応じて設定値を明示的に更新できます。

Analyzers と Lucene 3.0 で作成されたインデックスとの互換性を維持する

```
hibernate.search.lucene_version = LUCENE_30
```

設定された **`SearchFactory`** はグローバルであり、該当するパラメーターを含むすべての Lucene API が影響を受けます。Lucene が使用され、Hibernate Search が省略された場合は、整合的な結果を得るために同じ値を適用してください。

13.3. アプリケーション用 HIBERNATE SEARCH

13.3.1. Hibernate Seach の最初のステップ

アプリケーションに対して Hiberate 検索を初めて使用する場合は、以下のトピックを参照してください。

- Maven を使用した Hibernate Seach の有効化
- インデックス化
- 検索
- アナライザー

13.3.2. Maven を使用した Hibernate Seach の有効化

Maven プロジェクトで以下の設定を使用して、**hibernate-search-orm** 依存関係を追加します。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-search-orm</artifactId>
      <version>5.5.1.Final-redhat-1</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

13.3.3. アノテーションの追加

本項では、本の詳細が含まれるデータベースを保有していると仮定します。アプリケーションには Hibernate によって管理される **example.Book** および **example.Author** クラスが含まれ、アプリケーションにフリーテキスト検索機能を追加して本の検索を有効にします。

例: Hibernate Search 固有のアノテーションを追加する前の Book および Author エンティティー

```
package example;
...
@Entity
public class Book {

  @Id
  @GeneratedValue
  private Integer id;

  private String title;

  private String subtitle;

  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  private Date publicationDate;

  public Book() {}

  // standard getters/setters follow here
  ...
}
```

```
package example;
...
```

```
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {}

  // standard getters/setters follow here
  ...
}
```

Book および Author クラスにいくつかのアノテーションを追加する必要があります。最初のアノテーション **@Indexed** は Book をインデックス可能とマーク付けします。設計上、Hibernate Search はトークン化されていない ID をインデックスに保存し、指定エンティティーのインデックスが単一になるようにします。**@DocumentId** はこのために使用するプロパティーをマーク付けし、ほとんどの場合でデータベースの主キーと同じになります。**@Id** アノテーションが存在する場合は、**@DocumentId** は任意のアノテーションになります。

次に、検索可能にしたいフィールドをそのようにマーク付けする必要があります。この例では、最初に **title** および **subtitle** の両方に **@Field** アノテーションを付けます。パラメーター **index=Index.YES** はテキストがインデックス化されるようにし、**analyze=Analyze.YES** はデフォルトの Lucene アナライザーを使用してテキストが分析されるようにします。通常、分析とは文を個別の単語にチャンク化し、「**a**」や「the」などの頻繁に使用される単語を潜在的に除外することを意味します。アナライザーの詳細は後で説明します。**@Field** 内に指定する 3 つ目のパラメーターは **store=Store.NO** であり、実際のデータがインデックスで保存されないようにします。このデーターがインデックスで保存されるかどうかは、検索の機能と関係ありません。Lucene の観点から見ると、インデックスが作成された後にデータを保持する必要はありません。データを保存する利点は、射影を用いて読み出しができることです。

射影を使用しない場合、Hibernate Search はクエリーの基準に一致するエンティティーのデータベース識別子を見つけるために、デフォルトどおりに Lucene クエリーを実行し、これらの識別子を使用してデータベースから管理対象オブジェクトを読み出します。射影を使用するかどうかは、ケースバイケースで決定します。デフォルトでは管理対象オブジェクトを返すのに対し、射影はオブジェクトアレイのみを返すため、デフォルトの動作が推奨されます。**index=Index.YES**、**analyze=Analyze.YES**、および **store=Store.NO** は 3 つのパラメーターのデフォルト値であり、省略できます。

次に説明するアノテーションは **@DateBridge** です。このアノテーションは、Hibernate Search の組み込みフィールドブリッジの 1 つです。Lucene インデックスは完全に文字列ベースです。そのため、Hibernate Search はインデックス化されたフィールドのデータ型を文字列に変換し、文字列をインデックス化されたフィールドのデータ型に変換する必要があります。複数の事前定義済みブリッジが提供されますが、その 1 つが java.util.Date を指定の解決方法で String に変換する DateBridge です。詳細については、ブリッジを参照してください。

最後のアノテーションは **@IndexedEmbedded** です。このアノテーションは、関連するエンティティー (**@ManyToMany**、**@*ToOne**、**@Embedded**、および **@ElementCollection**) を所有するエンティティーの一部としてインデックス化するために使用されます。Lucene インデックスドキュメントはオブジェクトの関係を認識しないフラットなデータ構造であるため、これが必要になります。著者名を検索可能にするには、著者名が本の一部としてインデックス化される必要があります。**@IndexedEmbedded** の他にも、インデックスに含めたい関連エンティティーのすべてのフィールドを **@Indexed** でマーク付けする必要があります。詳細については、埋め込みオブジェクトおよび関連付けられたオブジェクトを参照してください。

エンティティーマッピングの詳細については、エンティティーのマッピングを参照してください。

例: Hibernate 検索アノテーションを追加した後のエンティティー

```
package example;
...
@Entity

public class Book {

  @Id
  @GeneratedValue
  private Integer id;


  private String title;


  private String subtitle;

 @Field(index = Index.YES, analyze=Analyze.NO, store = Store.YES)
  @DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;


  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

```
package example;
...
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;


  private String name;

  public Author() {
  }

  // standard getters/setters follow here
  ...
}
```

13.3.4. インデックス化

Hibernate Search は、Hibernate Core から永続化、更新、または削除された各エンティティーを透過的にインデックス化します。しかし、データベースにすでに存在するデータに対して最初の Lucene インデックスを作成する必要があります。上記のプロパティーとアノテーションを追加したら、本の最初のバッチインデックスをトリガーします。これには、以下のコードスニペットの 1 つを使用します。

例: Hibernate セッションを使用したデータのインデックス化

```
FullTextSession fullTextSession =
org.hibernate.search.Search.getFullTextSession(session);
fullTextSession.createIndexer().startAndWait();
```

例: JPA を使用したデータのインデックス化

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
fullTextEntityManager.createIndexer().startAndWait();
```

上記のコードを実行した後、**/var/lucene/indexes/example.Book** 以下に Lucene インデックスが作成されるはずです。Luke でこのインデックスを確認してみてください。Hibernate Search がどのように機能するかがわかるはずです。

13.3.5. 検索

検索を実行するには、Lucene API または Hibernate Search クエリー DSL のいずれかを使用して Lucene クエリーを作成します。クエリーを org.hibernate.Query でラップして、Hibernate API から必要な機能を取得します。以下のコードは、インデックス化されたフィールドに対してクエリーを準備します。このコードを実行すると、Book のリストが返されます。

例: Hibernate Search セッションを使用した検索の作成および実行

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();

// create native Lucene query using the query DSL
// alternatively you can write the Lucene query using the Lucene query
parser
// or the Lucene programmatic API. The Hibernate Search DSL is recommended
though
QueryBuilder qb = fullTextSession.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
  .keyword()
  .onFields("title", "subtitle", "authors.name", "publicationDate")
  .matching("Java rocks!")
  .createQuery();

// wrap Lucene query in a org.hibernate.Query
org.hibernate.Query hibQuery =
    fullTextSession.createFullTextQuery(query, Book.class);

// execute search
List result = hibQuery.list();
```

```
tx.commit();
session.close();
```

例: JPA を使用した検索の作成および実行

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
em.getTransaction().begin();

// create native Lucene query using the query DSL
// alternatively you can write the Lucene query using the Lucene query
parser
// or the Lucene programmatic API. The Hibernate Search DSL is recommended
though
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
  .keyword()
  .onFields("title", "subtitle", "authors.name", "publicationDate")
  .matching("Java rocks!")
  .createQuery();

// wrap Lucene query in a javax.persistence.Query
javax.persistence.Query persistenceQuery =
    fullTextEntityManager.createFullTextQuery(query, Book.class);

// execute search
List result = persistenceQuery.getResultList();

em.getTransaction().commit();
em.close();
```

13.3.6. アナライザー

インデックス化された本のエンティティーの題名が **Refactoring: Improving the Design of Existing Code** であり、**refactor**、**refactors**、**refactored**、および **refactoring** クエリーのヒットが必要であると仮定します。インデックス化と検索を行うときに、言葉のステミングを適用する Lucene のアナライザークラスを選択します。Hibernate Search はアナライザーを設定する方法を複数提供します (詳細については、デフォルトのアナライザーとクラスによるアナライザーを参照)。

- 設定ファイルで **analyzer** プロパティーを設定します。指定されたクラスがデフォルトのアナライザーになります。
- エンティティーレベルで **@Analyzer** アノテーションを設定します。
- フィールドレベルで **@Analyzer** アノテーションを設定します。

完全修飾クラス名または使用するアナライザーを指定するか、**@Analyzer** アノテーションを使用して **@AnalyzerDef** アノテーションによって定義されたアナライザーを確認します。アナライザーを確認する場合はファクトリーを持つ Solr アナライザーフレームワークが使用されます。ファクトリークラスの詳細については、Solr JavaDoc または Solr Wiki (http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters) の対応する項を参照してください。

この例では StandardTokenizerFactory が LowerCaseFilterFactory と SnowballPorterFilterFactory の 2 つのフィルターファクトリーによって使用されます。トークナイザーは句読点とハイフンで言葉を分割しますが、メールアドレスとインターネットのホスト名は分割しません。この操作とその他の一般的な操作には、標準のトークナイザーが適しています。小文字フィルターはトークンのすべての文字を小文字に変換し、snowball フィルターは言語固有のステミングを適用します。

Solr フレームワークを使用する場合は、任意数のフィルターを用いてトークナイザーを使用します。

例: @AnalyzerDef および Solr フレームワークを使用したアナライザーの定義および使用

```
@Indexed
@AnalyzerDef(
   name = "customanalyzer",
   tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
   filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = SnowballPorterFilterFactory.class,
         params = { @Parameter(name = "language", value = "English") })
 })
public class Book implements Serializable {

  @Field
  @Analyzer(definition = "customanalyzer")
  private String title;

  @Field
  @Analyzer(definition = "customanalyzer")
  private String subtitle;

  @IndexedEmbedded
  private Set authors = new HashSet();

  @Field(index = Index.YES, analyze = Analyze.NO, store = Store.YES)
  @DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

@AnalyzerDef を使用してアナライザーを定義した後、@Analyzer を使用してエンティティーおよびプロパティーに適用します。この例では、**customanalyzer** は定義されていますが、エンティティーに適用されていません。アナライザーは **title** および **subtitle** プロパティーのみに適用されます。アナライザーの定義はグローバルです。エンティティーに対してアナライザーを定義し、必要に応じて他のエンティティーの定義を再使用します。

13.4. インデックス構造へのエンティティーのマッピング

13.4.1. エンティティーのマッピング

エンティティーをインデックス化するために必要なメタデータ情報はすべてアノテーションを用いて記述されるため、XML マッピングファイルは必要ありません。基本的な Hibernate 設定には Hibernate

マッピングを使用できますが、Hibernate Search 固有の設定はアノテーションを用いて表現する必要があります。

13.4.1.1. 基本的なマッピング

最初に、エンティティーのマッピングで最も一般的に使用されるアノテーションについて説明します。

Lucene ベースの Query API は、以下の一般的なアノテーションを使用してエンティティーをマップします。

- @Indexed
- @Field
- @NumericField
- @Id

13.4.1.2. @Indexed

最初に、永続クラスをインデックス可能であると宣言する必要があります。これを行うには、クラスに **@Indexed** アノテーションを付けます (**@Indexed**アノテーションが付いていないエンティティーはインデックス化プロセスによって無視されます)。

```
@Entity
@Indexed
public class Essay {
...
}
```

また、任意で @Indexed アノテーションの **index** 属性を指定して、インデックスのデフォルト名を変更することもできます。

13.4.1.3. @Field

エンティティーの各プロパティー (または属性) に対して、インデックス化する方法を記述できます。デフォルト (アノテーションなし) では、プロパティーはインデックス化プロセスで無視されます。

注記

Hibernate Search 5 よりも前のリリースでは、数値フィールドのエンコーディングは **@NumericField** で明示的に要求された場合のみ選択されました。Hibernate Search 5 では、このエンコーディングは数値の種類に応じて自動的に選択されます。数値のエンコーディングを回避するために、**@Field.bridge** または **@FieldBridge** を使用して非数値フィールドブリッジを明示的に指定できます。パッケージ **org.hibernate.search.bridge.builtin** には、数字を文字列としてエンコードする一連のブリッジ (**org.hibernate.search.bridge.builtin.IntegerBridge** など) が含まれます。

@Field はプロパティーをインデックス化されたプロパティーとして宣言します。また、以下の属性を 1 つまたは複数設定することにより、インデックス化プロセスの複数の側面を設定できます。

- **name** : この名前でプロパティーが Lucene Document に保存されます。デフォルト値はプロパティー名 (JavaBeans 慣例に準拠する) になります。

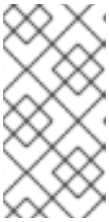
- **store** : プロパティーを Lucene インデックスに保存するかどうかを定義します。値を保存する場合は **Store.YES**を指定します (インデックス領域の消費が増えますが、射影は許可されます)。圧縮して保存する場合は **Store.COMPRESS** を指定します (CPU の消費が増えます)。保存しない場合は **Store.NO** を指定します (デフォルト値です)。プロパティーが保存されると、元の値を Lucene Document から取得できます。これは、要素のインデックス化の有無には関係しません。

- **index**: プロパティーがインデックス化されるかどうかを示します。**Index.NO** を指定するとインデックス化されず、クエリーでは見つかりません。**Index.YES** を選択すると、要素がインデックス化され、検索可能になります。デフォルト値は **Index.YES** です。**Index.NO** は、プロパティーを検索可能にする必要がなく、射影を利用できるようにする場合に便利です。

注記

analyze および **norms** はプロパティーのインデックス化が必要になるため、**Index.NO** を **Analyze.YES** または **Norms.YES** とともに使用しても意味がありません。

- **analyze**: プロパティーが分析されるかどうかを決定します。**Analyze.YES** の場合は分析され、**Analyze.NO** の場合は分析されません。デフォルト値は **Analyze.YES** です。



注記

プロパティーを分析するかどうかは、要素をそのまま検索するか、または要素に含まれる言葉を基に検索するかによって異なります。テキストフィールドを分析することは意味がありますが、データフィールドの分析はほとんど必要ないでしょう。

注記

ソートに使用されるフィールドは分析しないでください。

- **norms**: インデックス時間のブースティング情報を保存するかどうかを示します。保存する場合は **Norms.YES**、保存しない場合は **Norms.NO** を指定します。保存しないと大量のメモリーを節約できますが、インデックス時間のブースティング情報を使用できません。デフォルト値は **Norms.YES** です。

- **termVector**: 単語 (term) と出現頻度 (frequency) のペアを示します。この属性を使用すると、インデックス化中にドキュメント内で term vector を保存できます。デフォルト値は **TermVector.NO** です。
この属性の値は次のとおりです。

値	定義
TermVector.YES	各ドキュメントに term vector を保存します。これにより、同期された 2 つのアレイが生成されます (1 つのアレイには document term が含まれ、もう 1 つのアレイには term の頻度が含まれます)。
TermVector.NO	term vector を保存しません。

値	定義
TermVector.WITH_OFFSETS	term vector およびトークンオフセット情報を保存します。これは、TermVector.YES と同じですが、言葉の開始および終了オフセット位置情報が含まれます。
TermVector.WITH_POSITIONS	term vector およびトークン位置情報を保存します。これは、TermVector.YES と同じですが、ドキュメントで言葉が発生する順序位置 (ordinal position) が含まれます。
TermVector.WITH_POSITION_OFFSETS	term vector、トークン位置、およびオフセット情報を保存します。これは、YES、WITH_OFFSETS、および WITH_POSITIONS の組み合わせです。

- **indexNullAs**：デフォルトでは、null の値は無視されインデックス化されませんが、**indexNullAs** を使用すると **null** の値のトークンとして挿入される文字列を指定できます。デフォルトでは、**null** の値がインデックス化されないことを示す **Field.DO_NOT_INDEX_NULL** に設定されます。**Field.DEFAULT_NULL_TOKEN** に設定すると、デフォルトの **null** トークンが使用されます。設定でデフォルトの **null** トークンを指定するには、**hibernate.search.default_null_token** を使用します。このプロパティーが設定されず、**Field.DEFAULT_NULL_TOKEN** が指定された場合は、デフォルトで文字列「**null**」が使用されます。

注記

indexNullAs パラメーターが使用される場合、検索で同じトークンを使用して **null** の値を検索することが重要になります。また、分析されないフィールド (**Analyze.NO**) のみでこの機能を使用することが推奨されます。

警告

カスタムの FieldBridge または TwoWayFieldBridge を実装する場合、null 値のインデックス化の処理は開発者に委ねられます (LuceneOptions.indexNullAs() の JavaDocs を参照してください)。

13.4.1.4. @NumericField

@NumericField は @Field と同種のアノテーションで、@Field または @DocumentId と同じスコープで指定できます。このアノテーションは Integer、Long、Float、および Double プロパティーに指定できます。インデックス化するときに、トライ木の構造を使用して値がインデックス化されます。プロパティーが数値のフィールドとしてインデックス化されると、効率的に範囲クエリーおよびソートを実行でき、標準の @Field プロパティーで同じクエリーを実行するよりもかなり高速に順序付けできます。@NumericField アノテーションには以下のパラメーターを使用できます。

値	定義
forField	(任意設定) 数値としてインデックス化される関連する @Field の名前を指定します。プロパティーに 1 つの @Field 宣言以外が含まれる場合のみ必須です。
precisionStep	(任意設定) トライ木構造がインデックスに保存される方法を変更します。precisionSteps を小さくするとディスク使用量が増え、範囲およびソートクエリーが高速になります。値を大きくすると、使用される容量が少なくなり、範囲クエリーの速度が通常の @Fields で実行される範囲クエリーの速度に近くなります。デフォルト値は 4 です。

@NumericField は Double、Long、Integer、および Float のみをサポートします。他の数値型に対して Lucene の同様の機能を利用できません。そのため、他の型はデフォルトまたはカスタムの TwoWayFieldBridge を用いた文字列のエンコーディングを使用する必要があります。

型の変換中に近似に対応できるのであればカスタムの NumericFieldBridge を使用できます。

例: カスタム NumericFieldBridge の定義

```
public class BigDecimalNumericFieldBridge extends NumericFieldBridge {
   private static final BigDecimal storeFactor = BigDecimal.valueOf(100);

   @Override
   public void set(String name, Object value, Document document,
LuceneOptions luceneOptions) {
      if ( value != null ) {
         BigDecimal decimalValue = (BigDecimal) value;
         Long indexedValue = Long.valueOf( decimalValue.multiply(
storeFactor ).longValue() );
         luceneOptions.addNumericFieldToDocument( name, indexedValue,
document );
      }
   }

    @Override
    public Object get(String name, Document document) {
        String fromLucene = document.get( name );
        BigDecimal storedBigDecimal = new BigDecimal( fromLucene );
        return storedBigDecimal.divide( storeFactor );
    }

}
```

13.4.1.5. @Id

エンティティーの **id** (識別子) プロパティーは、特定のエンティティーのインデックスを一意にするために Hibernate Search によって使用される特別なプロパティーです。設計上、**id** は保存する必要があり、トークン化できません。プロパティーをインデックス識別子としてマーク付けするには、**@DocumentId** アノテーションを使用します。JPA を使用し、@Id が指定済みである場合は、@DocumentId を省略できます。選択したエンティティー識別子はドキュメント識別子としても使用されます。

Infinispan クエリーはエンティティーの **id** プロパティーを使用してインデックスが一意に識別されるようにします。設計上、ID は保存され、トークンに変換しないようにする必要があります。プロパティーをインデックス ID としてマークするには、**@DocumentId** アノテーションを使用します。

例: インデックス化されたプロパティーの指定

```
@Entity
@Indexed
public class Essay {
    ...
    @Id
    @DocumentId
    public Long getId() { return id; }

    @Field(name="Abstract", store=Store.YES)
    public String getSummary() { return summary; }

    @Lob
    @Field
    public String getText() { return text; }

    @Field @NumericField( precisionStep = 6)
    public float getGrade() { return grade; }
}
```

上記の例では、**id**、**Abstract**、**text**、および **grade** の 4 つのフィールドでインデックスを定義します。デフォルトでは、JavaBean 仕様に従ってフィールド名に大文字が使用されていないことに注意してください。**grade** フィールドは、デフォルトよりも若干 precision step (精度ステップ) が大きく、数値としてアノテーションが付けられます。

13.4.1.6. プロパティーを複数回マッピングする

場合によっては、インデックスごとに若干異なるインデックス化ストラテジーでプロパティーを複数回マッピングする必要があることがあります。たとえば、フィールドでクエリーをソートするには、フィールドが分析されていない必要があります。このプロパティーで単語を基に検索し、ソートも行うには、インデックス化する必要があります (1 度分析し、1 度分析しません)。@Fields はこれを可能にします。

例: @Fields を使用してプロパティーを複数回マッピングする

```
@Entity
@Indexed(index = "Book" )
public class Book {
    @Fields( {
            @Field,
            @Field(name = "summary_forSort", analyze = Analyze.NO, store
= Store.YES)
            } )
    public String getSummary() {
        return summary;
    }

    ...
}
```

この例では、フィールド **summary** が 2 回インデックス化されます。**summary** としてトークン化される方法で 1 度インデックス化され、**summary_forSort** として非トークン化される方法でもう 1 度インデックス化されます。

13.4.1.7. 埋め込みオブジェクトおよび関連付けられたオブジェクト

関連付けられたオブジェクトおよび埋め込みオブジェクトは、ルートエンティティーインデックスの一部としてインデックス化できます。これは、関連付けられたオブジェクトのプロパティーを基にエンティティーを検索する場合に便利です。この目的は、関連付けられた市が Atlanta (Lucene クエリーパーサー言語では **address.city:Atlanta** に変換されます) である場所を返すことです。場所 (place) フィールドは **Place** インデックスでインデックス化されます。**Place** インデックスドキュメントには、クエリーが可能な **address.id**、**address.street**、および **address.city** フィールドも含まれます。

例: アソシエーションのインデックス化

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field
    private String street;

    @Field
    private String city;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}
```

@IndexedEmbedded 技術を使用するときに Lucene インデックスでデータが非正規化されるため、Hibernate Search は Place オブジェクトと Address オブジェクトの変更を認識し、インデックスを最新の状態にする必要があります。Address の変更時に Lucene ドキュメントが確実に更新されるように、双方向の関係の逆側を **@ContainedIn** でマーク付けします。

注記

@ContainedIn は、エンティティーを示すアソシエーションと埋め込みオブジェクト (コレクション) の両方に便利です。

これまでの説明を踏まえ、**@IndexedEmbedded** をネストする例を以下に示します。

例: @IndexedEmbedded と @ContainedIn のネストの使用

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field
    private String street;

