カスタムデータソースの使い方

この機能は、Release10.0 以降で利用できます。

このドキュメントは、Java クラスに基づく NXJ アプリケーションのカスタムデータソースを作成する方法について説明します。このドキュメントには、以下のセクションがあります。

- カスタムデータソースとは?
- カスタムデータソースを作成する方法
- カスタムデータソースのマスタ / 詳細関係
- 例

この機能をサポートするクラスの説明については、NXJ Javadoc を参照してください。

カスタムデータソースとは?

NXJ は JDBC データベースのテーブルへの自動接続を提供しますが、利用できる JDBC ドライバを持っていない（レガシーデータベースシステム、単層ファイルなど）多くのデータソースがあります。これらのソースは、データにアクセスするために Java メソッドを使用することができます。NXJ のカスタムデータソースは、外部の非 JDBC データソースであるインタフェースの 1 組の Java クラスであり、それが 1 組のテーブルとして NXJ で使用されることを可能にします。

NXJ のカスタムデータソースでは、各テーブルは Java クラスによって定義されます。Java クラスのプロパティは、テーブルの中の列を表します。XML ファイルは、テーブルと列にクラスとこれらのプロパティをマップします。また、実装クラスは外部データと NXJ 間で実際に接続するように注意します。

カスタムデータソースを作成する方法

カスタムデータソースを作成するには、以下のタスクを実行します。

タスク 1: カスタムデータソースの定義

NXJ のカスタムデータソースを作成する最初のステップは、データソースのテーブルを定義するために Java クラスを作成します。場合によっては、外部データへのインタフェースライブラリは、データソースのテーブルを定義するためにすぐに使用することができるクラスを提供します。

これらのクラスは、JavaBeans の設計パターンを踏襲する必要があります。クラス内の一般のフィールドや、一般の accessor/mutator メソッド（つまり、get*XXX*()/set*XXX*() メソッド）として定義されたプロパティを提供しています。これらの各プロパティは、クラスによって定義されるテーブルの列を意味します。例えば、顧客を定義するクラスは以下のようになります。

```
public class Customer {
    public int id;
    public String name;
    ...
```

あるいは、

```
public class Customer {
    ...
    public int getId() { ... }
    public void setId(int id) { ... }
    public String getName() { ... }
    public void setName() { ... }
    ...
```

これら両方の場合、同じプロパティ “id” と “name” は、データソースでのテーブルの列として定義されます。

タスク 2: クラスパスの更新

テーブルを定義するために使用するクラスのすべてを作成または見つけた場合は、NXJ プロジェクトのクラスパスにこれらを追加します。クラスパスを更新するには、**プロジェクト** > **プロパティ** > **クラスパス** を選択します。

タスク 3: カスタムデータソース ウィザードの実行

アプリケーションデザイナ メニューから、**ファイル** > **データソースの作成** を選択して、カスタムデータソース ウィザードを実行する準備を行います。

カスタムデータソース ウィザードのデータソースの作成ダイアログで、名称フィールドに作成するデータソースの名前を入力します。これは、基本的なデータがどこから来るのかを識別するためのものです。

プロジェクトのクラスパスにデータソースを作成するために必要となるすべてのクラスを含む jar ファイルを追加するために、**次へ** ボタンをクリックしてクラスパスダイアログに移動します。

クラスダイアログに移動するために、**次へ** ボタンをクリックします。クラスダイアログでは、クラスパスダイアログやマスタプロジェクトからのクラスパスに追加した jar ファイル中にある、すべてのクラスを含んでいるプロジェクトのクラスパスの全クラスの一覧を得ます。ここで、クラスツリーの中の 1 つ以上のクラスを選択するために “Ctrl+ クリック ” を使用して、データソースのテーブルを表すクラスを選択します。

マスタ / 詳細関係ダイアログに移動するために、**次へ** ボタンをクリックします。データタイプダイアログに移動するため、再び**次へ** ボタンをクリックしてください（マスタ / 詳細関係ダイアログについては、このドキュメントの後でより詳細に説明します）。