    @Field
    private String city;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_")
    private Owner ownedBy;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}

@Embeddable
public class Owner {
    @Field
    private String name;
   ...
}
```

@*ToMany、**@*ToOne**、および **@Embedded** 属性には **@IndexedEmbedded** アノテーションを付けることができます。その後、関連付けられたクラスの属性は主要なエンティティーインデックスへ追加されます。インデックスに以下のフィールドが含まれます。

- id
- name
- address.street
- address.city
- address.ownedBy_name

デフォルトの接頭辞は **propertyName.** で、従来のオブジェクトナビゲーションの慣例に従います。これをオーバーライドするには **ownedBy** プロパティーで示されたように **prefix** 属性を使用します。

注記

空の文字列には接頭辞を設定できません。

オブジェクトグラフにクラス (インスタンスではない) の循環依存関係が含まれる場合は、**depth** プロパティーが必要になります。Owner が Place を参照する場合がこの例になります。Hibernate Search では、想定された深さに達すると (またはオブジェクトグラフの境界に達すると)、インデックス化された埋め込み属性が含まれなくなります。自己参照を持つクラスが循環依存関係の例になります。この例では、**depth** が 1 に設定されているため、Owner (存在する場合) の **@IndexedEmbedded** 属性は無視されます。

オブジェクトアソシエーションに **@IndexedEmbedded** を使用すると、以下のように Lucene のクエリー構文を使用してクエリーを表現できます。

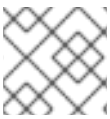
- 名前に JBoss が含まれ、住所の市が Atlanta である場所を返す場合、Lucene クエリーでは次のようになります。

```
+name:jboss +address.city:atlanta
```

- 名前に JBoss が含まれ、所有者の名前に Joe が含まれる場所を返す場合、Lucene クエリーでは次のようになります。

```
+name:jboss +address.ownedBy_name:joe
```

この挙動は、関係結合操作をより効率的に模擬しますが、データが重複されます。そのままの状態では Lucene インデックスにはアソシエーションの観念がなく、結合操作は存在しないことに注意してください。完全テキストのインデックス速度や機能充実の利点を活かしながらリレーショナルモデルの正規化を維持するのに役立つ可能性があります。



注記

関連付けされたオブジェクト自体が **@Indexed** である場合があります。

@IndexedEmbedded がエンティティーを参照する場合、関連付けは指向性を持つ必要があり、前述の例のように逆側には **@ContainedIn** アノテーションを付ける必要があります。このアノテーションを付けないと、関連付けられたエンティティーが更新されたときに Hibernate Search はルートインデッ

クスを更新できません (前述の例では、関連付けられた Address インスタンスが更新されたときに **Place** インデックスドキュメントを更新する必要があります)。

場合によっては、**@IndexedEmbedded** アノテーションが付けられたオブジェクト型が Hibernate および Hibernate Search が対象とするオブジェクト型でないことがあります。これは、実装の代わりにインターフェースが使用される場合に特に当てはまります。このため、**targetElement** パラメーターを使用して Hibernate Search が対象とするオブジェクト型をオーバーライドできます。

例: @IndexedEmbedded の targetElement プロパティーの使用

```
@Entity
@Indexed
public class Address {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field
    private String street;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_", )
    @Target(Owner.class)
    private Person ownedBy;


    ...
}

@Embeddable
public class Owner implements Person { ... }
```

13.4.1.8. 特定パスへ埋め込むオブジェクトの制限

@IndexedEmbedded アノテーションは、depth の代わりとして使用でき、depth とともに使用できる includePaths 属性も提供します。

depth のみを使用すると、埋め込まれた型のインデックス化されたフィールドすべてが同じ深さで再帰的に追加されます。このため、必要でない可能性がある他のすべてのフィールドを追加せずに特定のパスのみを選択することが難しくなります。

不必要なエンティティーのロードおよびインデックス化を回避するために、必要なパスのみを指定できます。通常のアプリケーションにはパスごとに異なる深さが必要になることがあり、以下の例で示されたようにパスを明示的に指定する必要があることがあります。

例: @IndexedEmbedded の includePaths プロパティーの使用

```
@Entity
@Indexed
public class Person {

   @Id
   public int getId() {
      return id;
   }
```

```
    @Field
    public String getName() {
       return name;
    }

    @Field
    public String getSurname() {
       return surname;
    }

    @OneToMany
    @IndexedEmbedded(includePaths = { "name" })
    public Set<Person> getParents() {
       return parents;
    }

    @ContainedIn
    @ManyToOne
    public Human getChild() {
       return child;
    }

     ...//other fields omitted
```

上記の例のようにマッピングを使用すると、**name**、**surname**、親の **name**、またはこれらの組み合わせで Person を検索できます。親の **surname** はインデックス化されないため、親の surname では検索ができませんが、インデックス化の速度が速くなり、スペースを節約でき、全体的なパフォーマンスが向上します。

@IndexedEmbeddedincludePaths には、指定されたパスと、depth の制限された値を指定して通常インデックス化するものが含まれます。includePaths を使用するときに depth を未定義にすると、depth`=0` を設定した場合と同様の挙動になり、含まれたパスのみがインデックス化されます。

例: @IndexedEmbedded の includePaths プロパティーの使用

```
@Entity
@Indexed
public class Human {

   @Id
   public int getId() {
      return id;
   }

   @Field
   public String getName() {
      return name;
   }

   @Field
   public String getSurname() {
      return surname;
   }

   @OneToMany
```

```
@IndexedEmbedded(depth = 2, includePaths = { "parents.parents.name" })
public Set<Human> getParents() {
   return parents;
}

@ContainedIn
@ManyToOne
public Human getChild() {
   return child;
}

 ...//other fields omitted
```

上記の例では、各 Human の name および surname 属性がインデックス化されます。depth 属性により、親の name と surname は再帰的に 2 つ目の行までインデックス化されます。Person、Person の親、または祖父母の name または surname を使用して検索できます。第 2 レベルを越えると、もう 1 つのレベルをインデックス化しますが、name のみで surname は対象になりません。

この結果、インデックスには以下のフィールドが含まれます。

- **id**: 主キーとして
- **_hibernate_class**: エンティティータイプを保存します
- **name**: 直接フィールドとして
- **surname**: 直接フィールドとして
- **parents.name**: depth 1 の埋め込まれたフィールドとして
- **parents.surname**: depth 1 の埋め込まれたフィールドとして
- **parents.parents.name**: depth 2 の埋め込まれたフィールドとして
- **parents.parents.surname**: depth 2 の埋め込まれたフィールドとして
- **parents.parents.parents.name**: includePaths によって指定された追加パスとして。最初の **parents.** はフィールド名から推測され、残りのパスは includePaths の属性になります。

必要なクエリーを最初に定義してアプリケーションを設計すると、その時点でユースケースの実装に必要なフィールドや不必要なフィールドが分かるため、インデックス化されたパスの明示的な制御が簡単になります。

13.4.2. ブースティング

Lucene は、特定のドキュメントやフィールドに異なる重要度を与えることができる**ブースティング** (boosting) という概念を持ちます。Lucene はインデックス時ブースティング (Index time boosting) と検索時ブースティング (Search time boosting) を区別します。今後の項では、Hibernate Search を使用してインデックス時ブースティングを実現する方法を説明します。

13.4.2.1. 静的なインデックス時ブースティング

インデックス化されたクラスやプロパティーの静的なブースト値を定義するには、**@Boost** アノテーションを使用します。このアノテーションを **@Field** 内で使用するか、メソッドまたはクラスレベルで直接指定します。

例: @Boost の異なる使用方法

```
@Entity
@Indexed

public class Essay {
    ...

    @Id
    @DocumentId
    public Long getId() { return id; }

    @Field(name="Abstract", store=Store.YES, boost=@Boost(2f))
    @Boost(1.5f)
    public String getSummary() { return summary; }

    @Lob
    @Field(boost=@Boost(1.2f))
    public String getText() { return text; }

    @Field
    public String getISBN() { return isbn; }

}
```

上記の例では、Essay が検索リストの最上部に達する可能性は 1.7 倍になります。summary フィールドは isbn フィールドよりも 3.0 倍重要になります (プロパティーの @Field.boost と @Boost は累積的であるため、2 x 1.5 になります)。text フィールドは isbn フィールドよりも 1.2 倍重要になります。この説明は厳密には正しくありませんが、わかりやすく実際の感覚では現実とほぼ同じになります。

13.4.2.2. 動的なインデックス時ブースティング

静的なインデックス時ブースティングで使用された **@Boost** アノテーションは静的ブースト係数を定義します。この係数は、実行時のインデックス化済みエンティティーのの状態とは関係ありません。ただし、ブースト係数がエンティティーの実際の状態に依存する可能性があるユースケースがあります。この場合は、**@DynamicBoost** アノテーションをカスタム BoostStrategy とともに使用できます。

例: 動的なブーストの例

```
public enum PersonType {
    NORMAL,
    VIP
}

@Entity
@Indexed
@DynamicBoost(impl = VIPBoostStrategy.class)
public class Person {
    private PersonType type;

    // ....
}

public class VIPBoostStrategy implements BoostStrategy {
    public float defineBoost(Object value) {
```

```
        Person person = ( Person ) value;
        if ( person.getType().equals( PersonType.VIP ) ) {
            return 2.0f;
        }
        else {
            return 1.0f;
        }
    }
}
```

上記の例では、VIPBoostStrategy を、インデックス化するときに使用される BoostStrategy インターフェースの実装として指定することにより、動的ブーストがクラスレベルで定義されています。**@DynamicBoost** はクラスまたはフィールドレベルに配置できます。エンティティー全体が defineBoost メソッドに渡されるか、またはアノテーションが付けられたフィールド/プロパティーの値のみが渡されるかはアノテーションの配置によります。渡されたオブジェクトを正しい型にキャストするのはユーザーに任されています。この例では、VIP の person のインデックス化された値はすべて普通の person の値よりも重要度が 2 倍になります。

注記

指定された BoostStrategy 実装は、パブリックの引数のないコンストラクターを定義する必要があります。

エンティティーで **@Boost**アノテーションと **@DynamicBoost** アノテーションを組み合わせることができます。定義されたブースト係数はすべて累積的です。

13.4.3. 分析

Analysis とはテキストを 1 つの言葉 (単語) に変換するプロセスのことであり、フルテキスト検索エンジンの主要機能の 1 つです。Lucene は Analyzer の概念を使用してこのプロセスを制御します。以下の項では、Hibernate Search が提供するアナライザーの設定方法を複数取り上げます。

13.4.3.1. デフォルトのアナライザーとクラスによるアナライザー

トークン化されたフィールドをインデックス化するために使用されるデフォルトのアナライザークラスは、**hibernate.search.analyzer** プロパティーから設定可能です。このプロパティーのデフォルト値は **org.apache.lucene.analysis.standard.StandardAnalyzer** です。

また、エンティティー、プロパティーおよび @Field ごとにアナライザークラスを定義することもできます。@Field ごとの定義は、単一のプロパティーから複数のフィールドをインデックス化するときに便利です。

Example: Different ways of using @Analyzer

```
@Entity
@Indexed
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {
    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;

    @Field
```

```
    private String name;

    @Field
    @Analyzer(impl = PropertyAnalyzer.class)
    private String summary;

    @Field(analyzer = @Analyzer(impl = FieldAnalyzer.class)
    private String body;

    ...
}
```

この例では、トークン化されたプロパティー (**name**) をインデックス化するために EntityAnalyzer が使用されます。例外は **summary** と **body** であり、これらはそれぞれ PropertyAnalyzer と FieldAnalyzer によってインデックス化されます。

警告

通常、同じエンティティーで異なるアナライザーを使用することは推奨されません。特に、クエリー全体で同じアナライザーを使用する QueryParser を使用する場合、クエリーの構築がより複雑になり、初心者にとって結果の予測がより困難になります。原則的に、フィールドではインデックス化とクエリーに同じアナライザーを使用します。

13.4.3.2. 名前付きのアナライザー

アナライザーの使用は非常に複雑になることがあります。そのため、Hibernate Search にはアナライザー定義の概念が導入されています。アナライザー定義は多くの @Analyzer 宣言による再使用が可能であり、以下のもので構成されます。

- **名前:** 定義を参照するために使用される一意の文字列。
- **文字フィルターのリスト:** 各文字フィルターはトークン化の前に入力文字を事前処理します。文字フィルターは文字を追加、変更、または削除できます。一般的な使用例の 1 つが文字の正規化です。
- **トークナイザー:** 入力ストリームを個別の単語にトークン化します。
- **フィルターのリスト:** 各フィルターは単語を削除または変更します。また、トークナイザーによって提供されたストリームに単語を追加することもあります。

文字フィルターのリスト、トークナイザー、およびそれに続くフィルターのリストによってタスクが分離され、各コンポーネントの再使用が容易になり、非常に柔軟にカスタマイズされたアナライザーを構築できます (レゴブロックのように)。通常、文字フィルターが文字入力の事前処理を行った後、Tokenizer が文字入力をトークンに変換してトークン化プロセスを開始します。その後、TokenFilter によってトークンがさらに処理されます。Hibernate Search は Solr アナライザーフレームワークを使用してこのインフラストラクチャーをサポートします。

ここで、以下に示された具体的な例を確認してみましょう。最初に、文字フィルターはそのファクトリーによって定義されます。例ではマッピング文字フィルターが使用され、マッピングファイルに指定されたルールを基にして入力の文字を置き換えます。次にトークナイザーが定義されます。この例では

標準のトークナイザーが使用されます。最後にフィルターのリストがファクトリーによって定義されます。この例では、専用の単語プロパティーファイルを読み取って StopFilter フィルターが構築されます。また、フィルターは大文字と小文字を区別しないことが想定されます。

例: @AnalyzerDef および Solr フレームワーク

```
@AnalyzerDef(name="customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value = "org/hibernate/search/test/analyzer/solr/mapping-
chars.properties")
    })
  },
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
      @Parameter(name="words",
        value=
"org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
      @Parameter(name="ignoreCase", value="true")
    })
})
public class Team {
    ...
}
```

注記

フィルターと文字フィルターは、@AnalyzerDef アノテーションに定義された順序で適用されます。順序が関係することに注意してください。

トークナイザー、トークンフィルター、または文字フィルターの一部には設定またはメタデータファイルなどのリソースをロードするものがあります。これには停止フィルターと類義語フィルターが該当します。リソース文字セットが VM のデフォルトを使用しない場合、**resource_charset** パラメーターを追加して明示的に指定できます。

例: 特定の文字セットを使用したプロパティーファイルのロード

```
@AnalyzerDef(name="customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value = "org/hibernate/search/test/analyzer/solr/mapping-
chars.properties")
    })
  },
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
      @Parameter(name="words",
```

```
        value=
"org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
      @Parameter(name="resource_charset", value = "UTF-16BE"),
      @Parameter(name="ignoreCase", value="true")
  })
})
public class Team {
    ...
}
```

以下の例で示されているように、アナライザー定義は定義後に @Analyzer 宣言で再使用できます。

例: 名前によるアナライザーの参照

```
@Entity
@Indexed
@AnalyzerDef(name="customanalyzer", ... )
public class Team {
    @Id
    @DocumentId
    @GeneratedValue
    private Integer id;

    @Field
    private String name;

    @Field
    private String location;

    @Field
    @Analyzer(definition = "customanalyzer")
    private String description;
}
```

@AnalyzerDef によって宣言されたアナライザーインスタンスは、SearchFactory で名前を用いて使用することもできます。これはクエリーの構築時に役に立ちます。

```
Analyzer analyzer =
fullTextSession.getSearchFactory().getAnalyzer("customanalyzer");
```

クエリーのフィールドは、共通の「言語」を話すよう、フィールドをインデックス化するために使用したのと同じアナライザーで分析する必要があります。クエリーとインデックス化プロセスの間では同じトークンが再使用されます。このルールには例外もありますが、ほとんどの場合に当てはまります。完全に理解して作業を行っている場合以外はこのルールに従ってください。

13.4.3.3. 使用可能なアナライザー

Solr および Lucene には便利なデフォルトの文字フィルター、トークナイザー、およびフィルターが多く含まれています。文字フィルターファクトリー、トークナイザーファクトリー、およびフィルターファクトリーの完全なリストについては、http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters を参照してください。この一部を以下に示します。

表13.8 利用可能な文字フィルターの例

ファクトリー	説明	パラメーター
MappingCharFilterFactory	リソースファイルに指定されたマッピングを基に、1 つ以上の文字を置き換えます。	**`mapping`**: "á" ⇒ "a"; "ñ" ⇒ "n"; "ø" ⇒ "o" という形式を使用したマッピングが含まれるリソースファイルを示します。
HTMLStripCharFilterFactory	標準の HTML タグを削除し、テキストは保持します。	none

表13.9 利用可能なトークナイザーの例

ファクトリー	説明	パラメーター
StandardTokenizerFactory	Lucene の標準トークナイザーを使用します。	none
HTMLStripCharFilterFactory	標準の HTML タグを削除してテキストは保持し、StandardTokenizer へ渡します。	none
PatternTokenizerFactory	指定の正規表現パターンでテキストを改行します。	**pattern**: トークン化に使用する正規表現。 **group**: トークンに抽出するパターングループを示します。

表13.10 利用可能なフィルターの例

ファクトリー	説明	パラメーター
StandardFilterFactory	頭字語からピリオドを削除し、単語からアポストロフィー (') を削除します。	none
LowerCaseFilterFactory	すべての言葉を小文字にします。	none
StopFilterFactory	ストップワードのリストと一致する言葉 (トークン) を削除します。	**words**: ストップワードが含まれるリソースファイルを示します。 **ignoreCase**: ストップワードを比較するときに case を無視する場合は true、無視しない場合は false。

ファクトリー	説明	パラメーター
SnowballPorterFilterFactory	単語を指定言語の語根にします。たとえば、protect、protects、および protection はすべて同じ語根を持ちます。このようなフィルターを使用すると、関連する単語に一致する検索を実行できます。	**language**: デンマーク語、オランダ語、英語、フィンランド語、フランス語、ドイツ語、イタリア語、ノルウェー語、ポルトガル語、ロシア語、スペイン語、スウェーデン語など。

IDEの **org.apache.lucene.analysis.TokenizerFactory** および **org.apache.lucene.analysis.TokenFilterFactory** の実装をすべてチェックし、利用できる実装を確認することが推奨されます。

13.4.3.4. 動的アナライザーの選択

これまでの説明では、アナライザーを指定する方法はすべて静的でした。しかし、多言語のアプリケーションなど、インデックス化されるエンティティーの現在の状態に応じてアナライザーを選択すると便利なユースケースもあります。たとえば、BlogEntry クラスの場合、アナライザーはエントリーの言語プロパティーに依存できます。このプロパティーに応じて、実際のテキストをインデックス化するために適切な言語固有のステマーを選択する必要があります。

動的なアナライザー選択を有効にするために、Hibernate Search では AnalyzerDiscriminator アノテーションが導入されました。以下の例は、このアノテーションの使用方法を示しています。

例: @AnalyzerDiscriminator の使用方法

```
@Entity
@Indexed
@AnalyzerDefs({
  @AnalyzerDef(name = "en",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = EnglishPorterFilterFactory.class
      )
    }),
  @AnalyzerDef(name = "de",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = GermanStemFilterFactory.class)
    })
})
public class BlogEntry {

    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;

    @Field
    @AnalyzerDiscriminator(impl = LanguageDiscriminator.class)
```

```
    private String language;

    @Field
    private String text;

    private Set<BlogEntry> references;

    // standard getter/setter
    ...
}
```

```
public class LanguageDiscriminator implements Discriminator {

    public String getAnalyzerDefinitionName(Object value, Object entity,
String field) {
        if ( value == null || !( entity instanceof BlogEntry ) ) {
            return null;
        }
        return (String) value;

    }
}
```

@AnalyzerDiscriminator を使用する前に、動的に使用されるすべてのアナライザーを **@AnalyzerDef** 定義で事前に定義する必要があります。この場合は、アナライザーを動的に選択するためのクラスまたはエンティティーの特定のプロパティーに **@AnalyzerDiscriminator** アノテーションを配置します。**AnalyzerDiscriminator** の **impl** パラメーターを使用して、Discriminator インターフェースの具体的な実装を指定します。このインターフェースに実装を提供することはユーザーに任されています。実装しなければならない唯一のメソッドは **getAnalyzerDefinitionName()** であり、このメソッドは Lucene ドキュメントにフィールドが追加されるたびに呼び出されます。インデックス化されるエンティティーもインターフェースメソッドへ渡されます。**AnalyzerDiscriminator** がクラスレベルではなくプロパティーレベルに配置された場合のみ **value** パラメーターが設定されます。この場合、値はこのプロパティーの現在の値を表します。

Discriminator インターフェースの実装は、既存アナライザー定義の名前を返す必要がありますが、デフォルトのアナライザーをオーバーライドしない場合は null を返す必要があります。上記の例では、言語パラメーターが **@AnalyzerDefs** で指定された名前に一致する「de」または「en」のどちらかであることを仮定します。

13.4.3.5. アナライザーの読み出し

ステミングや音声的近似などを活用するために、ドメインモデルで複数のアナライザーが使用された場合にアナライザーを読み出すことができます。この場合、同じアナライザーを使用してクエリーを構築します。この代わりに、正しいアナライザーを自動的に選択する Hibernate Search クエリー DSL を使用することもできます。

Lucene プログラム API または Lucene クエリーパーサーのいずれを使用する場合でも、指定のエンティティーのスコープ指定されたアナライザーを読み出しできます。スコープ指定されたアナライザーは、インデックス化されたフィールドに応じて適切なアナライザーを適用するアナライザーです。各フィールドで動作する各エンティティーには複数のアナライザーを定義できます。スコープ指定されたアナライザーはすべてのアナライザーをコンテキストを意識したアナライザーに統合します。この理論は若干複雑ですが、クエリーで正しいアナライザーを使用することは簡単です。

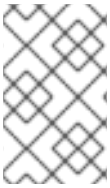
注記

子エンティティーにプログラムを用いたマッピングを使用する場合、子エンティティーによって定義されたフィールドのみが表示されます。親エンティティーから継承されたフィールドやメソッドは設定できません。親エンティティーから継承されたプロパティーを設定するには、子エンティティーでプロパティーをオーバーライドするか、親エンティティーに対してプログラムを用いたマッピングを作成します。これは、子エンティティーで再定義しないと親エンティティーのフィールドまたはメソッドにアノテーションをつけられない場合にアノテーションの使用を模擬します。

例: フルテキストのクエリー構築時におけるスコープ指定されたアナライザーの使用

```
org.apache.lucene.queryParser.QueryParser parser = new QueryParser(
    "title",
    fullTextSession.getSearchFactory().getAnalyzer( Song.class )
);

org.apache.lucene.search.Query luceneQuery =
    parser.parse( "title:sky Or title_stemmed:diamond" );

org.hibernate.Query fullTextQuery =
    fullTextSession.createFullTextQuery( luceneQuery, Song.class );

List result = fullTextQuery.list(); //return a list of managed objects
```

上記の例では、歌のタイトルが 2 つのフィールドでインデックス化されます。標準のアナライザーは **`title`** フィールドで使用され、ステミングアナライザーは **`title_stemmed`** フィールドで使用されます。検索ファクトリーによって提供されたアナライザーを使用して、クエリーは目的のフィールドに応じて適切なアナライザーを使用します。



注記

@AnalyzerDef によって定義されたアナライザーは、**`searchFactory.getAnalyzer(String)`** を使用して定義名で取得することもできます。

13.4.4. ブリッジ

これまでの説明では、エンティティーの基本的なマッピングで重要な条件の 1 つが無視されていました。Lucene では、すべてのインデックスフィールドを文字列として表す必要があります。インデックス化するには **`@Field`** アノテーションが付けられたエンティティープロパティーをすべて文字列に変換する必要があります。これまでの説明で取り上げなかった理由は、Hibernate Search のビルトインブリッジによってほとんどのプロパティーが変換されるためです。しかし、場合によっては変換プロセスでさらに細かい制御が必要になることがあります。

13.4.4.1. ビルトインブリッジ

Hibernate Search には、Java プロパティー型とそのフルテキスト表現との間のビルドインブリッジが含まれています。

null

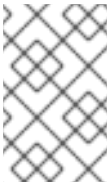
デフォルトでは **`null`** 要素はインデックス化されません。 Lucene は null 要素をサポートしませんが、場合によっては **`null`** の値を表すカスタムトークンを挿入すると便利なことがあります。

java.lang.String

文字列は short、Short、integer、Integer、long、Long、float、Float、double と同様にインデックス化されます。

Double、BigInteger、BigDecimal

数字は文字列の表現に変換されます。Lucene はそのままでは数字を比較できず (範囲指定のクエリーで使用)、パディングを行う必要があることに注意してください。



注記

Range クエリーの使用には欠点があるため、代わりに結果クエリーを適切な範囲にフィルターする Filter クエリーを使用できます。Hibernate Search では、カスタム StringBridge の使用もサポートされています (カスタムブリッジを参照)。

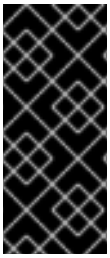
java.util.Date

日付はグリニッジ標準時 (GMT) で yyyyMMddHHmmssSSS の形式で保存されます。たとえば、アメリカ東部標準時 (EST) の 2006 年 11 月 7 日午後 4 時 3 分 12 秒は 200611072203012 になります。内部の形式は気にする必要はありません。TermRangeQuery を使用するときに重要なのは、日付がグリニッジ標準時で表されることを認識することです。
通常、日付をミリ秒まで保存する必要はありません。**`@DateBridge`** はインデックスに保存するのに適切な精度 (resolution) を定義します (**`@DateBridge(resolution=Resolution.DAY)`**)。日付パターンはこの定義に従って省略されます。

```
@Entity
@Indexed
public class Meeting {
    @Field(analyze=Analyze.NO)

    private Date date;
    ...
```

警告

`MILLISECOND` よりも小さい精度を持つ日付を **`@DocumentId`** にすることはできません。



重要

デフォルトの Date ブリッジは Lucene の DateTools を使用して String からの変換と String への変換を行います。つまり、すべての日付は GMT 時間で表されます。日付を固定の時間帯で保存することが要件である場合は、カスタムの日付ブリッジを実装する必要があります。日付のインデックス化および検索に関するアプリケーションの要件を理解するようにしてください。

java.net.URI、java.net.URL

URI および URL は文字列の表現に変換されます。

java.lang.Class

クラスは完全修飾クラス名に変換されます。クラスがリハイドレートされる場合は、スレッドコンテキストクラスローダーが使用されます。

13.4.4.2. カスタムブリッジ

場合によっては Hibernate Search のビルトインブリッジが一部のプロパティータイプに対応しなかったり、ブリッジによって使用される String 表現が要件に合わなかったりすることがあります。この問題に対応する方法を以下に説明します。

13.4.4.2.1. StringBridge

最も簡単な方法は、想定される Object から String へのブリッジの実装を Hibernate Search に提供することです。これを行うには、**`org.hibernate.search.bridge.StringBridge`** インターフェースを実装する必要があります。同時に使用されるため、すべての実装はスレッドセーフである必要があります。

例: カスタムの StringBridge 実装

```
/**
 * Padding Integer bridge.
 * All numbers will be padded with 0 to match 5 digits
 *
 * @author Emmanuel Bernard
 */
public class PaddedIntegerBridge implements StringBridge {

    private int PADDING = 5;

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > PADDING)
            throw new IllegalArgumentException( "Try to pad on a number
too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < PADDING ;
padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}
```

前の例で定義された文字列ブリッジの場合、**`@FieldBridge`** アノテーションによってすべてのプロパティーおよびフィールドでこのブリッジを使用できます。

```
@FieldBridge(impl = PaddedIntegerBridge.class)
private Integer length;
```

13.4.4.2.2. パラメーター化されたブリッジ

ブリッジ実装をより柔軟にするために、パラメーターをブリッジ実装に渡すこともできます。以下の例では、ParameterizedBridge インターフェースが実装され、パラメーターは **`@FieldBridge`** アノテーションを介して渡されます。

例: ブリッジ実装へパラメーターを渡す

```
public class PaddedIntegerBridge implements StringBridge,
ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map<String,String> parameters) {
        String padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = Integer.parseInt( padding );
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a number
too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < padding ;
padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}


//property
@FieldBridge(impl = PaddedIntegerBridge.class,
             params = @Parameter(name="padding", value="10")
            )
private Integer length;
```

ParameterizedBridge インターフェース
は、**StringBridge**、**TwoWayStringBridge**、**FieldBridge** 実装によって実装できます。

すべての実装はスレッドセーフである必要がありますが、パラメーターは初期化時に設定され、この段階では特別な措置は必要ありません。

13.4.4.2.3. 型対応ブリッジ

以下に適用されているブリッジの型を取得すると便利な場合があります。

- フィールド/ゲッターレベルのブリッジに対するプロパティーの戻り値の型。
- クラスレベルのブリッジに対するクラス型。

例としては、独自の方法で列挙を処理し、実際の列挙型にアクセスする必要があるブリッジが挙げられます。AppliedOnTypeAwareBridge を実装するすべてのブリッジは、ブリッジが適用されている型をインジェクトします。パラメーターと同様に、インジェクトされた型はスレッドセーフに関しては特別な注意が必要ありません。

13.4.4.2.4. 双方向ブリッジ

@DocumentId アノテーションが付けられた ID プロパティーでブリッジ実装を使用することが想定される場合、TwoWayStringBridge を使用する必要があります。これは **StringBridge** が若干拡張されたものです。Hibernate Search は識別子の文字列表現を読み取り、オブジェクトを生成する必要があります。**@FieldBridge** アノテーションの使用方法は同じです。

例: ID プロパティーに使用できる TwoWayStringBridge の実装

```
public class PaddedIntegerBridge implements TwoWayStringBridge,
ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a number
too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < padding ;
padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }

    public Object stringToObject(String stringValue) {
        return new Integer(stringValue);
    }
}


//id property
@DocumentId
@FieldBridge(impl = PaddedIntegerBridge.class,
             params = @Parameter(name="padding", value="10")
private Integer id;
```

重要

「object = stringToObject(objectToString(object))」のように、双方向処理をべき等にすることが重要になります。

13.4.4.2.5. FieldBridge

一部のユースケースでは、プロパティーを Lucene インデックスにマップするときにオブジェクトから文字列への単純な変換以上のものが必要になることがあります。柔軟性を最大化にするために、ブリッジを FieldBridge として実装することもできます。このインターフェースはプロパティー値を提供し、

その値を Lucene Document で自由にマップできるようにします。たとえば、1 つのプロパティーを 2 つの異なるドキュメントフィールドに保存できます。インターフェースは Hibernate UserType の概念と非常に似ています。

例: FieldBridge インターフェースの実装

```
/**
 * Store the date in 3 different fields - year, month, day - to ease Range
Query per
 * year, month or day (eg get all the elements of December for the last 5
years).
 * @author Emmanuel Bernard
 */
public class DateSplitBridge implements FieldBridge {
    private final static TimeZone GMT = TimeZone.getTimeZone("GMT");

    public void set(String name, Object value, Document document,
LuceneOptions luceneOptions) {
        Date date = (Date) value;
        Calendar cal = GregorianCalendar.getInstance(GMT);
        cal.setTime(date);
        int year = cal.get(Calendar.YEAR);
        int month = cal.get(Calendar.MONTH) + 1;
        int day = cal.get(Calendar.DAY_OF_MONTH);

        // set year
        luceneOptions.addFieldToDocument(
            name + ".year",
            String.valueOf( year ),
            document );

        // set month and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".month",
            month < 10 ? "0" : "" + String.valueOf( month ),
            document );

        // set day and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".day",
            day < 10 ? "0" : "" + String.valueOf( day ),
            document );
    }
}

//property
@FieldBridge(impl = DateSplitBridge.class)
private Date date;
```

上記の例では、フィールドは直接 Document へ追加されていません。追加は LuceneOptions ヘルパーに委譲されています。このヘルパーは **Store** や **TermVector** などの **@Field** で選択されたオプションを適用したり、選択された @Boost の値を適用したりします。特に、**COMPRESS** 実装の複雑さをカプセル化するのに便利です。LuceneOptions に委譲して Document にフィールドを追加することが推奨されますが、必要な場合は Document を直接編集し、LuceneOptions を無視することも可能です。

注記

LuceneOptions のようなクラスは、Lucene API の変更からアプリケーションを守り、コードを簡単にするために作成されます。可能な場合はこのようなクラスを使用してください。ただし、柔軟性がさらに必要な場合、使用は強制されません。

13.4.4.2.6. ClassBridge

場合によっては、エンティティーのプロパティーを複数組み合わせ、特定の方法で Lucene インデックスへインデックス化すると便利です。@ClassBridge および @ClassBridges アノテーションは、プロパティーレベルではなくクラスレベルで定義できます。この場合、カスタムフィールドブリッジ実装は特定のプロパティーでなく値パラメーターとしてエンティティーインスタンスを受け取ります。以下の例では示されていませんが、@ClassBridge は項基本的なマッピングで説明されている termVector 属性をサポートします。

例: クラスブリッジの実装

```
@Entity
@Indexed
(name="branchnetwork",
             store=Store.YES,
             impl = CatFieldsClassBridge.class,
             params = @Parameter( name="sepChar", value=" " ) )
public class Department {
    private int id;
    private String network;
    private String branchHead;
    private String branch;
    private Integer maxEmployees
    ...
}

public class CatFieldsClassBridge implements FieldBridge,
ParameterizedBridge {
    private String sepChar;

    public void setParameterValues(Map parameters) {
        this.sepChar = (String) parameters.get( "sepChar" );
    }

    public void set( String name, Object value, Document document,
LuceneOptions luceneOptions) {
        // In this particular class the name of the new field was passed
        // from the name field of the ClassBridge Annotation. This is not
        // a requirement. It just works that way in this instance. The
        // actual name could be supplied by hard coding it below.
        Department dep = (Department) value;
        String fieldValue1 = dep.getBranch();
        if ( fieldValue1 == null ) {
            fieldValue1 = "";
        }
        String fieldValue2 = dep.getNetwork();
        if ( fieldValue2 == null ) {
            fieldValue2 = "";
        }
        String fieldValue = fieldValue1 + sepChar + fieldValue2;
```

```
        Field field = new Field( name, fieldValue,
luceneOptions.getStore(),
            luceneOptions.getIndex(), luceneOptions.getTermVector() );
        field.setBoost( luceneOptions.getBoost() );
        document.add( field );
    }
}
```

この例では、特定の CatFieldsClassBridge が **`department`** インスタンスへ適用され、フィールドブリッジによってブランチとネットワークの両方が連結され、連結がインデックス化されます。

13.5. クエリー

Hibernate SearchHibernate Search は Lucene クエリーを実行し、InfinispanHibernate セッションによって管理されるドメインオブジェクトを読み出しできます。検索は Hibernate パラダイムの範囲で Lucene の機能を提供し、Hibernate の従来の検索メカニズム (HQL、Criteria クエリー、ネイティブ SQL クエリー) に他の可能性を提供します。

クエリーの準備および実行は、以下の 4 つの段階で構成されます。

- FullTextSession の作成
- Hibernate QueryHibernate Search クエリー DSL (推奨) または Lucene Query API を使用した Lucene クエリーの作成
- org.hibernate.Query を使用した Lucene クエリーのラッピング
- サンプル list() または scroll() の呼び出しによる検索の実行

クエリー機能にアクセスするには、FullTextSession を使用します。この検索固有のセッションはクエリーおよびインデックス化の機能を提供するために正規の org.hibernate.Session をラップします。

例: FullTextSession の作成

```
Session session = sessionFactory.openSession();
...
FullTextSession fullTextSession = Search.getFullTextSession(session);
```

FullTextSession を使用して、Hibernate SearchHibernate Search クエリー DSL またはネイティブ Lucene クエリーを使用したフルテキストクエリーを構築します。

Hibernate SearchHibernate Search クエリー DSL を使用する場合は以下のコードを使用します。

```
final QueryBuilder b =
fullTextSession.getSearchFactory().buildQueryBuilder().forEntity(
Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
    b.keyword()
        .onField("history").boostedTo(3)
        .matching("storm")
        .createQuery();
```

```
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(
luceneQuery );
List result = fullTextQuery.list(); //return a list of managed objects
```

この代わりに、Lucene クエリーパーサーまたは Lucene プログラム API のいずれかを使用して Lucene クエリーを記述します。

例: QueryParser を用いた Lucene クエリーの作成

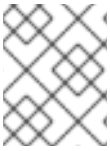
```
SearchFactory searchFactory = fullTextSession.getSearchFactory();
org.apache.lucene.queryParser.QueryParser parser =
    new QueryParser("title", searchFactory.getAnalyzer(Myth.class) );
try {
    org.apache.lucene.search.Query luceneQuery = parser.parse(
"history:storm^3" );
}
catch (ParseException e) {
    //handle parsing failure
}

org.hibernate.Query fullTextQuery =
fullTextSession.createFullTextQuery(luceneQuery);
List result = fullTextQuery.list(); //return a list of managed objects
```

Lucene クエリー上に構築された Hibernate クエリーは org.hibernate.Query です。このクエリーは HQL (Hibernate Query Language)、Native、Criteria などの他の Hibernate クエリー機能と同じパラダイム内に留まります。クエリーには list()、uniqueResult()、iterate() and scroll() などのメソッドを使用します。

Hibernate Java Persistence API では同じ拡張を使用できます。

例: JPA API を使用した Search クエリーの作成

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

FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);

...
final QueryBuilder b = fullTextEntityManager.getSearchFactory()
    .buildQueryBuilder().forEntity( Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
    b.keyword()
        .onField("history").boostedTo(3)
        .matching("storm")
        .createQuery();

javax.persistence.Query fullTextQuery =
fullTextEntityManager.createFullTextQuery( luceneQuery );

List result = fullTextQuery.getResultList(); //return a list of managed
objects
```



注記

以下の例では Hibernate API を使用しますが、FullTextQuery が読み出される方法を調整するだけで Java Persistence API を使用して同じ例を簡単に書き直すことができます。

13.5.1. クエリーの構築

Hibernate Search クエリーは Lucene クエリー上で構築されるため、ユーザーはすべての Lucene クエリータイプを使用できます。クエリーが構築されると、Hibernate Search は org.hibernate.Query をクエリー操作 API として使用してさらにクエリー処理を行います。

13.5.1.1. Lucene API を使用した Lucene クエリーの構築

Lucene API では、クエリーパーサー (簡単なクエリー) または Lucene プログラム API (複雑なクエリー) を使用します。Lucene クエリーの構築は、Hibernate Search ドキュメントの範囲外になります。詳細については、オンラインの Lucene ドキュメント、**Lucene in Action**、または **Hibernate Search in Action** を参照してください。

13.5.1.2. Lucene クエリーの構築

Lucene プログラム API はフルテキストクエリーを有効にします。しかし、Lucene プログラム API を使用する場合はパラメーターを同等の文字列に変換し、さらに正しいアナライザーを適切なフィールドに適用する必要があります。たとえば、N-gram アナライザーは複数の N-gram を指定の言葉のトークンとして使用し、そのように検索する必要があります。この作業には QueryBuilder の使用が推奨されます。

Hibernate Search クエリー API は以下の特徴を持ちます。

- メソッド名は英語になります。そのため、API 操作は一連の英語のフレーズや指示として読み取りおよび理解されます。
- 現在入力した接頭辞の補完を可能にし、ユーザーが適切なオプションを選択できる IDE 自動補完を使用します。
- チェイニングメソッドパターンを頻繁に使用します。
- API 操作の使用および読み取りは簡単です。

API を使用するには、最初に指定の **`indexedentitytype`** にアタッチされるクエリービルダーを作成します。この QueryBuilder は、使用するアナライザーと適用するフィールドブリッジを認識します。複数の QueryBuilder を作成できます (クエリーのルートに関係する各エンティティー型ごと)。QueryBuilder は SearchFactory から派生します。

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder().forEntity(
Myth.class ).get();
```

指定のフィールドに使用するアナライザーをオーバーライドすることもできます。

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder()
    .forEntity( Myth.class )
        .overridesForField("history","stem_analyzer_definition")
    .get();
```

クエリービルダーは Lucene クエリーの構築に使用されるようになりました。Lucene のクエリーパーサーを使用して生成されたカスタマイズ済みクエリーまたは Lucene プログラム API を使用してアセンブルされた Query オブジェクトは、Hibernate Search DSL とともに使用されます。

13.5.1.3. キーワードクエリー

以下の例は特定の単語を検索する方法を示しています。

```
Query luceneQuery =
mythQB.keyword().onField("history").matching("storm").createQuery();
```

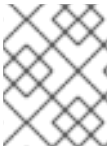
表13.11 キーワードクエリーパラメーター

パラメーター	説明
keyword()	このパラメーターを使用して特定の単語を見つけます。
onField()	このパラメーターを使用して単語を検索する Lucene フィールドを指定します。
matching()	このパラメーターを使用して検索する文字列の一致を指定します。
createQuery()	Lucene クエリーオブジェクトを作成します。

- 値「storm」は **history** FieldBridge から渡されます。これは、数字や日付が関係する場合に便利です。
- その後、フィールドブリッジの値はフィールド **history** をインデックス化するために使用されるアナライザーへ渡されます。これにより、クエリーがインデックス化と同じ用語変換を使用するようにします (小文字、N-gram、ステミングなど)。分析プロセスが指定の単語に対して複数の用語を生成する場合、ブール値クエリーは **SHOULD** ロジック (おおよそ **OR** ロジックと同様) とともに使用されます。

文字列型でないプロパティーを検索します。

```
@Indexed
public class Myth {
  @Field(analyze = Analyze.NO)
  @DateBridge(resolution = Resolution.YEAR)
  public Date getCreationDate() { return creationDate; }
  public Date setCreationDate(Date creationDate) { this.creationDate =
creationDate; }
  private Date creationDate;

  ...
}

Date birthdate = ...;
Query luceneQuery =
mythQb.keyword().onField("creationDate").matching(birthdate).createQuery()
;
```



注記

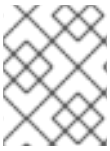
プレーンな Lucene では、Date オブジェクトは文字列の表現に変換する必要がありました (この例では年)。

FieldBridge に objectToString メソッドがある場合 (および組み込みのすべての FieldBridge 実装にこのメソッドがある場合)、この変換はどのオブジェクトに対しても実行できます。

次の例は、N-gram アナライザーを使用するフィールドを検索します。N-gram アナライザーは単語の N-gram の連続をインデックス化します。これは、ユーザーによる誤字を防ぐのに役立ちます。たとえば、単語 hibernate の 3-gram は hib、ibe、ber、ern、rna、nat、ate になります。

```
@AnalyzerDef(name = "ngram",
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class ),
  filters = {
    @TokenFilterDef(factory = StandardFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class),
    @TokenFilterDef(factory = NGramFilterFactory.class,
      params = {
        @Parameter(name = "minGramSize", value = "3"),
        @Parameter(name = "maxGramSize", value = "3") } )
  }
)

public class Myth {
  @Field(analyzer=@Analyzer(definition="ngram")
  public String getName() { return name; }
  public String setName(String name) { this.name = name; }
  private String name;

  ...
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword().onField("name").matching("Sisiphus")
   .createQuery();
```

一致する単語「Sisiphus」は小文字に変換され、3-gram (sis、isi、sip、iph、phu、hus) に分割されます。各 N-gram はクエリーの一部になります。その後、ユーザーは Sysiphus (i ではなく **y**) myth (シーシュポスの神話) を検索できます。ユーザーに対してすべてが透過的に行われます。



注記

特定のフィールドがフィールドブリッジまたはアナライザーを使用しないようにするには、ignoreAnalyzer() または ignoreFieldBridge() 関数を呼び出すことができます。

同じフィールドで可能な単語を複数検索し、すべてを一致する句に追加します。

```
//search document with storm or lightning in their history
Query luceneQuery =
    mythQB.keyword().onField("history").matching("storm
lightning").createQuery();
```

複数のフィールドで同じ単語を検索するには、onFields メソッドを使用します。

```
Query luceneQuery = mythQB
    .keyword()
    .onFields("history","description","name")
    .matching("storm")
    .createQuery();
```

同じ用語を検索する場合でも、あるフィールドに対して他のフィールドとは異なる処理をする必要があることがあります。このような場合は andField() メソッドを使用します。

```
Query luceneQuery = mythQB.keyword()
    .onField("history")
    .andField("name")
      .boostedTo(5)
    .andField("description")
    .matching("storm")
    .createQuery();
```

前述の例では、フィールド名のみが 5 にブーストされます。

13.5.1.4. ファジークエリー

レーベンシュタイン距離 (Levenshtein Distance) アルゴリズムを基にしてファジークエリーを実行するには、**keyword** クエリーで開始して **fuzzy** フラグを追加します。

```
Query luceneQuery = mythQB
    .keyword()
      .fuzzy()
        .withThreshold( .8f )
        .withPrefixLength( 1 )
    .onField("history")
    .matching("starm")
    .createQuery();
```

threshold を越えると 2 つの用語の一致を考慮します。これは、0 から 1 の間の小数であり、デフォルト値は 0.5 になります。**prefixLength** は「ファジーの度合い」によって無視される接頭辞の長さになります。デフォルト値は 0 ですが、大量の異なる用語が含まれるインデックスではゼロ以外の値を指定することが推奨されます。

13.5.1.5. ワイルドカードクエリー

ワイルドカードクエリーは、単語の一部のみがわかる場合に便利です。**?** は単一の文字を表し、* は複数の文字を表します。パフォーマンス上の理由で、クエリーの最初に **?** または * を使用しないことが推奨されます。

```
Query luceneQuery = mythQB
    .keyword()
      .wildcard()
    .onField("history")
    .matching("sto*")
    .createQuery();
```