データタイプダイアログでは、クラスダイアログで選択したデータソーステーブルクラスの可能な全フィールドの一覧が表示されます。ここで、データソースに表示する列に NXJ データタイプとサイズを割り当てることができます。例えば、Java String フィールドは、NXJ の String あるいは Text 列のいずれかとして使用することができます。使用するタイプとして String を選択する場合、文字列の最大の長さも指定する必要があります。

列のタイプを割り当てたら、実際にデータソースを作成するために **完了** ボタンをクリックします。この時点で、3 つのオブジェクトは、NXJ プロジェクトに追加されます。

- データソースの列に、Java クラスフィールドを割り当てる XML ファイルの data sourceName.xml
- データソースに必要なコードのスケルトンバージョンを含む data sourceNameImpl.java と名づけられた Java クラス
- connection data sourceName

タスク 4: 新規コネクションを使用するためのフォームを開発

新規コネクションを使用するフォームを開発することができます。しかし、これらのフォームを配備して実行しようとして Find 操作を行っても、レコードを返しません。コードがまだ記述されていないため、追加、更新、削除操作は UnsupportedOperationException をスローします。

タスク 5: データソース実装クラスの記述

カスタムデータソースウィザードを終了すると、実際のバックエンドストレージメカニズムから、実際に retrieve/insert/modify/delete データへのデータソースによって使用される Java クラス data sourceNameImpl.java を持ちます。クラスは、**com.unify.NXJ.mgr.dataConnection.javaAdapter.NXJJavaClassData Source** のサブクラスで、コンストラクタ、close() と createTargetController() メソッドのためのオーバライドされたデフォルトメソッドを含みます。例えば、バックエンドストレージメカニズムから接続して切り離すために、コンストラクタや close() を変更することができます。通常、XML ファイル（これについては、このドキュメントでは説明していません。）を手で編集して、手動でテーブルをデータソースに追加しない限り、createTargetController を変更する必要はありません。

さらに重要なことには、実装クラスは、**com.unify.NXJ.mgr.dataConnection.javaAdapter.NXJJavaClassTargetController**（データソースの各テーブルの 1 つ）から、サブに分類される一般の内部クラスのシリーズを含みます。実装クラス自体の場合、これらのクラスはデフォルトメソッドを所定の位置にコード化させます。このコードは、追加、更新、削除操作のための UnsupportedOperationException をスローして、どんな検索操作に対しても空のセットを返します。実用的なデータソースを作るためにこれらのメソッドを具体化することが、カスタムデータソース開発者の責任となります。

検索メソッド

実装するべき最初のメソッドは findAllRecords() であり、オプションでは findSpecifiedRecords() です。カスタムデータソースウィザードが、findSpecifiedRecords() のために、デフォルトメソッドスクリプトを作成しないことに注意してください。**NXJJavaClassTargetController** クラスのデフォルトの実装は、通常そのまま使用できます。findSpecifiedRecords() メソッドを記述する方法の一例である **NXJJavaClassTargetController** について、Javadoc を参照してください。

findAllRecords() の記述は、一般に簡単です。テーブルデータを表すために使用されているクラスのオブジェクトの集まりに反復子（つまり、java.util.Iterator インタフェースを実装するクラス）を返す必要があります。通常、バックエンドデータリポジトリは、データを返すためのメソッドを提供します。findAllRecords() メソッドの実装は、単に返されたデータを使用して正しいクラスの新しいオブジェクトを作成し、ArrayList にこれらを追加して ArrayList に反復子を返します。

これらのメソッドは、反復子を返すために定義されるので、データソースは反復子セマンティクスを使用することができることに注意してください。これは、必要に応じてオンデマンドのデータローディングを行うことができることを意味します。

挿入メソッド

データソースが追加操作をサポートする場合、prepareInsertRow() と insert() の 2 つのメソッドを実装する必要があります。

挿入が実際に行われる前に prepareInsertRow() を呼び出して、レコードクラスの新しい空のオブジェクトを返す必要があります。このメソッドの実装は、簡単でなければなりません。

prepareInsertRow() が呼び出された後、NXJ は返されたオブジェクトを保持して、挿入する値でそのフィールドの値を置き換えます。prepareInsertRow() によって返されたオブジェクトは、パラメータとして insert() に渡されます。insert() のコードは、実際のバックエンドデータリポジトリを更新するために、このオブジェクトで値を使用することができます。insert() は、その後、選択セットの新しいレコードとして、NXJ によって使用されるレコードクラスのオブジェクトにリファレンスを返さなければなりません。通常、これは渡されたものと同じオブジェクトになります。