注記

ワイルドカードクエリーは、アナライザーを一致する用語に適用しません。***** または **?** が適切に処理されないリスクが大変高くなります。

13.5.1.6. フレーズクエリー

これまでは、単語または単語のセットを検索しましたが、完全一致の文または近似の文を検索することも可能です。文の検索には phrase() を使用します。

```
Query luceneQuery = mythQB
    .phrase()
    .onField("history")
    .sentence("Thou shalt not kill")
    .createQuery();
```

おおよその文はスロップ (slop) 係数を追加すると検索可能になります。スロップ係数は、その文で許可される別の単語の数を表します。これは、within または near 演算子と同様に動作します。

```
Query luceneQuery = mythQB
    .phrase()
      .withSlop(3)
    .onField("history")
    .sentence("Thou kill")
    .createQuery();
```

13.5.1.7. 範囲クエリー

範囲クエリーは指定の境界の間、上、または下で値を検索します。

```
//look for 0 <= starred < 3
Query luceneQuery = mythQB
    .range()
    .onField("starred")
    .from(0).to(3).excludeLimit()
    .createQuery();

//look for myths strictly BC
Date beforeChrist = ...;
Query luceneQuery = mythQB
    .range()
    .onField("creationDate")
    .below(beforeChrist).excludeLimit()
    .createQuery();
```

13.5.1.8. クエリーの組み合わせ

クエリーを集合 (組み合わせ) するとさらに複雑なクエリーを作成できます。以下の集合演算子を使用できます。

- **SHOULD**: クエリーにはサブクエリーの一致要素が含まれるはずです。
- **MUST**: クエリーにはサブクエリーの一致要素が含まれていなければなりません。

- **MUST NOT**: クエリーにはサブクエリーの一致要素が含まれていてはなりません。

サブクエリーはブール値クエリー自体を含む Lucene クエリーです。

例: SHOULD クエリー

```
//look for popular myths that are preferably urban
Query luceneQuery = mythQB
    .bool()
      .should(
mythQB.keyword().onField("description").matching("urban").createQuery() )
      .must( mythQB.range().onField("starred").above(4).createQuery() )
    .createQuery();
```

例: MUST クエリー

```
//look for popular urban myths
Query luceneQuery = mythQB
    .bool()
      .must(
mythQB.keyword().onField("description").matching("urban").createQuery() )
      .must( mythQB.range().onField("starred").above(4).createQuery() )
    .createQuery();
```

例: MUST NOT クエリー

```
//look for popular modern myths that are not urban
Date twentiethCentury = ...;
Query luceneQuery = mythQB
    .bool()
      .must(
mythQB.keyword().onField("description").matching("urban").createQuery() )
        .not()
      .must( mythQB.range().onField("starred").above(4).createQuery() )
      .must( mythQB
        .range()
        .onField("creationDate")
        .above(twentiethCentury)
        .createQuery() )
    .createQuery();
```

13.5.1.9. クエリーオプション

Hibernate Search クエリー DSL は簡単に使用および読み取りできるクエリー API です。Lucene クエリーを許可および作成すると、DSL によってサポートされていないクエリー型を取り入れることができます。

クエリー型およびフィールドのクエリーオプションの概要は次のとおりです。

- **boostedTo** (クエリー型およびフィールド上) はクエリー全体または特定のフィールドを指定の係数にブーストします。
- **withConstantScore** (クエリー上) は、定数スコア (constant score) がブーストと等しいクエリーに一致するすべての結果を返します。

- **filteredBy(Filter)** (クエリー上) は Filter インスタンスを使用してクエリー結果をフィルターします。
- **ignoreAnalyzer** (フィールド上) は、このフィールドの処理時にアナライザーを無視します。
- **ignoreFieldBridge** (フィールド上) は、このフィールドの処理時にフィールドブリッジを無視します。

例: クエリーオプションの組み合わせ

```
Query luceneQuery = mythQB
    .bool()
      .should(
mythQB.keyword().onField("description").matching("urban").createQuery() )
      .should( mythQB
        .keyword()
        .onField("name")
          .boostedTo(3)
          .ignoreAnalyzer()
        .matching("urban").createQuery() )
      .must( mythQB
        .range()
          .boostedTo(5).withConstantScore()
        .onField("starred").above(4).createQuery() )
    .createQuery();
```

13.5.1.10. Hibernate Search クエリーの構築

13.5.1.10.1. 一般論

Lucene クエリーを構築したら、Hibernate クエリー内でラップします。クエリーは、インデックス化されたエンティティーをすべて検索し、インデックス化されたクラスのすべての型を返します。この挙動を変更するには、明示的に設定する必要があります。

例: Hibernate クエリー内の Lucene クエリーのラッピング

```
FullTextSession fullTextSession = Search.getFullTextSession( session );
org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(
luceneQuery );
```

パフォーマンスを向上するには、以下のように戻り値の型を制限します。

例: エンティティー型での検索結果のフィルター

```
fullTextQuery = fullTextSession
    .createFullTextQuery( luceneQuery, Customer.class );

// or

fullTextQuery = fullTextSession
    .createFullTextQuery( luceneQuery, Item.class, Actor.class );
```

2 つ目の例の最初の部分は、一致する Customer のみを返します。同じ例の 2 つ目の部分は一致する Actor および Item を返します。型制限は多形です。このため、ベースクラス Person の 2 つのサブクラ

スである Salesman および Customer が返される場合は、Person.class を指定して結果の型に基づいてフィルターします。

13.5.1.10.2. ページネーション

パフォーマンスの劣化を防ぐため、クエリーごとに返されるオブジェクトの数を制限することが推奨されます。ユーザーがあるページから別のページへ移動するユースケースは大変一般的です。ページネーションを定義する方法は、プレーン HQL または Criteria クエリーでページネーションを定義する方法に似ています。

例: 検索クエリーに対するページネーションの定義

```
org.hibernate.Query fullTextQuery =
    fullTextSession.createFullTextQuery( luceneQuery, Customer.class );
fullTextQuery.setFirstResult(15); //start from the 15th element
fullTextQuery.setMaxResults(10); //return 10 elements
```

注記

`fulltextQuery.getResultSize()` を用いたページネーションに関係なく、一致する要素の合計数を取得できます。

13.5.1.10.3. 並び順

Apache Lucene には、柔軟で強力な結果ソートメカニズムが含まれています。デフォルトでは関連性でソートされます。他のプロパティーでソートするようソートメカニズムを変更するには、Lucene Sort オブジェクトを使用して Lucene ソートストラテジーを適用します。

例: Lucene Sort の指定

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( query,
Book.class );
org.apache.lucene.search.Sort sort = new Sort(
    new SortField("title", SortField.STRING));

List results = query.list();
```

注記

ソートに使用されるフィールドはトークン化しないでください。トークン化に関する詳細については、@Fieldを参照してください。

13.5.1.10.4. フェッチングストラテジー

戻り値の型が 1 つのクラスに制限される場合、Hibernate SearchHibernate Search は単一のクエリーを使用してオブジェクトをロードします。Hibernate SearchHibernate Search は、ドメインモデルに定義された静的なフェッチングストラテジーによって制限されます。以下のように、特定のユースケースに対してフェッチングストラテジーを絞り込むと便利です。

例: クエリーでの FetchMode の指定

```
Criteria criteria =
    s.createCriteria( Book.class ).setFetchMode( "authors", FetchMode.JOIN
);
```

```
s.createFullTextQuery( luceneQuery ).setCriteriaQuery( criteria );
```

この例では、クエリーは LuceneQuery に一致するすべての Book を返します。authors のコレクションは SQL の外部結合を使用して同じクエリーからロードされます。

Criteria クエリーの定義では、提供された Criteria クエリーを基に型が推測されます。そのため、戻り値のエンティティー型を制限する必要はありません。

重要

フェッチモードのみが調整可能なプロパティーです。Criteria とともに制限を使用すると、getResultSize() によって SearchException が発生するため、Criteria クエリーで制限 (where 句) を使用しないでください。

複数のエンティティーが想定される場合は、**`setCriteriaQuery`** を使用しないでください。

13.5.1.10.5. 射影 (Projection)

プロパティーの小さなサブセットのみが必要になることがあります。以下のように、Hibernate Search を使用してプロパティーのサブセットを返します。

Hibernate Search は Lucene インデックスからプロパティーを抽出し、オブジェクトの表現に変換して Object[] のリストを返します。射影により、時間がかかるデータベースラウンドトリップは回避されますが、以下の制約があります。

- 射影されたプロパティーはインデックス (**`@Field(store=Store.YES)`**) に保存される必要があります。これにより、インデックスのサイズが大きくなります。
- 射影されたプロパティーは org.hibernate.search.bridge.TwoWayFieldBridge または **`org.hibernate.search.bridge.TwoWayStringBridge`** (より単純なバージョン) を実装する **`FieldBridge`** を使用する必要があります。

  注記

  Hibernate Search の組み込み型はすべて双方向です。

- インデックス化されたエンティティーのシンプルなプロパティーまたは埋め込みされた関連のみを射影できます。埋め込みエンティティー全体は射影できません。
- 射影は、@IndexedEmbedded を用いてインデックス化されたコレクションまたはマップでは動作しません。

Lucene はクエリー結果のメタデータ情報を提供します。射影定数を使用してメタデータを読み出します。

例: 射影を使用したメタデータの読み出し

```
org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
query.;
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
```

```
float score = firstResult[0];
Book book = firstResult[1];
String authorName = firstResult[2];
```

フィールドは、以下の射影定数と組み合わせることができます。

- **FullTextQuery.THIS:**: 初期化された管理対象エンティティーを返します (射影されていないクエリーが行うとおり)。
- **FullTextQuery.DOCUMENT:**: 射影されたオブジェクトに関連する Lucene Document を返します。
- **FullTextQuery.OBJECT_CLASS:** インデックス化されたエンティティーのクラスを返します。
- **FullTextQuery.SCORE:** クエリーのドキュメントスコアを返します。スコアはある 1 つのクエリーの結果を特定のクエリーの別の結果と比較するのに便利ですが、さまざまなクエリーの結果を比較する場合は役に立ちません。
- **FullTextQuery.ID:** 射影されたオブジェクトの ID プロパティー値。
- **FullTextQuery.DOCUMENT_ID:** Lucene ドキュメント ID。Lucene のドキュメント ID は 2 つの異なる IndexReader が開かれる間に変更される可能性があるため、この値を使用する場合は注意してください。
- **FullTextQuery.EXPLANATION:** 該当するクエリーのオブジェクト/ドキュメントに一致する Lucene Explanation オブジェクトを返します。これは、大量のデータの読み出しには適していません。通常、Explanation の実行で使用するリソースの量は、一致する要素ごとに Lucene クエリー全体を実行する場合に匹敵します。そのため、射影が推奨されます。

13.5.1.10.6. オブジェクト初期化ストラテジーのカスタマイズ

デフォルトでは、Hibernate SearchHibernate Search は最も適切なストラテジーを使用してフルテキストクエリーに一致するエンティティーを初期化します。1 つまたは複数のクエリーを実行して、必要なエンティティーを読み出します。この方法により、永続コンテキスト (セッション) または 2 次キャッシュに読み出されたエンティティーがほとんど存在しない場合にデータベーストリップが最小化されます。

2 次キャッシュにエンティティーが存在する場合は、データベースオブジェクトを読み出す前にキャッシュをチェックするよう Hibernate SearchHibernate Search を強制します。

例: クエリー使用前の 2 次キャッシュのチェック

```
FullTextQuery query = session.createFullTextQuery(luceneQuery,
User.class);
query.initializeObjectWith(
    ObjectLookupMethod.SECOND_LEVEL_CACHE,
    DatabaseRetrievalMethod.QUERY
);
```

`ObjectLookupMethod` は、オブジェクトに簡単に (データベースからフェッチせずに) アクセスできるかどうかをチェックするストラテジーを定義します。他のオプションは次のとおりです。

- **`ObjectLookupMethod.PERSISTENCE_CONTEXT`** は、一致するエンティティーの多くがすでに永続コンテキストにロードされている場合に使用されます (Session または EntityManager にロードされます)。

- **`ObjectLookupMethod.SECOND_LEVEL_CACHE`** は永続コンテキストをチェックし、その後に 2 次キャッシュをチェックします。

以下を設定して 2 次キャッシュを検索します。

- 2 次キャッシュを適切に設定および有効にします。
- 関連するエンティティーに対して 2 次キャッシュを有効にします。これは、@Cacheable などのアノテーションを使用して行われます。
- Session、EntityManager、または Query に対する 2 次キャッシュの読み取りアクセスを有効にします。Hibernate ネイティブ API では **`CacheMode.NORMAL`** を使用し、Java Persistence API では **`CacheRetrieveMode.USE`** を使用します。

警告

2 次キャッシュの実装が EHCache または Infinispan でない場合は、ObjectLookupMethod.SECOND_LEVEL_CACHE を使用しないでください。他の 2 次キャッシュプロバイダーはこの操作を効率的に実装しません。

次のように **`DatabaseRetrievalMethod`** を使用して、データベースからオブジェクトがロードされる方法をカスタマイズします。

- **QUERY** (デフォルト値) はクエリーのセットを使用してバッチごとに複数のオブジェクトをロードします。この方法が推奨されます。
- **FIND_BY_ID** は **`Session.get`** または **`EntityManager.find`** セマンティックを使用して 1 度に 1 つずつオブジェクトをロードします。これは、Hibernate Core がバッチでエンティティーをロードできるようにバッチサイズがエンティティー用に設定されている場合に推奨されます。

13.5.1.10.7. クエリーの時間制限

次のように、Hibernate Guide でクエリーが要する時間を制限します。

- 制限に達したら例外を発生させます。
- 時間制限に達したら読み出された結果の数を制限します。

13.5.1.10.8. 時間制限での例外の発生

定義された時間を越えてクエリーが実行された場合、QueryTimeoutException が発生します (プログラム API に応じて org.hibernate.QueryTimeoutException または javax.persistence.QueryTimeoutException が発生します)。

ネイティブ Hibernate API を使用して制限を定義するには、以下の方法の 1 つを使用します。

例: クエリー例外でのタイムアウトの定義

```
Query luceneQuery = ...;
FullTextQuery query = fullTextSession.createFullTextQuery(luceneQuery,
```

```
User.class);

//define the timeout in seconds
query.setTimeout(5);

//alternatively, define the timeout in any given time unit
query.setTimeout(450, TimeUnit.MILLISECONDS);

try {
    query.list();
}
catch (org.hibernate.QueryTimeoutException e) {
    //do something, too slow
}
```

getResultSize(), iterate()、iterate()、および scroll() はメソッド呼び出しが終わるまでタイムアウトを考慮します。そのため、Iterable または ScrollableResults はタイムアウトを無視します。さらに explain() はこのタイムアウト期間に従いません。このメソッドはデバッグや、クエリーのパフォーマンスが遅い理由をチェックするために使用されます。

Java Persistence API (JPA) を使用して実行時間を制限する標準的な方法は次のとおりです。

例: クエリー例外でのタイムアウトの定義

```
Query luceneQuery = ...;
FullTextQuery query = fullTextEM.createFullTextQuery(luceneQuery,
User.class);

//define the timeout in milliseconds
query.setHint( "javax.persistence.query.timeout", 450 );

try {
    query.getResultList();
}
catch (javax.persistence.QueryTimeoutException e) {
    //do something, too slow
}
```

重要

サンプルコードは、クエリーが指定された結果の値で停止することを保証しません。

13.5.2. 結果の読み出し

Hibernate クエリーは構築後に HQL または Criteria クエリーと同じように実行されます。同じパラダイムとオブジェクトセマンティックが Lucene Query クエリーに適用され、**`list()`**、**`uniqueResult()`**、**`iterate()`**、**`scroll()`** などの一般的な操作を使用できます。

13.5.2.1. パフォーマンスに関する注意点

妥当な数の結果が予期され (ページネーションを使用する場合など)、すべての結果で作業を行うことが想定される場合、**`list()`** または **`uniqueResult()`** の使用が推奨されます。**`list()`** はエンティティー **`batch-size`** が適切に設定されている場合に最適です。**`list()`**、**`uniqueResult()`**、および

`iterate()` を使用する場合は Hibernate Search がすべての Lucene Hits 要素 (ページネーション内) を処理する必要があることに注意してください。

Lucene ドキュメントのローディングを最小限にしたい場合は、**`scroll()`** の使用が適しています。ScrollableResults オブジェクトは Lucene リソースを保持するため、終了後はこのオブジェクトを閉じるようにしてください。scroll の使用が想定される場合にオブジェクトを一括してロードするには、query.**`setFetchSize()`** を使用できます。ロードされていないオブジェクトがアクセスされた場合、Hibernate Search は次の **`fetchSize`** オブジェクトを 1 度にロードします。

重要

スクローリングよりもページネーションの使用が推奨されます。

13.5.2.2. 結果サイズ

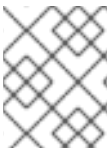
以下のような場合、一致するドキュメントの合計数が分かると便利です。

- Google 検索による検索結果の合計を表す機能を提供する場合 (例:「検索結果約 888,000,000 件中 1 - 10 件を表示」)
- 高速なページネーションのナビゲーションを実装する場合
- 制限されたクエリーがゼロを返すか、十分な結果を返さないときに近似値を追加する、複数ステップの検索エンジンを実装する場合

一致するドキュメントをすべて読み出すには大量のリソースを消費します。Hibernate Search では、ページネーションパラメーターに関係なく、一致するドキュメントの合計数を読み出しできます。さらに、オブジェクトのロードを発生させずに一致する要素の数を読み出しすることもできます。

例: クエリーの結果サイズの決定

```
org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
//return the number of matching books without loading a single one
assert 3245 == ;

org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
query.setMaxResult(10);
List results = query.list();
//return the total number of matching books regardless of pagination
assert 3245 == ;
```



注記

Google と同様に、インデックスがデータベースに対して完全に最新の状態になっていない場合は (非同期クラスターなど)、結果の数は近似値になります。

13.5.2.3. ResultTransformer

射影の結果は Object 配列として返されます。オブジェクトに使用されたデータ構造がアプリケーションの要件と一致しない場合は、ResultTransformer を適用します。ResultTransformer はクエリーの実行後に必要なデータ構造を構築します。

射影の結果は Object 配列として返されます。オブジェクトに使用されたデータ構造がアプリケーションの要件と一致しない場合は、ResultTransformer を適用します。ResultTransformer はクエリーの実行後に必要なデータ構造を構築します。

例: 射影への ResultTransformer の使用

```
org.hibernate.search.FullTextQuery query =
    s.createFullTextQuery( luceneQuery, Book.class );
query.setProjection( "title", "mainAuthor.name" );

query.setResultTransformer( new StaticAliasToBeanResultTransformer(
BookView.class, "title", "author" ) );
List<BookView> results = (List<BookView>) query.list();
for(BookView view : results) {
    log.info( "Book: " + view.getTitle() + ", " + view.getAuthor() );
}
```

ResultTransformer 実装の例は、Hibernate Core のコードベースにあります。

13.5.2.4. 結果の理解

クエリーの結果が想定外であった場合、結果を理解するには **Luke** ツールを使用すると便利です。しかし、 Hibernate Search では (特定クエリーの) 結果の Lucene Explanation オブジェクトへアクセスすることもできます。このクラスは、Lucene ユーザーにとっては非常に高度なクラスですが、オブジェクトのスコアを理解するのに便利です。特定の結果の Explanation オブジェクトへアクセスする方法は 2 つあります。

- **fullTextQuery.explain(int)** メソッドの使用
- 射影の使用

最初の方法は、ドキュメント ID をパラメーターとして取り、Explanation オブジェクトを返します。ドキュメント ID は射影および **FullTextQuery.DOCUMENT_ID** 定数を使用して読み出しできます。

警告

ドキュメント ID はエンティティー ID とは関係ありません。これらを混同しないようにしてください。

2 つ目の方法は、**FullTextQuery.EXPLANATION** 定数を使用して Explanation オブジェクトを射影します。

例: 射影を使用した Lucene Explanation オブジェクトの読み出し

```
FullTextQuery ftQuery = s.createFullTextQuery( luceneQuery, Dvd.class )
        .setProjection(
             FullTextQuery.DOCUMENT_ID,
             ,
             FullTextQuery.THIS );
@SuppressWarnings("unchecked") List<Object[]> results = ftQuery.list();
```

```
for (Object[] result : results) {
    Explanation e = (Explanation) result[1];
    display( e.toString() );
}
```

Explanation オブジェクトの使用で消費するリソースは Lucene クエリーを再実行するのとほぼ同じため、必要な場合のみ使用してください。

13.5.2.5. Filters

Apache Lucene には、カスタムのフィルター処理に従ってクエリーの結果をフィルターできる強力な機能が含まれています。これは、フィルターをキャッシュおよび再使用できるため、データの制限を追加で適用する大変強力な方法です。ユースケースには以下が含まれます。

- security
- 一時データ (例: 閲覧専用の先月のデータ)
- 入力 (population) フィルター (例: 指定のカテゴリーに限定される検索)

Hibernate Search では、さらに、透過的にキャッシュされるパラメーター化可能な名前付きフィルターの概念が導入されます。Hibernate Core フィルターの概念を知っている場合、その API は非常に似ています

例: クエリーに対するフルテキストフィルターの有効化

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("bestDriver");
fullTextQuery.enableFullTextFilter("security").setParameter( "login",
"andre" );
fullTextQuery.list(); //returns only best drivers where andre has
credentials
```

この例では、クエリー上で 2 つのフィルターが有効になっています。フィルターは好きなだけ (有効または無効) にできます。

フィルターの宣言は、@FullTextFilterDef アノテーションから行われます。このアノテーションは、フィルターが後で適用されるクエリーに関係なく、 **@Indexed** エンティティーである場合があります。これは、暗黙的に、フィルター定義がグローバルであり、その名前が一意である必要があることを示します。SearchException は、同じ名前で 2 つの異なる @FullTextFilterDef アノテーションが定義された場合にスローされます。各名前付きフィルターは実際のフィルター実装を指定する必要があります。

例: フィルターの定義および実装

```
@FullTextFilterDefs( {
    @FullTextFilterDef(name = "bestDriver", impl =
BestDriversFilter.class),
    @FullTextFilterDef(name = "security", impl =
SecurityFilterFactory.class)
})
public class Driver { ... }
```

```
public class BestDriversFilter extends org.apache.lucene.search.Filter {
```

```
    public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
        OpenBitSet bitSet = new OpenBitSet( reader.maxDoc() );
        TermDocs termDocs = reader.termDocs( new Term( "score", "5" ) );
        while ( termDocs.next() ) {
            bitSet.set( termDocs.doc() );
        }
        return bitSet;
    }
}
```

BestDriversFilter は、スコアが 5 のドライバーに結果セットを削減する単純な Lucene フィルターの例です。この例では、指定されたフィルターは **`org.apache.lucene.search.Filter`** を直接実装し、引数がないコンストラクターを含みます。

フィルターの作成で追加の手順が必要な場合、または使用するフィルターに引数がないコンストラクターが含まれない場合は、ファクトリーパターンを使用できます。

例: ファクトリーパターンを使用したフィルターの作成

```
@FullTextFilterDef(name = "bestDriver", impl =
BestDriversFilterFactory.class)
public class Driver { ... }

public class BestDriversFilterFactory {

@Factory
    public Filter getFilter() {
        //some additional steps to cache the filter results per
IndexReader
        Filter bestDriversFilter = new BestDriversFilter();
        return new CachingWrapperFilter(bestDriversFilter);
    }
}
```

Hibernate Search は、**`@Factory`** アノテーションが付いたメソッドを探し、そのメソッドを使用してフィルターインスタンスを構築します。ファクトリーには引数がないコンストラクターが含まれる必要があります。

Infinispan クエリーは @Factory アノテーションが付いたメソッドを使用してフィルターインスタンスを構築します。ファクトリーには引数がないコンストラクターが含まれる必要があります。

名前付きフィルターは、パラメーターをフィルターに渡す必要がある場合に便利です。たとえば、セキュリティーフィルターが、適用するセキュリティーレベルを認識する場合を考えてみます。

例: 定義されたフィルターにパラメーターを渡す

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("security").setParameter( "level", 5 );
```

各パラメーター名では、対象となる名前付きフィルター定義のフィルターまたはフィルターファクトリーのいずれかでセッターが関連付けられているとします。**例: 実際のフィルター実装でのパラメーターの使用**

```
public class SecurityFilterFactory {
    private Integer level;
```

```
    /**
     * injected parameter
     */
    public void setLevel(Integer level) {
        this.level = level;
    }

    @Key public FilterKey getKey() {
        StandardFilterKey key = new StandardFilterKey();
        key.addParameter( level );
        return key;
    }

    @Factory
    public Filter getFilter() {
        Query query = new TermQuery( new Term("level", level.toString() )
);
        return new CachingWrapperFilter( new QueryWrapperFilter(query) );
    }
}
```

@Key アノテーションが付いたメソッドは FilterKey オブジェクトを返すことに注意してください。返されたオブジェクトには特別なコントラクトがあります (キーオブジェクトは equals() / hashCode() を実装して、特定のフィルタータイプが同じであり、パラメーターセットが同じである場合のみ 2 つのキーが同じになるようにします)。つまり、2 つのフィルターキーは、キーが生成されるフィルターが交換可能である場合のみ同じになります。キーオブジェクトはキャッシュメカニズムでキーとして使用されます。

@Key メソッドは以下の場合のみ必要です。

- フィルターキャッシュシステムが有効である (デフォルトで有効)
- フィルターにパラメーターが含まれる

ほとんどの場合、**`StandardFilterKey`** 実装を使用するだけで十分です。これにより、equals() / hashCode() 実装はパラメーターの各 equals および hashcode メソッドに委譲されます。

これまでに説明したように、定義されたフィルターはデフォルトでキャッシュされ、キャッシュはハード参照とソフト参照の組み合わせを使用して必要な場合にメモリーの破棄を許可します。ハード参照キャッシュは最後に使用されたフィルターを追跡し、使用頻度が最も低いフィルターを必要に応じて SoftReferences に変換します。ハード参照キャッシュのサイズを調整するには、**`hibernate.search.filter.cache_strategy.size`** (デフォルト値は 128) を使用します。フィルターキャッシュの高度な使用については、独自の FilterCachingStrategy を実装してください。クラス名は **`hibernate.search.filter.cache_strategy`** によって定義されます。

このフィルターキャッシュメカニズムを実際のフィルター結果と混同しないでください。Lucene では、CachingWrapperFilter に IndexReader を使用してフィルターをラップすることが一般的です。このラッパーは、コストがかかる再計算を回避するために getDocIdSet(IndexReader リーダー) メソッドから返された DocIdSet をキャッシュします。リーダーは開いたときのインデックスの状態を表すため、計算される DocIdSet は同じ IndexReader インスタンスに対してのみキャッシュできることに注意してください。ドキュメントリストは開いた IndexReader 内で変更できません。ただし、別の/新しい IndexReader インスタンスがドキュメントの別のセット (別のインデックスのもの、またはインデックスが変更されたため) で動作することがあります。この場合、キャッシュされた DocIdSet は再計算する必要があります。

Hibernate Search は、キャッシュのこの側面でも役に立ちます。デフォルトでは、@FullTextFilterDef の **cache** フラグは **FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS** に設定され、フィルターインスタンスが自動的にキャッシュされ、指定されたフィルターが CachingWrapperFilter の Hibernate 固有の実装にラップされます。このクラスの Lucene のバージョンとは異なり、SoftReferences はハード参照数 (フィルターキャッシュに関する説明を参照) とともに使用されます。ハード参照数は、**hibernate.search.filter.cache_docidresults.size** (デフォルト値は 5) を使用して調整できます。ラップの動作は **@FullTextFilterDef.cache** パラメーターを使用して制御できます。このパラメーターには以下の 3 つの異なる値があります。

値	定義
FilterCacheModeType.NONE	フィルターインスタンスなしと結果なしは Hibernate Search によってキャッシュされます。フィルターの呼び出しごとに、新しいフィルターインスタンスが作成されます。この設定は、頻繁に変更するデータやメモリーの制約が大きい環境に役に立つことがあります。
FilterCacheModeType.INSTANCE_ONLY	フィルターインスタンスはキャッシュされ、同時 Filter.getDocIdSet() 呼び出しで再使用されます。この設定は、フィルターが独自のキャッシュメカニズムを使用する場合、またはフィルター結果が動的に変更される場合に役に立ちます (アプリケーション固有のイベントにより両方のケースで DocIdSet キャッシュが不必要になることが原因)。
FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS	フィルターインスタンスの結果と DocIdSet の結果の両方がキャッシュされます。これはデフォルト値です。

最後に、フィルターをキャッシュする理由について説明します。フィルターのキャッシュが必要になる状況は 2 つあります。

その状況は以下のとおりです。

- システムが対象となるエンティティーインデックスを頻繁に更新しない (つまり、IndexReader が頻繁に再利用される)
- フィルターの DocIdSet の計算のコストが高い (クエリーを実行するのにかかる時間と比較して)

13.5.2.6. シャード化された環境におけるフィルターの使用

シャード化された環境では、使用できるシャードのサブセットでクエリーを実行できます。これは、2 つのステップで行われます。

インデックスシャードのサブセットをクエリーする

1. フィルター設定に応じて、IndexManagers のサブセットを選択するシャードストラテジーを作成します。
2. クエリーの実行時にフィルターを有効にします。

例: インデックスシャードのサブセットをクエリーする

この例では、**customer** フィルターが有効になるとクエリーが特定のカスタマーシャードに対して実行されます。

```
public class CustomerShardingStrategy implements IndexShardingStrategy {

     // stored IndexManagers in an array indexed by customerID
     private IndexManager[] indexManagers;

     public void initialize(Properties properties, IndexManager[]
indexManagers) {
       this.indexManagers = indexManagers;
     }

     public IndexManager[] getIndexManagersForAllShards() {
       return indexManagers;
     }

     public IndexManager getIndexManagerForAddition(
         Class<?> entity, Serializable id, String idInString, Document
document) {
       Integer customerID =
Integer.parseInt(document.getFieldable("customerID").stringValue());
       return indexManagers[customerID];
     }

     public IndexManager[] getIndexManagersForDeletion(
         Class<?> entity, Serializable id, String idInString) {
       return getIndexManagersForAllShards();
     }

      /**
      * Optimization; don't search ALL shards and union the results; in
this case, we
      * can be certain that all the data for a particular customer Filter
is in a single
      * shard; simply return that shard by customerID.
      */
     public IndexManager[] getIndexManagersForQuery(
         FullTextFilterImplementor[] filters) {
       FullTextFilter filter = getCustomerFilter(filters, "customer");
       if (filter == null) {
         return getIndexManagersForAllShards();
       }
       else {
         return new IndexManager[] { indexManagers[Integer.parseInt(
           filter.getParameter("customerID").toString())] };
       }
     }

     private FullTextFilter getCustomerFilter(FullTextFilterImplementor[]
filters, String name) {
       for (FullTextFilterImplementor filter: filters) {
         if (filter.getName().equals(name)) return filter;
       }
       return null;
     }
    }
```

この例では、**customer** という名前のフィルターが存在する場合はこのカスタマー専用のシャードのみがクエリーされます。存在しない場合はすべてのシャードが返されます。指定のシャードストラテジーはパラメーターに応じて 1 つ以上のフィルターに反応します。

2 つ目の手順では、クエリー実行時にフィルターを有効にします。クエリーの後に Lucene の結果もフィルターする通常のフィルターですが、シャードストラテジーのみへ渡される特別なフィルターを使用できます (シャードストラテジーへ渡されないと無視されます)。

この機能を使用するには、フィルターの宣言時に ShardSensitiveOnlyFilter クラスを指定します。

```
@Indexed
@FullTextFilterDef(name="customer", impl=ShardSensitiveOnlyFilter.class)
public class Customer {
    ...
}

FullTextQuery query = ftEm.createFullTextQuery(luceneQuery,
Customer.class);
query.enableFulltextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")
List<Customer> results = query.getResultList();
```

ShardSensitiveOnlyFilter を使用する場合、Lacene フィルターを実装する必要はありません。シャード化された環境でクエリーの実行を迅速にするために、フィルターおよびこれらにフィルターに反応するシャードストラテジーの使用が推奨されます。

13.5.3. ファセット

ファセット検索 (faceted search) は、クエリーの結果を複数のカテゴリーに分割できるテクニックです。このカテゴリー化には、各カテゴリーのヒット数の計算や、ファセット (カテゴリー) を基にして検索結果をさらに制限する機能が含まれます。以下の例はファセットの例を示しています。検索結果のヒット数は 15 であり、ページの主要部分に表示されます。左側のナビゲーションバーには **Computers & Internet** カテゴリーと、サブカテゴリーの **Programming**、**Computer Science**、**Databases**、**Software**、**Web Development**、**Networking**、および **Home Computing** が表示されます。各サブカテゴリーには、メインの検索基準に一致し、それぞれのサブカテゴリーに属する本の数が表示されます。カテゴリー **Computers & Internet** のこのような分割は、具体的な検索ファセットの 1 つです。別のファセットの例としては、カスタマーレビューの平均が挙げられます。

ファセット検索では、クエリーの結果がカテゴリーに分割されます。カテゴリー化には、各カテゴリーのヒット数の計算が含まれ、これらのファセット (カテゴリー) に基づいて検索結果がさらに制限されます。以下の例では、主要なページにファセット検索の結果の 15 件が表示されています。

左側のナビゲーションバーは、カテゴリーとサブカテゴリーを示しています。各サブカテゴリーに対して、本の数は主要な検索基準に一致し、各サブカテゴリーに属します。カテゴリー Computers & Internet のこのような分割は具体的な検索ファセットの 1 つです。別のファセットの例としては、カスタマーレビューの平均が挙げられます。

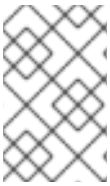
例: Amazon での Hibernate Search の検索

Hibernate Search では、QueryBuilder および FullTextQuery クラスがファセット API へのエントリーポイントになります。QueryBuilder はファセットリクエストを作成し、FullTextQuery は FacetManager へアクセスします。FacetManager はファセットリクエストをクエリーに適用し、検索結果を絞り込むために既存のクエリーへ追加されるファセットを選択します。例では、以下の例で示されたようにエンティティー Cd が使用されます。

例: エンティティー Cd

```
@Indexed
public class Cd {

    private int id;

    @Fields( {
        @Field,
        @Field(name = "name_un_analyzed", analyze = Analyze.NO)
    })
    private String name;

    @Field(analyze = Analyze.NO)
```

```
    @NumericField
    private int price;

    Field(analyze = Analyze.NO)
    @DateBridge(resolution = Resolution.YEAR)
    private Date releaseYear;

    @Field(analyze = Analyze.NO)
    private String label;


// setter/getter
...
```



注記

Hibernate Search 5.2 よりも前は、@Facet アノテーションを明示的に使用する必要がありませんでした。Hibernate Search 5.2 では、Lucene のネイティブファセット API を使用するために、@Facet アノテーションを明示的に使用することが必要になりました。

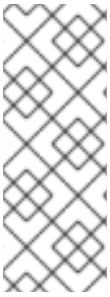
13.5.3.1. ファセットリクエストの作成

ファセット検索を行うための最初の手順は FacetingRequest を作成することです。現在、**離散ファセット (discrete faceting)** と**範囲ファセット (range faceting)** の 2 つの種類のファセットリクエストがサポートされています。離散ファセットリクエストの場合、ファセット (カテゴリー化) を使用したいインデックスフィールドと、適用するファセットオプションを指定します。離散ファセットリクエストの例を以下に示します。

例: 離散ファセットリクエストの作成

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
        .forEntity( Cd.class )
            .get();
FacetingRequest labelFacetingRequest = builder.facet()
    .name( "labelFaceting" )
    .onField( "label")
    .discrete()
    .orderedBy( FacetSortOrder.COUNT_DESC )
    .includeZeroCounts( false )
    .maxFacetCount( 1 )
    .createFacetingRequest();
```

このファセットリクエストを実行すると、インデックス化されたフィールド **`label`** の各離散値に対して Facet インスタンスが作成されます。Facet インスタンスは実際のフィールド値を記録します。この値には、元のクエリー結果内でこの特定のフィールド値が発生する頻度が含まれます。orderedBy、includeZeroCounts、および maxFacetCount はすべてのファセットリクエストに適用できる任意のパラメーターです。orderedBy を使用すると、作成されたファセットが返される順序を指定できます。デフォルト値は **`FacetSortOrder.COUNT_DESC`** ですが、フィールド値と範囲が指定された順番をソートすることもできます。includeZeroCount はカウント数が 0 のファセットが結果に含まれるかどうかを決定します (デフォルトでは含まれます)。maxFacetCount を使用すると返されるファセットの最大数を制限できます。



注記

現時点では、インデックス化されたフィールドにファセットを適用するにはそのフィールドが複数の事前条件を満たしている必要があります。インデックス化されたプロパティーは String 型、Date 型、または Number のサブタイプでなければなりません。**null** 値は使用しないようにしてください。さらに、プロパティーは **Analyze.NO** でインデックス化する必要があり、数値のプロパティーの場合は @NumericField を指定する必要があります。

範囲ファセットリクエストの作成は離散ファセットリクエストの作成と非常に似ていますが、ファセットを使用するフィールド値の範囲を指定する必要があります。3 つの異なる価格範囲が指定されている、範囲ファセットリクエストの例を以下に示します。**below** および **above** は 1 度だけ指定できますが、**from** - **to** の範囲は何度でも指定できます。範囲の境界に excludeLimit を使用して、範囲に含まれるかどうかを指定することもできます。

例: 範囲ファセットリクエストの作成

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
        .forEntity( Cd.class )
            .get();
FacetingRequest priceFacetingRequest = builder.facet()
    .name( "priceFaceting" )
    .onField( "price" )
    .range()
    .below( 1000 )
    .from( 1001 ).to( 1500 )
    .above( 1500 ).excludeLimit()
    .createFacetingRequest();
```

13.5.3.2. ファセットリクエストの適用

ファセットリクエストは、FullTextQuery クラスを用いて読み出しできる FacetManager クラスを使用して、クエリーに適用されます。

ファセットリクエストはいくつでも有効にでき、ファセットリクエスト名を指定して getFacets() で読み出しできます。ファセットリクエスト名を指定してそのファセットを無効にできる disableFaceting() メソッドもあります。

ファセットリクエストは、FullTextQuery を用いて読み出しできる FacetManager を使用して、クエリーに適用できます。

例: ファセットリクエストの適用

```
// create a fulltext query
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery(
luceneQuery, Cd.class );

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cds
List<Cd> cds = fullTextQuery.list();
```

```
...

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
...
```

複数のファセットリクエストは getFacets() を使用し、ファセットリクエスト名を指定することにより読み出しできます。

disableFaceting() メソッドは、ファセットリクエスト名を指定してファセットリクエストを無効にします。

13.5.3.3. クエリー結果の制限

最後に、「ドリルダウン」機能を実装するために、返された Facet のいずれかを元のクエリーの追加基準として適用できます。この場合は、FacetSelection を使用できます。FacetSelection は FacetManager から利用できます。これにより、クエリー基準としてのファセットの選択 (selectFacets)、単一のファセット制限の削除 (deselectFacets)、すべてのファセット制限の消去 (clearSelectedFacets)、および現在選択されているすべてのファセットの読み出し (getSelectedFacets) が可能になります。以下にコード例を示します。

```
// create a fulltext query
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery(
luceneQuery, clazz );

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cd
List<Cd> cds = fullTextQuery.list();
assertTrue(cds.size() == 10);

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
assertTrue(facets.get(0).getCount() == 2)

// apply first facet as additional search criteria
facetManager.getFacetGroup( "priceFaceting" ).selectFacets( facets.get( 0
) );

// re-execute the query
cds = fullTextQuery.list();
assertTrue(cds.size() == 2);
```

13.5.4. クエリー処理の最適化

クエリーのパフォーマンスは複数の基準に依存します。

- Lucene クエリー。
- ロードされたオブジェクトの数: ページネーション (常に使用) またはインデックス射影 (必要な場合) の使用。

- Hibernate Search が Lucene リーダーと対話する方法: 適切なリーダーストラテジーの定義。
- インデックスから頻繁に抽出される値のキャッシュ: インデックス値のキャッシュ: FieldCacheを参照。

13.5.4.1. インデックス値のキャッシュ: FieldCache

Lucene インデックスの主な機能はクエリーの一致を特定することです。クエリーが実行された後、有用な情報を抽出するために結果を分析する必要があります。通常、Hibernate Search は Class タイプと主キーを抽出する必要があります。

インデックスから必要な値を抽出すると、パフォーマンスに負担がかかります。負担が大変小さくて気がつかないこともありますが、キャッシュした方がよい場合もあります。

要件は使用される射影の種類によって異なります。クエリーコンテキストなどから推測できる場合、Class 型は必要ありません。

@CacheFromIndex アノテーションを使用すると、Hibernate Search が必要とする主なメタデータフィールドの異なるキャッシュ方法を試すことができます。

```
import static org.hibernate.search.annotations.FieldCacheType.CLASS;
import static org.hibernate.search.annotations.FieldCacheType.ID;

@Indexed
@CacheFromIndex( { CLASS, ID } )
public class Essay {
    ...
```

このアノテーションを使用して Class 型および ID をキャッシュすることはできません。

- **`CLASS`**: Hibernate Search は Lucene FieldCache を使用して、インデックスから Class 型を抽出するパフォーマンスを向上します。
  この値はデフォルトでは有効になっています。@CacheFromIndex アノテーションを指定しないと、この値が Hibernate Search によって適用されます。
- **`ID`**: 主識別子を抽出するとキャッシュが使用されます。パフォーマンスが最も優れたクエリーが提供されるはずですが、メモリーを大量に消費するため、パフォーマンスが低下する可能性があります。

注記

ウォームアップの後 (一部のクエリーを実行した後) に、パフォーマンスとメモリー消費の影響を計測します。フィールドキャッシュを有効にするとパフォーマンスが向上する可能性がありますが、向上しないこともあります。

FieldCache を使用するデメリットは 2 つあります。

- メモリーの使用量: これらのキャッシュはメモリーを大量に使用します。通常、CLASS キャッシュの要件は ID キャッシュよりも少なくなります。
- インデックスのウォームアップ: フィールドキャッシュを使用する場合、新しいインデックスまたはセグメントの最初のクエリーはキャッシュが有効になっていない場合よりも遅くなります。

クエリーによっては、クラス型が必要でない場合があります。このような場合、**CLASS** フィールドキャッシュを有効にしても使用されないことがあります。たとえば、単一のクラスを対象とする場合、戻り値はすべてその型になります (クエリーの実行ごとに評価されます)。

ID FieldCache を使用する場合、対象エンティティーの ID は (構築するすべてのブリッジとして) TwoWayFieldBridge を使用する必要があります。特定のクエリーにロードされるすべての型は ID にフィールド名を使用し、同じ型の ID を持つ必要があります (クエリーの実行ごとに評価されます）。

13.6. 手動によるインデックスの変更

Hibernate Core はデータベースに変更を適用するため、Hibernate Search はこのような変更を検出し、インデックスを自動的に更新します (EventListeners が無効になっている場合を除く)。バックアップをリストアしないとデータに影響するときなど、場合によっては Hibernate を使用せずにデータベースに変更が加えられることがあります。このような場合、Hibernate Search は手動インデックス API を公開して明示的にインデックスから単一のエンティティーを更新または削除します。さらに、データベース全体に対してインデックスを再構築したり、特定の型への参照をすべて削除したりします。

これらのメソッドは Lucene インデックスのみに影響し、変更はデータベースには適用されません。

13.6.1. インデックスへのインスタンスの追加

FullTextSession.index(T entity) を使用すると、特定のオブジェクトインスタンスを直接インデックスへ追加したり、更新したりできます。このエンティティーがすでにインデックス化されている場合は、インデックスが更新されます。インデックスへの変更は、トランザクションのコミット時のみに適用されます。

FullTextSession.index(T entity) を使用してオブジェクトまたはインスタンスをインデックスに直接追加します。インデックスは、エンティティーがインデックス化されるときに更新されます。Infinispan Query は、トランザクションのコミット時にインデックスに変更を適用します。

例: FullTextSession.index(T entity) を用いたエンティティーのインデックス化

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
Object customer = fullTextSession.load( Customer.class, 8 );
fullTextSession.index(customer);
tx.commit(); //index only updated at commit time
```

1 つの型またはインデックス化されたすべての型のすべてのインスタンスを追加したい場合は、MassIndexer を使用する方法が推奨されます。

MassIndexer を使用して 1 つの型 (またはインデックス化されたすべての型) を追加します。詳細については、MassIndexer の使用を参照してください。

13.6.2. インデックスからのインスタンスの削除

データベースから物理的に削除しなくても、指定の型の 1 つまたはすべてのエンティティーを Lucene インデックスから削除できます。この操作はパージと呼ばれ、**`FullTextSession`** を使用して実行されます。

パージ操作により、Lucene インデックスからある型の 1 つのエンティティーまたはすべてのエンティティーを削除できます (データベースから物理的に削除する必要はありません)。この操作は FullTextSession を使用して実行されます。