更新メソッド

2 つのメソッド、prepareUpdateRow() と update() は、更新操作を許可するデータソースのために実装する必要があります。

更新される前に、prepareUpdateRow() が呼び出されます。現在の選択セットのレコード（変更された値なし）が渡されて、レコードに対して新しい値で NXJ によって記入されるレコードクラスのインスタンスを返す必要があります。これは、

渡されたレコードのコピーか、渡されたレコードのいずれかです。後者の場合、そのレコードのインスタンスが変更されるので、レコードの識別子が変更される場合は、注意しなければなりません。

NXJがprepareUpdateRow()から返されたレコードを変更する場合、それはupdate()を呼び出し、オリジナルのレコードと変更されたレコードの両方を渡します。prepareUpdateRow()がレコード中で渡されて返された場合、両方のパラメータは同じオブジェクトを示します。update()は、新しいレコードのデータで、バックエンドリポジトリを更新するための実際の作業を行う必要があります。渡された古いレコードは、ユニークなレコード識別子が変更された場合に更新されなければならないレコードを識別するために使用されます。

削除メソッド

削除操作を実行するには、単一メソッドのdelete()の実装を必要とします。

delete()は、パラメータとして削除されるレコードのオブジェクトを取得します。メソッドの実装は、削除すべきバックエンドリポジトリのデータを識別するために渡されたレコードを使用し、それを削除するために適切な呼び出しを行う必要があります。

カスタムデータソースのマスタ / 詳細関係

リレーショナルデータベースでは、マスタ / 詳細関係は通常、マスタテーブルのプライマリキーを参照する詳細テーブルの外部キーによって定義されます。データソースがリレーショナルデータベースではない場合、マスタ / 詳細関係は、データを定義するJavaオブジェクトおよびクラスに実際に組み込まれる可能性があります。例えば、Invoiceオブジェクトは、そのフィールドの内の1つとしてInvoiceItemsのベクタを持っている可能性があり、ユーザがこのベクタを検索することを可能にするメソッドを提供します。InvoiceItemsがInvoiceクラスのベクタのみに存在するので、InvoiceItemsレコードクラスは、Invoiceクラスのプライマリキーに一致するためにフィールドを含む必要はありません。

NXJのカスタムデータソースは、この種のマスタ / 詳細関係でオブジェクトを使用することができます。カスタムデータソースウィザードのマスタ / 詳細関係パネルは、この種の関係を指定することを可能にします。マスタテーブル、詳細テーブル、詳細テーブルのレコードクラスのオブジェクトの集まりを返すマスタテーブ

ルのレコードクラスのメソッドを指定することができます。NXJ アプリケーションデザイナでマスタ / 詳細関係を作成するときに、マスタキーと詳細キーを指定せずにデータソースで定義するならば、メソッド名を指定することができます。

カスタムデータソースでマスタ / 詳細関係を使用する場合、追加、更新、削除操作をサポートするために実装される必要がある特別なメソッドがあります。それらは、prepareDetailInsertRow()、insertDetail()、prepareDetailUpdateRow()、updateDetail()、deleteDetail() です。デフォルトで、それらのメソッドは、それぞれ prepareInsertRow()、insert()、prepareUpdateRow()、update()、delete() を単に呼び出します。マスタレコードを変更する必要がある場合、マスタレコード、マスタテーブル名と関係メソッド名は、データソース開発者が最新版の正確な状況を決定し、マスタレコードを変更することを可能にする xxxDetailxxx() フォームメソッドに渡されます。

例

このセクションには、3 種類のカスタムデータソースの例があります。

- 単純なメモリデータソース
- 読み取り専用の Entity EJB データソース
- 完全な機能の Entity EJB データソース

単純なメモリデータソース

この例は、メモリに完全に格納される単純なデータソースを明示します。データソース実装クラスコンストラクタは、テーブルデータで一組のベクタを初期化します。様々なコントローラクラスのメソッドは、単純に新しいデータでベクタを変更します。