例: インデックスから 1 つのエンティティーの特定インスタンスをパージする

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
fullTextSession.purgeAll( Customer.class );
//optionally optimize the index
//fullTextSession.getSearchFactory().optimize( Customer.class );
tx.commit(); //index is updated at commit time
```

このような操作の後にインデックスを最適化することが推奨されます。

注記

FullTextEntityManager では index、purge、および purgeAll メソッドも利用できます。

注記

すべての手動インデックス化メソッド (index、purge、および purgeAll) はインデックスのみに影響し、データベースには影響しませんが、トランザクションに対応するためトランザクションが正常にコミットされるまで (または flushToIndexes が使用されるまで) 適用されません。

13.6.3. インデックスの再構築

インデックスへのエンティティーマッピングを変更するとき、ほとんどの場合でインデックス全体を更新する必要があります。たとえば、異なるアナライザーを使用して既存のフィールドをインデックス化する場合は、影響のある型のインデックスを再構築する必要があります。また、データベースが置き換えられた場合 (バックアップからリストアされたり、レガシーシステムからインポートされた場合など) は、既存データからインデックスを再構築できるようにしたいことがあります。Hibernate Search には 2 つのメインストラテジーがあります。

インデクサーのエンティティーマッピングを変更するには、インデックス全体を更新する必要がある場合があります。異なるアナライザーを使用して既存のフィールドをインデックス化する場合は、影響を受ける型に対してインデックスを再構築する必要があります。

また、バックアップから復元するか、レガシーシステムからインポートすることによりデータベースが置き換えられた場合は、既存のデータからインデックスを再構築する必要があります。Infinispan Query は 2 つの主要なストラテジーを提供します。

- すべてのエンティティーで **`FullTextSession.index()`** を使用し、**`FullTextSession.flushToIndexes()`** を定期的に使用する。
- **`MassIndexer`** を使用する。

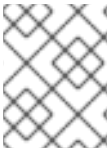
13.6.3.1. flushToIndexes() の使用

このストラテジーでは、既存のインデックスを削除した後、**`FullTextSession.purgeAll()`** および **`FullTextSession.index()`** を使用してすべてのエンティティーをインデックスに戻しますが、メモリーと効率性の制約があります。効率性を最大化にするために、Hibernate Search はインデックスの操作を一括してコミット時に実行します。大量のデータをインデックス化することが想定される場合は、トラザクションがコミットされるまですべてのドキュメントがキューに保持されるため、メモリーの消費に注意する必要があります。キューを周期的に空にしないと **`OutOfMemoryException`** が発生する可能性があります。キューを空にするには、**`fullTextSession.flushToIndexes()`** を使用しま

す。**fullTextSession.flushToIndexes()** が呼び出されるたびに (またはトランザクションがコミットされると) バッチキューが処理され、インデックスのすべての変更が適用されます。1 度フラッシュすると変更をロールバックできないことに注意してください。

例: index() および flushToIndexes() を使用したインデックスの再構築

```
fullTextSession.setFlushMode(FlushMode.MANUAL);
fullTextSession.setCacheMode(CacheMode.IGNORE);
transaction = fullTextSession.beginTransaction();
//Scrollable results will avoid loading too many objects in memory
ScrollableResults results = fullTextSession.createCriteria( Email.class )
    .setFetchSize(BATCH_SIZE)
    .scroll( ScrollMode.FORWARD_ONLY );
int index = 0;
while( results.next() ) {
    index++;
    fullTextSession.index( results.get(0) ); //index each element
    if (index % BATCH_SIZE == 0) {
        fullTextSession.flushToIndexes(); //apply changes to indexes
        fullTextSession.clear(); //free memory since the queue is
processed
    }
}
transaction.commit();
```



注記

この明示的な API はより優れた制御機能を提供するため、**hibernate.search.default.worker.batch_size** は廃止されました。

アプリケーションがメモリー不足にならないバッチサイズを使用するようにしてください。バッチサイズを大きくするとデータベースからオブジェクトをフェッチする速度が速くなりますが、より多くのメモリーが必要になります。

13.6.3.2. MassIndexer の使用

Hibernate Search の MassIndexer は複数の平行スレッドを使用してインデックスを再構築します。オプションで、リロードが必要なエンティティーを選択したり、すべてのエンティティーを再インデックス化したりできます。この方法はパフォーマンスを最大化するために最適化されますが、アプリケーションをメンテナンスモードに設定する必要があります。MassIndexer がビジー状態のときはインデックスのクエリーは推奨されません。

例: MassIndexer を使用したインデックスの再構築

```
fullTextSession.createIndexer().startAndWait();
```

これは、インデックスを再構築して削除し、データベースからすべてのエンティティーをリロードします。これは簡単に使用できますが、プロセスを迅速にするために多少の調整を行うことが推奨されます。

警告

MassIndexer の実行中はインデックスの内容が未定義になります。MassIndexer の動作中にクエリーが実行されると、結果の一部が失われる可能性があります。

例: 調整された MassIndexer の使用

```
fullTextSession
 .createIndexer( User.class )
 .batchSizeToLoadObjects( 25 )
 .cacheMode( CacheMode.NORMAL )
 .threadsToLoadObjects( 12 )
 .idFetchSize( 150 )
 .progressMonitor( monitor ) //a MassIndexerProgressMonitor implementation
 .startAndWait();
```

この例では、すべての User インスタンス (およびサブタイプ) のインデックスが再構築され、クエリーごとに 25 個のオブジェクトのバッチを使用して User インスタンスをロードするために 12 個の平行スレッドが作成されます。Lucene ドキュメントを出力するために、これらの 12 個のスレッドは、インデックス化された埋め込み関係と、カスタムの **FieldBridges** または **ClassBridges** を処理する必要もあります。スレッドによって、変換の処理中に追加属性のレイジーローディングがトリガーされます。そのため、平行して動作するスレッドが大量に必要になります。実際のインデックスの書き込みを行うスレッドの数は、各インデックスのバックエンド設定によって定義されます。

インデックス化を行うほとんどの場合で、キャッシュは不必要なオーバーヘッドになるため、cacheMode を **CacheMode.IGNORE** (デフォルト) のままにしておくことが推奨されます。メインエントリーがインデックスに含まれる列挙に似たデータである場合は、データに応じて他の **CacheMode** の一部を有効にするとパフォーマンスが向上する可能性があります。

注記

最良のパフォーマンスを実現するスレッド数は、全体のアーキテクチャー、データベースの設計、およびデータの値に大きく依存します。内部スレッドグループの名前はすべて意味があるため、threaddumps などの解析ツールを使用すれば簡単に特定できるはずです。

注記

MassIndexer はトランザクションを認識しないため、トランザクションを開始したりコミットする必要はありません。トランザクションではないため、処理中にユーザーによるシステムの使用を許可しないことが推奨されます。処理中にユーザーが結果を見つけられる可能性は低く、システムの負荷が大変高くなる可能性があります。

インデックス化の時間やメモリーの消費に影響するその他のパラメーターは次のとおりです。

- **hibernate.search.[default|<indexname>].exclusive_index_use**
- **hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs**
- **hibernate.search.[default|<indexname>].indexwriter.max_merge_docs**

- **hibernate.search.[default|<indexname>].indexwriter.merge_factor**
- **hibernate.search.[default|<indexname>].indexwriter.merge_min_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_max_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_max_optimize_size**
- **hibernate.search.[default|<indexname>].indexwriter.merge_calibrate_by_deletes**
- **hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size**
- **hibernate.search.[default|<indexname>].indexwriter.term_index_interval**

以前のバージョンでは **max_field_length** を使用できましたが、これは Lucene から削除されました。**LimitTokenCountAnalyzer** を使用すると同様の効果を得ることができます。

.indexwriter はすべて Lucene 固有のパラメーターで、Hibernate Search はこれらのパラメーターを渡します。

MassIndexer は、前方のみスクロール可能な結果を使用して、ロードされる主キーで反復処理を行いますが、MySQL の JDBC ドライバーはメモリーのすべての値をロードします。この「最適化」が実行されないようにするには、**idFetchSize** を **Integer.MIN_VALUE** に設定します。

13.7. インデックスの最適化

Lucene インデックスは時々最適化する必要があります。基本的にデフラグメンテーションを行います。最適化がトリガーされるまで、Lucene は削除されたドキュメントのみをマーク付けするため、物理的な削除は適用されません。最適化処理中に削除が適用され、Lucene Directory のファイル数にも影響します。

Lucene インデックスの最適化により検索速度は向上しますが、インデックス化 (更新) のパフォーマンスには影響ありません。最適化中に検索を実行できますが、ほとんどの場合、検索速度が遅くなります。インデックスの更新はすべて停止されます。以下の場合に最適化をスケジュールすることが推奨されます。

Lucene インデックスの最適化により検索は短縮化されますが、インデックス更新のパフォーマンスには影響ありません。最適化中に検索を実行できますが、検索の処理は遅くなります。最適化中にインデックスの更新はすべて停止されます。したがって、以下の場合に最適化をスケジュールすることが推奨されます。

- アイドルシステム上、または検索の頻度が最も低い場合。
- インデックスに多くの変更が加えられた後。

MassIndexer (MassIndexer の使用を参照) はデフォルトで処理の最初と最後にインデックスを最適化します。このデフォルトの挙動を変更するには、**MassIndexer.optimizeAfterPurge** および **MassIndexer.optimizeOnFinish** を使用します。

13.7.1. 自動最適化

Hibernate Search は、以下のいずれかを行った後にインデックスを自動的に最適化します。

以下の実行後に、Infinispan Query によりインデックスが自動的に最適化されます。

- 一定量の操作 (挿入または削除)。
- 一定量のトランザクション。

インデックスの自動最適化の設定は、グローバルまたはインデックスごとに定義できます。

定義: 自動最適化パラメーターの定義

```
hibernate.search.default.optimizer.operation_limit.max = 1000
hibernate.search.default.optimizer.transaction_limit.max = 100
hibernate.search.Animal.optimizer.transaction_limit.max = 50
```

以下のいずれかが発生すると、即座に最適化が **`Animal`** インデックスへトリガーされます。

- 追加または削除の数が **`1000`** に達した場合。
- トランザクションの数が **`50`**に達した場合 (**`hibernate.search.Animal.optimizer.transaction_limit.max`** は **`hibernate.search.default.optimizer.transaction_limit.max`** よりも優先されます)。

これらのパラメーターがすべて未定義であると、最適化は自動的に処理されません。

OptimizerStrategy のデフォルト実装をオーバーライドするには、**`org.hibernate.search.store.optimization.OptimizerStrategy`** を実装し、**`optimizer.implementation`** プロパティーを実装の完全修飾名に設定します。この実装はインターフェースを実装する必要があります。またパブリッククラスである必要があり、引数を取らないパブリックコンストラクターを持つ必要があります。

例: カスタム OptimizerStrategy のロード

```
hibernate.search.default.optimizer.implementation =
com.acme.worlddomination.SmartOptimizer
hibernate.search.default.optimizer.SomeOption = CustomConfigurationValue
hibernate.search.humans.optimizer.implementation = default
```

キーワード **`default`** を使用して Hibernate Search のデフォルト実装を選択できます。**`.optimizer`** キーセパレーターの後のすべてのプロパティーは、最初に実装の initialize メソッドへ渡されます。

13.7.2. 手動の最適化

SearchFactory を用いると、プログラミングによって Hibernate Search から Lucene インデックスを最適化 (デフラグメンテーション) できます。

例: プログラミングによるインデックスの最適化

```
FullTextSession fullTextSession =
Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();

searchFactory.optimize(Order.class);
// or
searchFactory.optimize();
```

最初の例は Orders を保持する Lucene インデックスを最適化し、2 番目の例はすべてのインデックスを最適化します。

注記

`searchFactory.optimize()` は JMS バックエンドには影響しません。最適化操作はマスターノードに適用する必要があります。

`searchFactory.optimize()` は、JMC バックエンドに影響を与えないため、マスターノードに適用されます。

13.7.3. 最適化の調整

Apache Lucene には最適化が実行される方法に影響するパラメーターが含まれています。Hibernate Search はこれらのパラメーターを公開します。

その他のインデックス最適化のパラメーターには以下が含まれます。

- **`hibernate.search.[default|<indexname>].indexwriter.max_buffered_docs`**
- **`hibernate.search.[default|<indexname>].indexwriter.max_merge_docs`**
- **`hibernate.search.[default|<indexname>].indexwriter.merge_factor`**
- **`hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size`**
- **`hibernate.search.[default|<indexname>].indexwriter.term_index_interval`**

13.8. 高度な機能

13.8.1. SearchFactory へのアクセス

SearchFactory オブジェクトは、Hibernate Search の基礎となる Lucene リソースを追跡します。これは、ネイティブで Lucene へアクセスするのに便利な方法です。**`SearchFactory`** は FullTextSession からアクセスできます。

例: SearchFactory へのアクセス

```
FullTextSession fullTextSession =
Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

13.8.2. IndexReader の使用

Lucene のクエリーは IndexReader 上で実行されます。Hibernate Search はパフォーマンスを最大限にするためにインデックスリーダーをキャッシュしたり、 更新された IndexReader を最小化する I/O 操作を読み出す効率的な他のストラテジーを提供したりできます。コードはこのようなキャッシュされたリソースへアクセスできますが、複数の要件があります。

例: IndexReader へのアクセス

```
IndexReader reader =
searchFactory.getIndexReaderAccessor().open(Order.class);
```

```
try {
   //perform read-only operations on the reader
}
finally {
   searchFactory.getIndexReaderAccessor().close(reader);
}
```

この例では、SearchFactory はこのエンティティーをクエリーするために必要なインデックスを決定します (シャードストラテジーを考慮します)。設定された ReaderProvider を各インデックスで使用すると、関係するすべてのインデックスの他に複合の **`IndexReader`** が返されます。この IndexReader は複数のクライアントで共有されるため、以下のルールに従う必要があります。

- indexReader.close() は呼び出さないでください。必要な場合は readerProvider.closeReader(reader) を使用しますが、finally ブロックで使用することが推奨されます。
- この IndexReader は変更操作では使用しないでください (読み取り専用の IndexReader で、変更操作での使用を試みると例外が発生します)。

これらのルールを守れば、IndexReader を自由に使用できます (特にネイティブ Lucene クエリーを実行する場合)。シャード化された IndexReaders を使用すると、ほとんどのクエリーはファイルシステムなどから直接開くよりも効率的になります。

open(Class… types) メソッドの代わりに、open(String… indexNames) を使用すると 1 つ以上のインデックス名を渡すことができます。このストラテジーを使用すると、シャードが使用されている場合にインデックス化された型のインデックスのサブセットも選択できます。

例: インデックス名による IndexReader へのアクセス

```
IndexReader reader =
searchFactory.getIndexReaderAccessor().open("Products.1", "Products.3");
```

13.8.3. Lucene Directory へのアクセス

Directory は、インデックスのストレージを表すために Lucene によって使用される最も一般的な抽象です。Hibernate Search は直接 Lucene Directory と対話しませんが、IndexManager を使用してこれらの対話を抽象化します。インデックスは Directory によって実装する必要はありません。

インデックスが Directory として表されていることを認識している場合にそのインデックスにアクセスするには、IndexManager を使用して Directory への参照を取得します。IndexManager を DirectoryBasedIndexManager へキャストし、**`getDirectoryProvider().getDirectory()`** を使用して基盤の Directory への参照を取得します。 IndexReader の使用が推奨され、この方法は推奨されません。

13.8.4. インデックスのシャード化

場合によっては、該当するエンティティーのインデックスデータを複数の Lucene インデックスに分割 (シャード化) することが役に立つことがあります。

警告

シャード化は、利点の方が欠点よりも多い場合にのみ実行してください。各検索に対してすべてのシャードをオープンにする必要があるため、通常、シャード化されたインデックスの検索には時間がかかります。

シャード化のユースケースは以下のとおりです。

- 単一のインデックスは非常に大きいため、インデックスの更新に時間がかかり、アプリケーションが低速になります。
- 通常の検索では、インデックスのサブセットのみがヒットされます (データが顧客、地域、またはアプリケーションにより、自然にセグメント化された場合など)。

デフォルトでは、シャード化はシャードの数が設定されていない限り有効になりません。これを行う場合は、**`hibernate.search.<indexName>.sharding_strategy.nbr_of_shards`** プロパティーを使用してください。

例: インデックスのシャード化の有効化 この例では、5 つのシャードが有効になります。

```
hibernate.search.<indexName>.sharding_strategy.nbr_of_shards = 5
```

データをサブインデックスに分割するには IndexShardingStrategy を使用します。デフォルトのシャード化ストラテジーでは、ID 文字列表現のハッシュ値 (FieldBridge により生成されます) に従ってデータが分割されます。これにより、調整されたシャード化が保証されます。デフォルトストラテジーは、カスタム IndexShardingStrategy を実装することにより置き換えることができます。カスタムストラテジーを使用するには、**`hibernate.search.<indexName>.sharding_strategy`** プロパティーを設定する必要があります。

例: カスタムシャード化ストラテジーの指定

```
hibernate.search.<indexName>.sharding_strategy =
my.shardingstrategy.Implementation
```

IndexShardingStrategy プロパティーを使用すると、クエリーを実行するシャードを選択して、検索を最適化することもできます。フィルターをアクティブ化することにより、シャード化ストラテジーでクエリー (IndexShardingStrategy.getIndexManagersForQuery) の応答に使用するシャードのサブセットを選択し、クエリーの実行を高速化できます。

各シャードには独立した IndexManager が存在し、異なるディレクトリープロバイダーおよびバックエンド設定を使用するよう設定できます。以下の例の Animal エンティティーの IndexManager インデックス名は **`Animal.0`** から **`Animal.4`** です。つまり、各シャードの独自のインデックスの名前の後に **`.`** (ドット) とインデックス番号が続きます。

例: エンティティー Animal のシャード化設定

```
hibernate.search.default.indexBase = /usr/lucene/indexes
hibernate.search.Animal.sharding_strategy.nbr_of_shards = 5
hibernate.search.Animal.directory_provider = filesystem
hibernate.search.Animal.0.indexName = Animal00
hibernate.search.Animal.3.indexBase = /usr/lucene/sharded
```

```
hibernate.search.Animal.3.indexName = Animal03
```

上記の例では、デフォルトの id 文字列ハッシュストラテジーが使用され、Animal インデックスが 5 サブインデックスにシャード化されます。すべてのサブインデックスはファイルシステムインスタンスであり、各サブインデックスが格納されるディレクトリーは以下のようになります。

- サブインデックス 0 の場合: **`/usr/lucene/indexes/Animal00`** (共有された indexBase、オーバーライドされた indexName)
- サブインデックス 1 の場合: **`/usr/lucene/indexes/Animal.1`** (共有された indexBase、デフォルトの indexName)
- サブインデックス 2 の場合: **`/usr/lucene/indexes/Animal.2`** (共有された indexBase、デフォルトの indexName)
- サブインデックス 3 の場合: **`/usr/lucene/shared/Animal03`** (オーバーライドされた indexBase、オーバーライドされた indexName)
- サブインデックス 4 の場合: **`/usr/lucene/indexes/Animal.4`** (共有された indexBase、デフォルトの indexName)

IndexShardingStrategy を実装する場合は、任意のフィールドを使用してシャード化の選択を決定できます。削除を処理するために (**`purge`** および **`purgeAll`** 操作)、実装がすべてのフィールド値またはプライマリー ID を読み取らずに 1 つまたは複数のインデックスを返す必要があることがあります。この場合は、単一のインデックスを取得するのに十分な情報が存在せず、すべてのインデックスを返す必要があります。この結果、削除操作が、削除するドキュメントを含むすべてのインデックスに伝播されます。

13.8.5. Lucene のスコア計算式のカスタマイズ

org.apache.lucene.search.Similarity を拡張すると Lucene のスコア計算式をカスタマイズできます。このクラスに定義される抽象メソッドは、ドキュメント d に対してクエリー q のスコアを算出する以下の計算式の係数に一致します。

org.apache.lucene.search.Similarity を拡張して Lucene のスコア計算式をカスタマイズします。以下のように、抽象メソッドは、ドキュメント **d** に対してクエリー **q** のスコアを算出するのに使用される計算式に一致します。

```
*score(q,d) = coord(q,d) · queryNorm(q) · ∑ ~t in q~ ( tf(t in d) ·
idf(t) ^2^ · t.getBoost() · norm(t,d) )*
```

係数	説明
tf(t ind)	文書 (d) の単語 (t) に対する単語頻度係数 (term frequency factor)。
idf(t)	単語の逆文書頻度 (inverse document frequency)。
coord(q,d)	指定の文書で見つかったクエリー対象の単語の数を基にしたスコア係数。
queryNorm(q)	クエリー間のスコアを比較できるようにするため使用される正規化係数 (normalizing factor)。

係数	説明
t.getBoost()	フィールドブースト。
norm(t,d)	一部の (インデックス化時間) ブースト係数と長さ係数をカプセル化します。

この計算式の詳細な説明は本書の範囲外になります。詳細については、Similarity の Javadoc を参照してください。

Hibernate Search では、Lucene の類似度計算を変更する方法は 3 つあります。

プロパティー **hibernate.search.similarity** を使用して Similarity 実装の完全指定されたクラス名を指定すると、デフォルトの類似度を設定できます。デフォルト値は org.apache.lucene.search.DefaultSimilarity です。

similarity プロパティーを設定して、特定のインデックスに使用される類似度をオーバーライドすることもできます。

```
hibernate.search.default.similarity = my.custom.Similarity
```

@Similarity アノテーションを使用して、クラスレベルでデフォルトの類似度をオーバーライドすることもできます。

```
@Entity
@Indexed
@Similarity(impl = DummySimilarity.class)
public class Book {
...
}
```

例として、文書で言葉が出現する頻度は重要でないと仮定しましょう。言葉が 1 度だけ出現する文書のスコアは、言葉が複数回出現する文書と同じになります。この場合、tf(float freq) メソッドのカスタム実装は 1.0 を返す必要があります。

警告

2 つのエンティティーが同じインデックスを共有する場合は、同じ Similarity 実装を宣言する必要があります。同じクラス階層のクラスは常にインデックスを共有するため、サブタイプの Similarity 実装をオーバーライドできません。

同様に、インデックス設定とクラスレベルの設定は競合するため、これらの設定で類似度を設定しても意味がなく、拒否されます。

13.8.6. 例外処理の設定

Hibernate Search では、インデックスの作成中に例外をどのように処理するかを設定できます。設定が提供されない場合は、デフォルトで例外がログ出力に記録されます。以下のように、例外ロギングメカニズムを明示的に宣言できます。

```
hibernate.search.error_handler = log
```

デフォルトの例外処理は、同期と非同期のインデックス作成両方で実行されます。Hibernate Search は、デフォルトのエラー処理実装をオーバーライドする簡単なメカニズムを提供します。

独自の実装を提供するには、**handle(ErrorContext context)** メソッドを提供する ErrorHandler インターフェースを実装する必要があります。**ErrorContext** は、プライマリー **LuceneWork** インスタンス、基礎となる例外、およびプライマリー例外が原因で処理できなかった後続の **LuceneWork** インスタンスへの参照を提供します。

```
public interface ErrorContext  {
   List<LuceneWork> getFailingOperations();
   LuceneWork getOperationAtFault();
   Throwable getThrowable();
   boolean hasErrors();
}
```

Hibernate Search でこのエラー処理を登録するには、設定プロパティーで ErrorHandler 実装の完全修飾クラス名を宣言する必要があります。

```
hibernate.search.error_handler = CustomerErrorHandler
```

13.8.7. Hibernate Search の無効化

Hibernate Search は、必要に応じて部分的または完全に無効にできます。インデックスが読み取り専用の場合、またはインデックス作成を自動的ではなく手動で実行する場合は、Hibernate Search のインデックス作成を無効にできます。また、Hibernate Search を完全に無効にしてインデックス作成と検索を回避することもできます。

インデックス作成の無効化

Hibernate Search インデックス作成を無効にするには、**indexing_strategy** 設定オプションを **manual** に変更し、JBoss EAP を再起動します。

```
hibernate.search.indexing_strategy = manual
```

Hibernate Search の完全な無効化

Hibernate Search を完全に無効にするには、**autoregister_listeners** 設定オプションを **false** に変更してすべてのリスナーを無効にし、JBoss EAP を再起動します。

```
hibernate.search.autoregister_listeners = false
```

13.9. モニタリング

Hibernate Search は、**SearchFactory.getStatistics()** を介して **Statistics** オブジェクトへのアクセスを提供します。たとえば、インデックスを作成するクラスやインデックスの格納するエントリーの数を決定できます。この情報は常に利用可能です。ただし、設定で

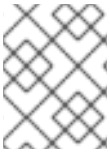
hibernate.search.generate_statistics プロパティーを指定することにより、Lucene クエリーおよびオブジェクトのロードのタイミングの合計と平均を収集することもできます。

JMX を介した統計へのアクセス

JMX を介した統計へのアクセスを有効にするには、プロパティー **hibernate.search.jmx_enabled** を **true** に設定します。これにより、**StatisticsInfoMBean** Bean が自動的に登録され、**Statistics** オブジェクトを介した統計へのアクセスが提供されます。設定に応じて、**IndexingProgressMonitorMBean** Bean を登録することもできます。

インデックス作成の監視

一括インデクサー API が使用されている場合は、**IndexingProgressMonitorMBean** Beanを介してインデックス作成の進捗を監視できます。インデックスの作成中、この Bean は JMX にのみバインドされます。



注記

JMX Bean は、システムプロパティー **com.sun.management.jmxremote** を **true** に設定することにより JConsole を使用してリモートでアクセスできます。

第14章 BEAN の検証

14.1. BEAN VALIDATION

Bean Validation あるいは JavaBeans Validation は、Java オブジェクトのデータを検証するモデルです。このモデルでは、組み込みのカスタムアノテーション制約を使い、アプリケーションデータの整合性を保ちます。この仕様は JSR 349: Bean Validation 1.1 で文書化されています。

Hibernate Validator は Bean Validation の JBoss EAP 実装であり、JSR の参照実装でもあります。

JBoss EAP は JSR 349 Bean Validation 1.1 仕様に完全準拠しています。また、Hibernate Validator によってこの仕様に追加機能が提供されます。

Bean Validation を初めて使用する場合は、JBoss EAP に同梱された **`bean-validation`** クイックスタートを参照してください。クイックスタートをダウンロードし、実行する方法については、クイックスタートサンプルの使用を参照してください。

14.2. バリデーション制約

14.2.1. バリデーション制約

バリデーション制約とは、フィールド、プロパティー、Bean などの Java 要素に適用するルールのことです。制約は通常、制限を設定する際に利用する一連の属性です。定義済みの制約がありますが、カスタムの制約も作成可能です。各制約は、アノテーション形式で表されます。

Hibernate Validator 用の同梱のバリデーション制約は、Hibernate Validator の制約にリストされています。

14.2.2. Red Hat JBoss Developer Studio での制約アノテーションの作成

概要

このタスクでは、Java アプリケーション内で使用できるように、Red Hat JBoss Developer Studio で制約アノテーションを作成するプロセスを説明します。

前提条件

1. Red Hat JBoss Developer Studio で Java プロジェクトを開きます。
2. **データセットを作成します。**
   制約アノテーションには、許容値を定義するデータセットが必要です。
   a. **Project Explorer (プロジェクトエクスプローラー)** パネルでプロジェクトルートフォルダーを右クリックします。
   b. **New** → **Enum** を選択します。
   c. 以下の要素を設定してください。
      - パッケージ:
      - 名前:
   d. **Add...** ボタンをクリックして必要なインターフェースを追加します。

e. **Finish** をクリックしてファイルを作成します。

f. 値セットをデータセットに追加し、**Save** をクリックします。

データセットの例

```
package com.example;

public enum CaseMode {
    UPPER,
    LOWER;
}
```

3. **アノテーションファイルを作成します。**
新しい Java クラスを作成します。

4. 制約アノテーションを設定し、**Save** をクリックします。
制約アノテーションファイルの例

```
package com.mycompany;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
public @interface CheckCase {

    String message() default "
{com.mycompany.constraints.checkcase}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    CaseMode value();

}
```

結果

許容値のあるカスタムの制約アノテーションが作成され、Java プロジェクトで使用することができます。

14.2.3. Hibernate Validator の制約

注記

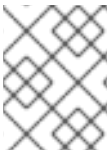
該当する場合は、アプリケーションレベルの制約により、以下の表の **Hibernate Metadata Impact** 列で説明されているデータベースレベルの制約が作成されます。

Java 固有のバリデーション制約

以下の表には、**`javax.validation.constraints`** パッケージに含まれる Java 仕様で定義されたバリデーション制約が示されています。

アノテーション	プロパティータイプ	ランタイムチェック	Hibernate Metadata の影響
`@AssertFalse`	ブール値	メソッドが false と評価することを確認します (アノテーションでなくコードで制約が表現されている場合に便利です)。	なし
`@AssertTrue`	ブール値	メソッドが true と評価することを確認します (アノテーションでなくコードで制約が表現されている場合に便利です)。	なし
`@Digits(integerDigits=1)`	数値または数値の文字列表現	プロパティーが **`integerDigits`** までの整数部と、**`fractionalDigits`** までの小数部を持つ数字であるかを確認します。	カラムの精度とスケールを定義します。
`@Future`	日付またはカレンダー	未来の日付であるかを確認します。	なし
`@Max(value=)`	数値または数値の文字列表現	値が最大値以下であるかを確認します。	カラムに check 制約を追加します。
`@Min(value=)`	数値または数値の文字列表現	値が最小値以上であるかを確認します。	カラムに check 制約を追加します。
`@NotNull`		値が null でないかを確認します。	カラムが null でないかを確認します。
`@Past`	日付またはカレンダー	過去の日付であるかを確認します。	カラムに check 制約を追加します。

アノテーション	プロパティータイプ	ランタイムチェック	Hibernate Metadata の影響
@Pattern(regexp="regexp", flag=) または **@Patterns({@Pattern(…)})**	文字列	プロパティーが一致フラグが指定された正規表現に一致するかどうかを確認します。**java.util.regex.Pattern** を参照してください。	なし
@Size(min=, max=)	アレイ、コレクション、マップ	要素サイズが最小値以上で最大値以下であるかどうかを確認します。	なし
@Valid	オブジェクト	紐付けされたオブジェクトに再帰的にバリデーションを実行します。オブジェクトがコレクションかアレイの場合は、要素は再帰的に検証されます。また、オブジェクトがマップの場合、値要素が再帰的に検証されます。	なし



注記

パラメーター **@Valid** は、**javax.validation.constraints** パッケージに存在しますが Bean Validation 仕様の一部です。

Hibernate Validator 固有のバリデーション制約

以下の表には、**org.hibernate.validator.constraints** パッケージに含まれるベンダー固有のバリデーション制約が含まれます。

アノテーション	プロパティータイプ	ランタイムチェック	Hibernate Metadata の影響
@Length(min=, max=)	文字列	文字列の長さが指定の範囲と一致するかを確認します。	カラムの長さを最大に設定します。
@CreditCardNumber	文字列	文字列が正規の形式のクレジットカード番号であるかどうかを確認します (Luhn アルゴリズムの派生)。	なし

アノテーション	プロパティータイプ	ランタイムチェック	Hibernate Metadata の影響
@EAN	文字列	文字列が正しくフォーマットされた EAN あるいは UPC-A コードであるかを確認します。	なし
@Email	文字列	文字列がメールアドレスの仕様に準拠するかどうかを確認します。	なし
@NotEmpty		文字列が null あるいは空でないかを確認します。接続が null あるいは空でないかを確認します。	カラムは文字列の null ではありません。
@Range(min=, max=)	数値または数値の文字列表現	値が最小値以上で最大値以下であるかどうかを確認します。	カラムに check 制約を追加します。

14.3. バリデーション設定

Bean バリデーションは、**/META-INF** ディレクトリーにある **validation.xml** ファイル内の XML を使用して設定できます。このファイルがクラスパスに存在する場合は、**ValidatorFactory** が作成されたときに設定が適用されます。

バリデーション設定ファイルの例

以下の例は、**validation.xml** ファイルの複数の設定オプションを示しています。これらすべての設定はオプションです。これらのオプションは、**javax.validation** パッケージを使用して設定することもできます。

```
<validation-config
xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration
">

    <default-provider>
        org.hibernate.validator.HibernateValidator
    </default-provider>
    <message-interpolator>

org.hibernate.validator.messageinterpolation.ResourceBundleMessageInterpol
ator
    </message-interpolator>
    <constraint-validator-factory>
        org.hibernate.validator.engine.ConstraintValidatorFactoryImpl
    </constraint-validator-factory>
```

```
    <constraint-mapping>
        /constraints-example.xml
    </constraint-mapping>

    <property name="prop1">value1</property>
    <property name="prop2">value2</property>
</validation-config>
```

ノード **default-provider** では、Bean バリデーションプロバイダーを選択できます。これは、クラスパスに複数のプロバイダーがある場合に役に立ちます。**message-interpolator** プロパティーと **constraint-validator-factory** プロパティーは、**javax.validation** パッケージで定義されたインターフェース **MessageInterpolator** および **ConstraintValidatorFactory** の使用済み実装をカスタマイズするために使用されます。**constraint-mapping** 要素は、実際の制約設定が含まれる追加の XML ファイルをリストします。

第15章 WEBSOCKET アプリケーションの作成

WebSocket プロトコルは、Web クライアントとサーバー間の双方向通信を提供します。クライアントとサーバー間の通信はイベントベースであるため、ポーリングベースの通信よりも処理が高速になり、帯域幅が小さくなります。WebSocket は、JavaScript API を用いて Web アプリケーションで使用したり、Java Websocket API を用いてクライアント WebSocket エンドポイントで使用したりできます。

最初に接続はクライアントとサーバー間で HTTP 接続として確立されます。その後、クライアントは **`Upgrade`** ヘッダーを使用して WebSocket 接続を要求します。同じ TCP/IP 接続上ではすべて全二重通信になり、データのオーバヘッドが最小化されます。各メッセージには不必要な HTTP ヘッダーコンテンツが含まれていないため、Websocket 通信で必要な帯域幅は小さくなります。その結果、通信パスのレイテンシーが低くなるため、リアルタイムの応答が必要なアプリケーションに適しています。

JBoss EAP WebSocket 実装は、サーバーエンドポイントに対して完全な依存関係注入サポートを提供しますが、クライアントエンドポイントに対して CDI サービスを提供しません。

WebSocket アプリケーションには以下のコンポーネントと設定変更が必要です。

- Java クライアントまたは WebSocket が有効になっている HTML クライアント。HTML クライアントのブラウザーのサポートについては http://caniuse.com/websockets で確認できます。
- WebSocket サーバーエンドポイントクラス。
- WebSocket API で依存関係を宣言するために設定されたプロジェクト依存関係。

WebSocket アプリケーションの作成

以下のコード例は、JBoss EAP に同梱される **`websocket-hello`** クイックスタートの一部です。これは、接続を開き、メッセージを送信し、接続を閉じる WebSocket アプリケーションの単純な例です。他の機能を実装せず、実際のアプリケーションで必要となるエラー処理を含みません。

1. JavaScript HTML クライアントを作成します。
   以下は WebSocket クライアントの例になります。この例には 3 つの JavaScript 関数が含まれています。

   - **`connect()`**: この関数は WebSocket URI を渡す WebSocket 接続を作成します。リソースの場所は、サーバーエンドポイントクラスに定義されたリソースと一致します。この関数は、WebSocket の **`onopen`**、**`onmessage`**、**`onerror`**、および **`onclose`** もインターセプトし、処理します。
   - **`sendMessage()`**: この関数はフォームに入力された名前を取得し、メッセージを作成します。さらに、WebSocket.send() コマンドを使用してメッセージを送信します。
   - **`disconnect()`**: この関数は WebSocket.close() コマンドを実行します。
   - **`displayMessage()`**: この関数は、ページ上の表示メッセージを WebSocket エンドポイントメソッドによって返された値に設定します。
   - **`displayStatus()`**: この関数は WebSocket の接続状態を表示します。

   アプリケーションの index.html コード例

```
<html>
  <head>
    <title>WebSocket: Say Hello</title>
    <link rel="stylesheet" type="text/css"
href="resources/css/hello.css" />
```

```
    <script type="text/javascript">
      var websocket = null;
      function connect() {
        var wsURI = 'ws://' + window.location.host + '/jboss-
websocket-hello/websocket/helloName';
        websocket = new WebSocket(wsURI);
        websocket.onopen = function() {
            displayStatus('Open');
            document.getElementById('sayHello').disabled =
false;
            displayMessage('Connection is now open. Type a name
and click Say Hello to send a message.');
        };
        websocket.onmessage = function(event) {
            // log the event
            displayMessage('The response was received! ' +
event.data, 'success');
        };
        websocket.onerror = function(event) {
            // log the event
            displayMessage('Error! ' + event.data, 'error');
        };
        websocket.onclose = function() {
            displayStatus('Closed');
            displayMessage('The connection was closed or timed
out. Please click the Open Connection button to reconnect.');
            document.getElementById('sayHello').disabled = true;
        };
      }
      function disconnect() {
        if (websocket !== null) {
            websocket.close();
            websocket = null;
        }
        message.setAttribute("class", "message");
        message.value = 'WebSocket closed.';
        // log the event
      }
      function sendMessage() {
        if (websocket !== null) {
            var content = document.getElementById('name').value;
            websocket.send(content);
        } else {
            displayMessage('WebSocket connection is not
established. Please click the Open Connection button.', 'error');
        }
      }
      function displayMessage(data, style) {
        var message = document.getElementById('hellomessage');
        message.setAttribute("class", style);
        message.value = data;
      }
      function displayStatus(status) {
        var currentStatus =
document.getElementById('currentstatus');
        currentStatus.value = status;
```

```
      }
    </script>
  </head>
  <body>
    <div>
      <h1>Welcome to Red Hat JBoss Enterprise Application
Platform!</h1>
      <div>This is a simple example of a WebSocket
implementation.</div>
      <div id="connect-container">
        <div>
          <fieldset>
            <legend>Connect or disconnect using websocket
:</legend>
            <input type="button" id="connect"
onclick="connect();" value="Open Connection" />
            <input type="button" id="disconnect"
onclick="disconnect();" value="Close Connection" />
          </fieldset>
        </div>
        <div>
            <fieldset>
              <legend>Type your name below, then click the `Say
Hello` button :</legend>
              <input id="name" type="text" size="40"
style="width: 40%"/>
              <input type="button" id="sayHello"
onclick="sendMessage();" value="Say Hello" disabled="disabled"/>
            </fieldset>
        </div>
        <div>Current WebSocket Connection Status: <output
id="currentstatus" class="message">Closed</output></div>
        <div>
          <output id="hellomessage" />
        </div>
      </div>
    </div>
  </body>
</html>
```

2. WebSocket サーバーエンドポイントを作成します。
以下の方法のいずれかを使用して WebSocket サーバーエンドポイントを作成できます。

   - プログラム的なエンドポイント(Programmatic Endpoint): エンドポイントは Endpoint クラスを拡張します。
   - **アノテーション付きエンドポイント**): エンドポイントクラスはアノテーションを使用して WebSocket イベントと対話します。これは、プログラム的なエンドポイントよりも簡単にコーティングできます。

   以下のコード例では、アノテーション付きエンドポイントが使用され、以下のイベントが処理されます。

   - **@ServerEndpoint** アノテーションは、このクラスを WebSocket サーバーエンドポイントとして識別し、パスを指定します。

- WebSocket 接続が開かれると **@OnOpen** アノテーションがトリガーされます。
- メッセージが受信されると、**@OnMessage** アノテーションがトリガーされます。
- WebSocket 接続が閉じられると、**@OnClose** アノテーションがトリガーされます。

WebSocket エンドポイントのコード例

```
package org.jboss.as.quickstarts.websocket_hello;

import javax.websocket.CloseReason;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/websocket/helloName")
public class HelloName {

    @OnMessage
    public String sayHello(String name) {
        System.out.println("Say hello to '" + name + "'");
        return ("Hello" + name);
    }

    @OnOpen
    public void helloOnOpen(Session session) {
        System.out.println("WebSocket opened: " +
session.getId());
    }

    @OnClose
    public void helloOnClose(CloseReason reason) {
        System.out.println("WebSocket connection closed with
CloseCode: " + reason.getCloseCode());
    }
}
```

3. プロジェクト POM ファイルで WebSocket API の依存関係を宣言します。
   Maven を使用する場合は、プロジェクト **pom.xml** ファイルに以下の依存関係を追加します。

Maven 依存関係の例

```
<dependency>
  <groupId>org.jboss.spec.javax.websocket</groupId>
  <artifactId>jboss-websocket-api_1.0_spec</artifactId>
  <version>1.0.0.Final</version>
  <scope>provided</scope>
</dependency>
```

JBoss EAP に同梱されるクイックスタートには、追加の WebSocket クライアントとエンドポイントのコード例が含まれます。

第16章 JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS)

16.1. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS)

JACC (Java Authorization Contract for Containers) はコンテナーと承認サービスプロバイダー間のコントラクトを定義する規格であり、これによりコンテナーによって使用されるプロバイダーの実装が可能になります。JACC は、Java コミュニティープロセスの JSR-115 で定義されています。この仕様の詳細については、Java™ Authorization Contract for Containers を参照してください。

JBoss EAP は **security** サブシステムのセキュリティー機能内に JACC のサポートを実装します。

16.2. JACC (JAVA AUTHORIZATION CONTRACT FOR CONTAINERS) のセキュリティーの設定

JACC (Java Authorization Contract for Containers) を設定するには、適切なモジュールでセキュリティードメインを設定し、必須のパラメーターが含まれるよう **jboss-web.xml** を編集する必要があります。

セキュリティードメインへの JACC サポートの追加

セキュリティードメインに JACC サポートを追加するには、**required** フラグセットで **JACC** 承認ポリシーをセキュリティードメインの承認スタックへ追加します。以下は JACC サポートを持つセキュリティードメインの例です。ただし、セキュリティードメインは直接 XML を変更せずに、管理コンソールまたは管理 CLI で設定することが推奨されます。

```
<security-domain name="jacc" cache-type="default">
    <authentication>
        <login-module code="UsersRoles" flag="required">
        </login-module>
    </authentication>
    <authorization>
        <policy-module code="JACC" flag="required"/>
    </authorization>
</security-domain>
```

JACC を使用するよう Web アプリケーションを設定

jboss-web.xml は デプロイメントの **WEB-INF/** ディレクトリーに存在し、Web コンテナーに対する追加の JBoss 固有の設定を格納し、上書きします。JACC が有効になっているセキュリティードメインを使用するには、**<security-domain>** 要素が含まれるようにし、 さらに **<use-jboss-authorization>** 要素を **true** に設定する必要があります。以下の XML は、上記の JACC セキュリティードメインを使用するよう設定されています。

```
<jboss-web>
    <security-domain>jacc</security-domain>
    <use-jboss-authorization>true</use-jboss-authorization>
</jboss-web>
```

JACC を使用するよう EJB アプリケーションを設定

セキュリティードメインと JACC を使用するよう EJB を設定する方法は Web アプリケーションの場合とは異なります。EJB の場合、**ejb-jar.xml** 記述子にてメソッドまたはメソッドのグループ上でメ

ソッドパーミッションを宣言できます。**<ejb-jar>** 要素内では、すべての子 **<method-permission>** 要素に JACC ロールに関する情報が含まれます。詳細については、設定例を参照してください。**EJBMethodPermission** クラスは Java EE 7 API の一部であり、http://docs.oracle.com/javaee/7/api/javax/security/jacc/EJBMethodPermission.html で説明されています。

EJB の JACC メソッドパーミッション例

```
<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles may access any
method of the EmployeeService bean </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

Web アプリケーションと同様にセキュリティードメインを使用して EJB の認証および承認メカニズムを指定することも可能です。セキュリティードメインは **<security>** 子要素の **jboss-ejb3.xml** 記述子に宣言されます。セキュリティードメインの他に、EJB が実行されるプリンシパルを変更する **<run-as-principal>** を指定することもできます。

EJB におけるセキュリティードメイン宣言の例

```
<ejb-jar>
    <assembly-descriptor>
        <security>
        <ejb-name>*</ejb-name>
        <security-domain>myDomain</security-domain>
        <run-as-principal>myPrincipal</run-as-principal>
        </security>
    </assembly-descriptor>
</ejb-jar>
```

第17章 JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS)

17.1. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) のセキュリティー

Java Authentication SPI for Containers (JASPI または JASPIC) は Java アプリケーションのプラグ可能なインターフェースであり、Java Community Process の JSR-196 に定義されています。この仕様の詳細については、http://www.jcp.org/en/jsr/detail?id=196 を参照してください。

17.2. JASPI (JAVA AUTHENTICATION SPI FOR CONTAINERS) のセキュリティーの設定

JASPI プロバイダーを認証するには、**<authentication-jaspi>** 要素をセキュリティードメインに追加します。設定は標準的な認証モジュールと似ていますが、ログインモジュール要素は **<login-module-stack>** 要素で囲まれています。設定の構成は次のとおりです。

例: authentication-jaspi 要素の構成

```
<authentication-jaspi>
    <login-module-stack name="...">
      <login-module code="..." flag="...">
        <module-option name="..." value="..."/>
      </login-module>
    </login-module-stack>
    <auth-module code="..." login-module-stack-ref="...">
      <module-option name="..." value="..."/>
    </auth-module>
</authentication-jaspi>
```

ログインモジュール自体は標準的な認証モジュールと同じように設定されます。

Web ベースの管理コンソールは JASPI 認証モジュールの設定を公開しないため、JBoss EAP を完全に停止してから、設定を **/domain/configuration/domain.xml** または **/standalone/configuration/standalone.xml** へ直接追加する必要があります。

第18章 JAVA バッチアプリケーション開発

JBoss EAP 7 以降、JBoss EAP では JSR-352 で定義された Java バッチアプリケーションがサポートされます。JBoss EAP の**バッチ**サブシステムにより、バッチ設定と監視が行えるようになります。

JBoss EAP でバッチ処理を使用するようアプリケーションを設定するには、必要な依存関係を指定する必要があります。バッチ処理向けの追加の JBoss EAP 機能には、Job Specification Language (JSL) 継承とバッチプロパティーインジェクションが含まれます。

18.1. 必要なバッチ依存関係

JBoss EAP にバッチアプリケーションをデプロイするには、バッチ処理に必要な追加の依存関係をアプリケーションの **pom.xml** で宣言する必要があります。必要なこれらの依存関係の例を以下に示します。ほとんどの依存関係は JBoss EAP にすでに含まれているため、スコープは **provided** に設定されます。

pom.xml のバッチ依存関係の例