Customer レコードは、以下のクラスに格納されます。

```
package customds;

public class Customer {
    public int id;
    public String name;
    public String addr1;
    public String addr2;
    public String addr3;
```

```
public String city;
public String state;
public String postcode;
public String phone;
public String fax;

public Customer(
   int id,
   String name,
   String addr1,
   String addr2,
   String addr3,
   String city,
   String state,
   String postcode,
   String phone,
   String fax
) {
   this.id = id;
   this.name = name;
   this.addr1 = addr1;
   this.addr2 = addr2;
   this.addr3 = addr3;
   this.city = city;
   this.state = state;
   this.postcode = postcode;
   this.phone = phone;
   this.fax = fax;
}

public Customer() {
   this.id = 0;
   this.name = null;
   this.addr1 = null;
   this.addr2 = null;
   this.addr3 = null;
   this.city = null;
   this.state = null;
   this.postcode = null;
   this.phone = null;
   this.fax = null;
};

public Customer(Customer cust) {
   this.id = cust.id;
   this.name = cust.name;
```

```
        this.addr1 = cust.addr1;
        this.addr2 = cust.addr2;
        this.addr3 = cust.addr3;
        this.city = cust.city;
        this.state = cust.state;
        this.postcode = cust.postcode;
        this.phone = cust.phone;
        this.fax = cust.fax;
    }
};
```

データソースを実装する CustomDSImpl **クラスのためのコンストラクタでは、以下のようにベクタを初期化します。**

```
...
public CustomDSImpl() throws Exception {
    // The "tables" will simply be held in memory in
    // the respective Vectors

    // Initialize Customer vector
    customerVec = new Vector();

    customerVec.add(
        new Customer(1, "Unify Corporation",
            "2101 Arena Blvd Suite 100",
            null, null, "Sacramento", "CA", "95834",
            "(916)928-6400", "(916)928-6403"));
    customerVec.add(
        new Customer(2, "Corporation Limited",
            "4321 Main Avenue",
             null, null, "Anytown", "KY", "43210",
            "(237)555-6400", "(237)555-6403"));
    customerVec.add(
       new Customer(3, "PlanMen Incorporated",
          "666 Mockingbird Lane",
          null, null, "Sleepy Hollow", "PA", "12303",
          "(212)555-1234", "(212)555-1234"));
...
```

コントローラの findAllRecords() **メソッドは、オブジェクトを返すベクタ** iterator() **メソッドを使用するのと同じぐらい単純です。**

```
public class customds_Customer__Controller
    extends NXJJavaClassTargetController
{
    public Iterator findAllRecords() throws Exception {
```

```
        return customerVec.iterator();
    } // findAllRecords()
...
```

prepareInsertRow() メソッドは、Customer クラスの新しい空のインスタンスを返します。

```
...
    public Object prepareInsertRow()
        throws Exception
    {
        return new Customer();
    } // prepareInsertRow()
...
```

insert() メソッドは、Customer テーブルのベクタに新しいレコードを追加します。

```
...
    public Object insert(
        Object insertRow
        )
        throws Exception
    {
        customerVec.add(insertRow);
        return insertRow;
    } // insert()
...
```

prepareUpdateRow() は、行のカレントコピーのクローンを作成してそれを返し、Customer クラスに便利なように追加したコピーコンストラクタを使用します。

```
...
    public Object prepareUpdateRow(
        Object updateRow
        )
        throws Exception
    {
       return new Customer((Customer)updateRow);
    } // prepareUpdateRow()
...
```

update() は、ベクタの中の古い行を新しい行に置き替えます。

```
...
    public Object update(
        Object updateRow,
        Object oldRow
        )
        throws Exception
    {
        customerVec.set(customerVec.indexOf(oldRow), updateRow);
        return updateRow;
    } // update()
...
```

delete() **は、ベクタからレコードを削除します。**

```
...
    public void delete(
        Object rowid
        )
        throws Exception
    {
        customerVec.remove((Customer)rowid);
    } // delete()
...
```

Customer table controller **クラスで正しい位置にコード化します。これで利用可能な完全な検索、追加、更新、削除の機能性を持つことができます。**

Invoice **と** InvoiceItem **レコードタイプは、マスタ / 詳細関係がデータソースにおいてどのように使用することができるかを示します。**Invoice **クラスは、その請求書と前述のベクタを操作するメソッドに対応する** InvoiceItems **をすべて保持するベクタを含みます。**

```
package customds;

import java.sql.Date;
import java.util.Vector;
import java.util.Collection;

public class Invoice {
    public int num;
    public int cus_id;
    public Date entry_date;
    public Date ship_date;
    private Vector invoiceItems;
```

```
public Invoice(
  int num,
  int cus_id,
  Date entry_date,
  Date ship_date
  ) {
  this.num = num;
  this.cus_id = cus_id;
  this.entry_date = entry_date;
  this.ship_date = ship_date;
  this.invoiceItems = new Vector();
  }

  public Invoice() {
     this.num = 0;
     this.cus_id = 0;
     this.entry_date = null;
     this.ship_date = null;
     this.invoiceItems = new Vector();
  }

  public Invoice(Invoice inv) {
     this.num = inv.num;
     this.cus_id = inv.cus_id;
     this.entry_date = inv.entry_date;
     this.ship_date = inv.ship_date;
     this.invoiceItems = inv.invoiceItems;
  }

  public void addInvoiceItem(InvoiceItem invoiceItem) {
      invoiceItems.add(invoiceItem);
  }

  public void removeInvoiceItem(InvoiceItem invoiceItem) {
      invoiceItems.remove(invoiceItem);
  }

  public void updateInvoiceItem(InvoiceItem oldInvoiceItem,
      InvoiceItem newInvoiceItem) {
      invoiceItems.set(invoiceItems.indexOf(oldInvoiceItem),
          newInvoiceItem);
  }

  public Collection getInvoiceItems() {
```

```
        return invoiceItems;
    }
}
```

Invoice オブジェクトのベクタに常に請求書番号が格納されると予想されるので、InvoiceItem クラスは請求書番号をどこにも格納する必要がありません。

```
package customds;

import java.math.BigDecimal;

public class InvoiceItem {
    public int row_num;
    public String item_sku;
    public BigDecimal unit_price;
    public int quantity;
    public BigDecimal extended_price;

    public InvoiceItem(
  int row_num,
  String item_sku,
  BigDecimal unit_price,
  int quantity
    ) {
  this.row_num = row_num;
  this.item_sku = item_sku;
  this.unit_price = unit_price;
  this.quantity = quantity;
  this.extended_price =
    unit_price.multiply(BigDecimal.valueOf(quantity));
    }

    public InvoiceItem() {
  this.row_num = 0;
  this.item_sku = null;
  this.unit_price = null;
  this.quantity = 0;
  this.extended_price = null;
    }

    public InvoiceItem(InvoiceItem invItem) {
  this.row_num = invItem.row_num;
  this.item_sku = invItem.item_sku;
  this.unit_price = invItem.unit_price;
  this.quantity = invItem.quantity;
```

```
  this.extended_price = invItem.extended_price;
    }
}
```

データソースコンストラクタで、Invoice **と** InvoiceItem **テーブルを初期化します。**

```
...
    // Initialize Invoice vector
    // There is no Vector containing InvoiceItems - instead each
    // Invoice has its own Vector of InvoiceItems managed by the
    // addInvoiceItem()/removeInvoiceItem()/getInvoiceItems()
    // methods.
    invoiceVec = new Vector();
    Invoice inv;

    inv = new Invoice(1, 1, Date.valueOf("2003-07-22"),
            Date.valueOf("2003-07-23"));
    invoiceVec.add(inv);
    inv.addInvoiceItem(new InvoiceItem(1, "GAD10.5",
        BigDecimal.valueOf(345, 2), 10));
    inv.addInvoiceItem(new InvoiceItem(2, "GAD11.5",
        BigDecimal.valueOf(456, 2), 3));

    inv = new Invoice(2, 3, Date.valueOf("2003-06-20"),
        Date.valueOf("2003-07-01"));
    invoiceVec.add(inv);

    inv.addInvoiceItem(new InvoiceItem(1, "DOO0.5",
        BigDecimal.valueOf(40, 2), 5));
    inv.addInvoiceItem(new InvoiceItem(2, "DOO1.0",
        BigDecimal.valueOf(85, 2), 3));
    inv.addInvoiceItem(new InvoiceItem(3, "WID11.5",
        BigDecimal.valueOf(22345, 2), 2));
    inv.addInvoiceItem(new InvoiceItem(4, "WID10.5",
        BigDecimal.valueOf(12345, 2), 1));
...
```