```
<dependencies>
    <dependency>
        <groupId>org.jboss.spec.javax.batch</groupId>
        <artifactId>jboss-batch-api_1.0_spec</artifactId>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>javax.enterprise</groupId>
        <artifactId>cdi-api</artifactId>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.jboss.spec.javax.annotation</groupId>
        <artifactId>jboss-annotations-api_1.2_spec</artifactId>
        <scope>provided</scope>
    </dependency>

    <!-- Include your application's other dependencies. -->
    ...
</dependencies>
```

18.2. JOB SPECIFICATION LANGUAGE (JSL) 継承

JBoss EAP **batch-jberet** サブシステムの機能を使用すると、Job Specification Language (JSL) 継承を使用してジョブ定義の共通の部分を抽象化できます。JSL 継承は JSR-352 1.0 仕様に含まれていないため、JBoss EAP **batch-jberet** サブシステムは JSL 継承 v1 ドラフトに基づいて JSL 継承を実装します。継承ルールと制限については、ドラフト版のドキュメントを参照してください。

例: 同じジョブ XML ファイル内の step および flow の継承
親要素 (step や flow など) は、直接的な実行から除外するために属性 **abstract="true"** でマークされます。子要素には、親要素を参照する **parent** 属性が含まれます。

```
<job id="inheritance" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
```

```
version="1.0">
    <!-- abstract step and flow -->
    <step id="step0" abstract="true">
        <batchlet ref="batchlet0"/>
    </step>

    <flow id="flow0" abstract="true">
        <step id="flow0.step1" parent="step0"/>
    </flow>

    <!-- concrete step and flow -->
    <step id="step1" parent="step0" next="flow1"/>

    <flow id="flow1" parent="flow0"/>
</job>
```

例: 別のジョブ XML ファイルからの step の継承
子要素 (step や job など) には以下の属性が含まれます。

- **jsl-name** 属性。親要素を含むジョブ XML ファイルの名前 (**.xml** 拡張子なし) を指定します。
- **parent** 属性。**jsl-name** で指定されたジョブ XML ファイルの親要素を参照します。

親要素は、直接的な実行から除外するために属性 **abstract="true"** でマークされます。

chunk-child.xml

```
<job id="chunk-child" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
version="1.0">
    <step id="chunk-child-step" parent="chunk-parent-step" jsl-
name="chunk-parent">
    </step>
</job>
```

chunk-parent.xml

```
<job id="chunk-parent" >
    <step id="chunk-parent-step" abstract="true">
        <chunk checkpoint-policy="item" skip-limit="5" retry-limit="5">
            <reader ref="R1"></reader>
            <processor ref="P1"></processor>
            <writer ref="W1"></writer>

            <checkpoint-algorithm ref="parent">
                <properties>
                    <property name="parent" value="parent"></property>
                </properties>
            </checkpoint-algorithm>
            <skippable-exception-classes>
                <include class="java.lang.Exception"></include>
                <exclude class="java.io.IOException"></exclude>
            </skippable-exception-classes>
            <retryable-exception-classes>
                <include class="java.lang.Exception"></include>
                <exclude class="java.io.IOException"></exclude>
```

```
            </retryable-exception-classes>
            <no-rollback-exception-classes>
                <include class="java.lang.Exception"></include>
                <exclude class="java.io.IOException"></exclude>
            </no-rollback-exception-classes>
        </chunk>
    </step>
</job>
```

18.3. バッチプロパティーインジェクション

JBoss EAP **batch-jberet** サブシステムの機能を使用すると、ジョブ XML ファイルで定義されたプロパティーをバッチアーティファクトクラスのフィールドにインジェクトできます。ジョブ XML ファイルで定義されたプロパティーは **@Inject** アノテーションと **@BatchProperty** アノテーションを使用してフィールドにインジェクトできます。

インジェクトフィールドは以下のいずれかの Java タイプになります。

- **java.lang.String**
- **java.lang.StringBuilder**
- **java.lang.StringBuffer**
- 以下のいずれかのプリミティブタイプおよびラッパータイプ:
  - **boolean**、**Boolean**
  - **int**、**Integer**
  - **double**、**Double**
  - **long**、**Long**
  - **char**、**Character**
  - **float**、**Float**
  - **short**、**Short**
  - **byte**、**Byte**
- **java.math.BigInteger**
- **java.math.BigDecimal**
- **java.net.URL**
- **java.net.URI**
- **java.io.File**
- **java.util.jar.JarFile**
- **java.util.Date**

- **java.lang.Class**
- **java.net.Inet4Address**
- **java.net.Inet6Address**
- **java.util.List、List<?>、List<String>**
- **java.util.Set、Set<?>、Set<String>**
- **java.util.Map、Map<?, ?>、Map<String, String>、Map<String, ?>**
- **java.util.logging.Logger**
- **java.util.regex.Pattern**
- **javax.management.ObjectName**

以下のアレイタイプもサポートされています。

- **java.lang.String[]**
- 以下のいずれかのプリミティブタイプおよびラッパータイプ:
  - **boolean[]、Boolean[]**
  - **int[]、Integer[]**
  - **double[]、Double[]**
  - **long[]、Long[]**
  - **char[]、Character[]**
  - **float[]、Float[]**
  - **short[]、Short[]**
  - **byte[]、Byte[]**
- **java.math.BigInteger[]**
- **java.math.BigDecimal[]**
- **java.net.URL[]**
- **java.net.URI[]**
- **java.io.File[]**
- **java.util.jar.JarFile[]**
- **java.util.zip.ZipFile[]**
- **java.util.Date[]**

- `java.lang.Class[]`

以下に、バッチプロパティーインジェクションの使用例をいくつか示します。

- 数字を Batchlet クラスにさまざまなタイプとしてインジェクトする
- 数字シーケンスを Batchlet クラスにさまざまなアレイとしてインジェクトする
- Batchlet クラスにクラスプロパティーをインジェクトする
- プロパティーインジェクション向けにアノテートされたフィールドにデフォルト値を割り当てる

例: 数字を Batchlet クラスにさまざまなタイプとしてインジェクトする

ジョブ XML ファイル

```
<batchlet ref="myBatchlet">
    <properties>
        <property name="number" value="10"/>
    </properties>
</batchlet>
```

アーティファクトクラス

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    int number;  // Field name is the same as batch property name.

    @Inject
    @BatchProperty (name = "number")  // Use the name attribute to locate
the batch property.
    long asLong;  // Inject it as a specific data type.

    @Inject
    @BatchProperty (name = "number")
    Double asDouble;

    @Inject
    @BatchProperty (name = "number")
    private String asString;

    @Inject
    @BatchProperty (name = "number")
    BigInteger asBigInteger;

    @Inject
    @BatchProperty (name = "number")
    BigDecimal asBigDecimal;
}
```

例: 数字シーケンスを Batchlet クラスにさまざまなアレイとしてインジェクトする

ジョブ XML ファイル

```
<batchlet ref="myBatchlet">
    <properties>
        <property name="weekDays" value="1,2,3,4,5,6,7"/>
    </properties>
</batchlet>
```

アーティファクトクラス

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    int[] weekDays;  // Array name is the same as batch property name.

    @Inject
    @BatchProperty (name = "weekDays")  // Use the name attribute to
locate the batch property.
    Integer[] asIntegers;  // Inject it as a specific array type.

    @Inject
    @BatchProperty (name = "weekDays")
    String[] asStrings;

    @Inject
    @BatchProperty (name = "weekDays")
    byte[] asBytes;

    @Inject
    @BatchProperty (name = "weekDays")
    BigInteger[] asBigIntegers;

    @Inject
    @BatchProperty (name = "weekDays")
    BigDecimal[] asBigDecimals;

    @Inject
    @BatchProperty (name = "weekDays")
    List asList;

    @Inject
    @BatchProperty (name = "weekDays")
    List<String> asListString;

    @Inject
    @BatchProperty (name = "weekDays")
    Set asSet;

    @Inject
    @BatchProperty (name = "weekDays")
    Set<String> asSetString;
}
```

例: クラスプロパティーを Batchlet クラスにインジェクトする

ジョブ XML ファイル

```
<batchlet ref="myBatchlet">
    <properties>
        <property name="myClass" value="org.jberet.support.io.Person"/>
    </properties>
</batchlet>
```

アーティファクトクラス

```
@Named
public class MyBatchlet extends AbstractBatchlet {
    @Inject
    @BatchProperty
    private Class myClass;
}
```

例: プロパティーインジェクション向けにアノテートされたフィールドにデフォルト値を割り当てる

ターゲットバッチプロパティーがジョブ XML ファイルで定義されていない場合は、アーティファクト Java クラスのフィールドにデフォルト値を割り当てることができます。ターゲットプロパティーが有効な値に解決される場合は、その値がそのフィールドにインジェクトされます。解決されない場合は、値がインジェクトされず、デフォルトのフィールド値が使用されます。

アーティファクトクラス

```
/**
 Comment character. If commentChar batch property is not specified in job
XML file, use the default value '#'.
 */
@Inject
@BatchProperty
private char commentChar = '#';
```

付録A リファレンス資料

A.1. 提供された UNDERTOW ハンドラー

AccessControlListHandler
クラス名: **io.undertow.server.handlers.AccessControlListHandler**

名前: **access-control**

リモートピアの属性に基づいて要求を受領または拒否できるハンドラー。

表A.1 パラメーター

名前	説明
acl	ACL ルール。このパラメーターは必須です。
attribute	Exchange 属性文字列。このパラメーターは必須です。
default-allow	ハンドラーがデフォルトで要求を受領または拒否するかどうかを指定するブール値。デフォルト値は `false` です。

AccessLogHandler
クラス名: **io.undertow.server.handlers.accesslog.AccessLogHandler**

名前: **access-log**

アクセスログハンドラー。このハンドラーは、提供された書式文字列に基づいてアクセスログメッセージを生成し、提供された AccessLogReceiver にそれらのメッセージを渡します。

このハンドラーは、`ExchangeAttribute` メカニズムにより提供されたすべての属性をログに記録できます。

このファクトリーは、以下のパターンのトークンハンドラーを生成します。

表A.2 パターン

パターン	説明
%a	リモート IP アドレス
%A	ローカル IP アドレス
%b	送信済みバイト数 (HTTP ヘッダーまたは `-` を除く (バイトが送信されなかった場合))
%B	送信済みバイト数 (HTTP ヘッダーを除く)
%h	リモートホスト名

パターン	説明
%H	要求プロトコル
%l	**identd** からのリモート論理ユーザー名 (常に **-** を返します)
%m	要求メソッド
%p	ローカルポート
%q	クエリー文字列 (**?** 文字を除く)
%r	要求の最初の行
%s	応答の HTTP ステータスコード
%t	Common Log Format 形式の日時
%u	認証されたリモートユーザー
%U	要求された URL パス
%v	ローカルサーバー名
%D	要求を処理するのにかかった時間 (ミリ秒単位)
%T	要求を処理するのにかかった時間 (秒単位)
%I	現在の要求スレッド名 (後でスタックトレースと比較できます)
common	**%h %l %u %t "%r" %s %b**
combined	**%h %l %u %t "%r" %s %b "%{i,Referer}" "%{i,User-Agent}"**

クッキー、受信ヘッダー、またはセッションから情報を書き込むこともできます。

Apache 構文に基づきます。

- **%{i,xxx}** (受信ヘッダーの場合)
- **%{o,xxx}** (送信応答ヘッダーの場合)
- **%{c,xxx}** (特定のクッキーの場合)
- **%{r,xxx}** (ここで、**xxx** は ServletRequest の属性です)

- **%{s,xxx}** (ここで、**xxx** は HttpSession の属性です)

表A.3 パラメーター

名前	説明
format	ログメッセージを生成するために使用する形式。これはデフォルトパラメーターです。

AllowedMethodsHandler
特定の HTTP メソッドのホワイトリストに登録するハンドラー。許可されたメソッドセットのメソッドを持つ要求のみが許可されます。

クラス名: **io.undertow.server.handlers.AllowedMethodsHandler**

名前: **allowed-methods**

表A.4 パラメーター

名前	説明
methods	許可されるメソッド (**GET**、**POST**、**PUT** など)。これはデフォルトパラメーターです。

BlockingHandler
ブロック要求を開始する HttpHandler。スレッドが現在 I/O スレッドで実行されている場合、スレッドはディスパッチされます。

クラス名: **io.undertow.server.handlers.BlockingHandler**

名前: **blocking**

このハンドラーにはパラメーターがありません。

ByteRangeHandler
範囲要求のハンドラー。これは、修正されたコンテンツの長さのリソース (たとえば、**content-length** ヘッダーが設定されたリソース) に対する範囲要求を処理できる汎用ハンドラーです。コンテンツすべてが生成され、破棄されるため、これは必ずしも、範囲要求を処理する最も効率的な方法ではありません。現時点では、このハンドラーは単純な単一範囲要求しか処理できません。複数の範囲が要求された場合は、**Range** ヘッダーが無視されます。

クラス名: **io.undertow.server.handlers.ByteRangeHandler**

名前: **byte-range**

表A.5 パラメーター

名前	説明
send-accept-ranges	承認範囲を送信するかどうかを決定するブール値。これはデフォルトパラメーターです。

CanonicalPathHandler
このハンドラーは、相対パスを正規のパスに変換します。

クラス名: **io.undertow.server.handlers.CanonicalPathHandler**

名前: **canonical-path**

このハンドラーにはパラメーターがありません。

DisableCacheHandler
ブラウザーおよびプロキシーによる応答キャッシュを無効にするハンドラー。

クラス名: **io.undertow.server.handlers.DisableCacheHandler**

名前: **disable-cache**

このハンドラーにはパラメーターがありません。

DisallowedMethodsHandler
特定の HTTP メソッドをブラックリストに登録するハンドラー。

クラス名: **io.undertow.server.handlers.DisallowedMethodsHandler**

名前: **disallowed-methods**

表A.6 パラメーター

名前	説明
methods	許可しないメソッド (たとえば、**GET**、**POST**、**PUT** など)。これはデフォルトパラメーターです。

EncodingHandler
このハンドラーは、コンテンツのエンコーディング実装の基礎となります。このハンドラーに委譲するものとして、エンコーディングハンドラーが、指定されたサーバー側の優先度で追加されます。

正しいハンドラーを決定するために **q** 値が使用されます。**q** 値なしで要求が行われた場合、サーバーは使用するエンコーディングとして最も優先度が高いハンドラーを選択します。

一致するハンドラーがない場合は、ID エンコーディングが選択されます。**q** 値が **0** であるため、ID エンコーディングが特別に許可されない場合は、ハンドラーにより応答コード **406 (Not Acceptable)** が設定され、返されます。

クラス名: **io.undertow.server.handlers.encoding.EncodingHandler**

名前: **compress**

このハンドラーにはパラメーターがありません。

FileErrorPageHandler
エラーページとして使用するファイルをディスクから提供するハンドラー。このハンドラーはデフォルトで応答コードを提供しません。応答コードは設定する必要があります。

クラス名: **io.undertow.server.handlers.error.FileErrorPageHandler**

名前: **error-file**

表A.7 パラメーター

名前	説明
file	エラーページとして使用するファイルの場所。
response-codes	定義されたエラーページファイルにリダイレクトする応答コードのリスト。

HttpTraceHandler
HTTP トレース要求を処理するハンドラー。

クラス名: **io.undertow.server.handlers.HttpTraceHandler**

名前: **trace**

このハンドラーにはパラメーターがありません。

IPAddressAccessControlHandler
リモートピアの IP アドレスに基づいて要求を受領または拒否できるハンドラー。

クラス名: **io.undertow.server.handlers.IPAddressAccessControlHandler**

名前: **ip-access-control**

表A.8 パラメーター

名前	説明
acl	アクセス制御リストを表す文字列。これはデフォルトパラメーターです。
failure-status	拒否された要求で返されるステータスコードを表す文字列。
default-allow	デフォルトで許可するかしないかを表すブール値。

JDBCLogHandler
クラス名: **io.undertow.server.handlers.JDBCLogHandler**

名前: **jdbc-access-log**

表A.9 パラメーター

名前	説明
format	JDBC ログパターンを指定します。デフォルト値は **`common`** です。また、**`combined`** を使用して、VirtualHost、要求メソッド、参照元、およびユーザーエージェント情報をログメッセージに追加することもできます。

名前	説明
datasource	ログするデータソースの名前。このパラメーターは必須であり、デフォルトパラメーターです。
tableName	テーブル名。
remoteHostField	リモートホストアドレス。
userField	ユーザー名。
timestampField	タイムスタンプ。
virtualHostField	VirtualHost。
methodField	メソッド。
queryField	クエリー。
statusField	ステータス。
bytesField	バイト数。
refererField	参照元。
userAgentField	UserAgent。

LearningPushHandler
ブラウザーが要求するリソースのキャッシュを構築し、サーバープッシュを使用してリソースをプッシュ (サポートされている場合) するハンドラー。

クラス名: **io.undertow.server.handlers.LearningPushHandler**

名前: **learning-push**

表A.10 パラメーター

名前	説明
max-age	キャッシュエントリーの最大期間を表す整数。
max-entries	キャッシュエントリーの最大数を表す整数。

LocalNameResolvingHandler
DNS ルックアップを実行してローカルアドレスを解決するハンドラー。フロントエンドサーバーが **`X-forwarded-host`** ヘッダーを送信した場合、または AJP が使用中の場合は、未解決のローカルアドレスが作成されることがあります。

クラス名: **io.undertow.server.handlers.LocalNameResolvingHandler**

名前: **resolve-local-name**

このハンドラーにはパラメーターがありません。

PathSeparatorHandler
URL のスラッシュでない区切り文字をスラッシュに変換するハンドラー。一般的に、Windows システムではバックスラッシュはスラッシュに変換されます。

クラス名: **io.undertow.server.handlers.PathSeparatorHandler**

名前: **path-separator**

このハンドラーにはパラメーターがありません。

PeerNameResolvingHandler
リバース DNS ルックアップを実行してピアアドレスを解決するハンドラー。

クラス名: **io.undertow.server.handlers.PeerNameResolvingHandler**

名前: **resolve-peer-name**

このハンドラーにはパラメーターがありません。

ProxyPeerAddressHandler
`X-Forwarded-For` ヘッダーの値にピアアドレスを設定するハンドラー。これは、このヘッダーを常に設定するプロキシーの背後でのみ使用してください。そのように使用しないと、攻撃者がピアアドレスを偽造することがあります。

クラス名: **io.undertow.server.handlers.ProxyPeerAddressHandler**

名前: **proxy-peer-address**

このハンドラーにはパラメーターがありません。

RedirectHandler
`302` リダイレクトを使用して、指定された場所にリダイレクトするリダイレクトハンドラー。場所は exchange 属性文字列として指定されます。

クラス名: **io.undertow.server.handlers.RedirectHandler**

名前: **redirect**

表A.11 パラメーター

名前	説明
value	リダイレクトの宛先。これはデフォルトパラメーターです。

RequestBufferingHandler
すべての要求データをバッファするハンドラー。

クラス名: **io.undertow.server.handlers.RequestBufferingHandler**

名前: **buffer-request**

表A.12 パラメーター

名前	説明
buffers	バッファーの最大数を定義する整数。これはデフォルトパラメーターです。

RequestDumpingHandler
exchange をログにダンプするハンドラー。

クラス名: **io.undertow.server.handlers.RequestDumpingHandler**

名前: **dump-request**

このハンドラーにはパラメーターがありません。

RequestLimitingHandler
同時リクエストの最大数を制限するハンドラー。この制限を超えたリクエストは、前のリクエストが完了するまでブロックされます。

クラス名: **io.undertow.server.handlers.RequestLimitingHandler**

名前: **request-limit**

表A.13 パラメーター

名前	説明
requests	同時リクエストの最大数を表す整数。これはデフォルトパラメーターであり、必須です。

ResourceHandler
リソースを提供するハンドラー。

クラス名: **io.undertow.server.handlers.resource.ResourceHandler**

名前: **resource**

表A.14 パラメーター

名前	説明
location	リソースの場所。これはデフォルトパラメーターであり、必須です。
allow-listing	ディレクトリーのリストを許可するかどうかを決定するブール値。

ResponseRateLimitingHandler
設定された数のバイト/時間にダウンロードレートを制限するハンドラー。

クラス名: **io.undertow.server.handlers.ResponseRateLimitingHandler**

名前: **response-rate-limit**

表A.15 パラメーター

名前	説明
bytes	ダウンロードレートを制限するバイトの数。このパラメーターは必須です。
time	ダウンロードレートを制限する時間 (秒単位)。このパラメーターは必須です。

SetHeaderHandler
修正された応答ヘッダーを設定するハンドラー。

クラス名: **io.undertow.server.handlers.SetHeaderHandler**

名前: **header**

表A.16 パラメーター

名前	説明
header	ヘッダー属性の名前。このパラメーターは必須です。
value	ヘッダー属性の値。このパラメーターは必須です。

SSLHeaderHandler
以下のヘッダーに基づいて接続の SSL 情報を設定するハンドラー。

- SSL_CLIENT_CERT
- SSL_CIPHER
- SSL_SESSION_ID

このハンドラーがチェーンに存在する場合は、これらのヘッダーが存在しなくても SSL セッション情報が常に上書きされます。

このハンドラーは、リバースプロキシーの背後にあるサーバーでのみ**使用する必要があります** (**リクエストごと**にこれらのヘッダーを常に設定するよう (または、SSL 情報が存在しない場合にこれらの名前を持つ既存のヘッダーを削除するよう) リバースプロキシーが設定された場合)。このように使用しないと、悪意のあるクライアントが SSL 接続をスプーフすることがあります。

クラス名: **io.undertow.server.handlers.SSLHeaderHandler**

名前: **ssl-headers**

このハンドラーにはパラメーターがありません。

StuckThreadDetectionHandler
このハンドラーは処理に時間がかかるリクエストを検出します (処理中のスレッドが停止していることを示すことがあります)。

クラス名: **io.undertow.server.handlers.StuckThreadDetectionHandler**

名前: **stuck-thread-detector**

表A.17 パラメーター

名前	説明
threshhold	リクエストを処理する時間のしきい値を決定する整数値 (秒単位)。デフォルト値は **600** (10 分) です。これはデフォルトパラメーターです。

URLDecodingHandler
指定された文字セットに URL およびクエリーパラメーターをデコードするハンドラー。このハンドラーを使用している場合は、UndertowOptions.DECODE_URL パラメーターを **false** に設定する必要があります。

これはパーサーの組み込み UTF-8 デコーダーを使用する場合よりも効率的ではありません。UTF-8 以外の文字セットにデコードする必要がない限り、パーサーのデコードを使用してください。

クラス名: **io.undertow.server.handlers.URLDecodingHandler**

名前: **url-decoding**

表A.18 パラメーター

名前	説明
charset	デコードする文字セット。これはデフォルトパラメーターであり、必須です。

Revised on 2018-04-05 09:32:00 EDT