Invoice **テーブルを操作する様々なメソッドは、**Customer **テーブル用に対応するものと同一です。しかし、**InvoiceItem **テーブル（詳細テーブル）のメソッドは変わります。**

最初に、findAllRecords() **メソッドは、**InvoiceItems **レコードの単一のリポジトリが無いことを考慮しなければなりません。代わりに、すべての** Invoice **レコードを繰り返し、リンクする** InvoiceItem **レコードのすべてを返します。**

```
public class customds_InvoiceItem__Controller
    extends NXJJavaClassTargetController
{
    public Iterator findAllRecords() throws Exception {
        List bag = new ArrayList();

        // Add to the bag an instance of the Java class
        // for each record.
        // Since there is no central list of InvoiceItems,
        // we need to iterate through all
        // Invoices and get the InvoiceItems for each of them.
        Iterator invoiceIterator = invoiceVec.iterator();
        while (invoiceIterator.hasNext()) {
            Iterator invoiceItemIterator =
                ((Invoice)invoiceIterator.next()).
                    getInvoiceItems().iterator();;
            while (invoiceItemIterator.hasNext())
                bag.add(invoiceItemIterator.next());
            }
        return bag.iterator();
    } // findAllRecords()
...
```

prepareInsertRow()、insert()、prepareUpdateRow()、update()、delete() **の代わりに、** prepareDetailInsertRow()、insertDetail()、prepareDetailUpdateRow()、updateDetail()、deleteDetail() **メソッドを実装する必要があります。**

```
...
    public Object prepareDetailInsertRow(
        String masterTableName,
        String relationshipName,
        Object masterInstance
        )
        throws Exception
    {
        return this.prepareInsertRow();
    } // prepareDetailInsertRow()

    public Object insertDetail(
        String masterTableName,
        String relationshipName,
        Object masterInstance,
        Object insertRow
        )
        throws Exception
```

```
    {
        ((Invoice)masterInstance).
            addInvoiceItem((InvoiceItem)insertRow);
        return insertRow;
    } // insertDetail()

    public Object prepareDetailUpdateRow(
        String masterTableName,
        String relationshipName,
        Object masterInstance,
        Object rowid
        )
        throws Exception
    {
        return this.prepareUpdateRow(rowid);
    } // prepareDetailUpdateRow()

    public Object updateDetail(
        String masterTableName,
        String relationshipName,
        Object masterInstance,
        Object updateRow,
        Object oldRow
        )
        throws Exception
    {
        ((Invoice)masterInstance).
            updateInvoiceItem((InvoiceItem)oldRow,
                              (InvoiceItem)updateRow);
        return updateRow;
    } // updateDetail()

    public void deleteDetail(
        String masterTableName,
        String relationshipName,
        Object masterInstance,
        Object rowid
        )
        throws Exception
    {
        ((Invoice)masterInstance).
            removeInvoiceItem((InvoiceItem)rowid);
    } // deleteDetail()
...
```

上記のルーチンでは、詳細データレコード格納を操作するために、渡された Invoice レコードをどのように使用するか注意してください。

読み取り専用の Entity EJB データソース

Entity EJB は、カスタムデータソースとしての使用に完全に適しているように見えます。Entity EJB は、JavaBean プロパティとしてデータを提供します。また、すべてのレコードを含んでいるコレクションを返すホームインタフェースには、便利なファインダメソッドがあります。読み取り専用データソースの場合、これは最適です。例えば、以下のリモートインタフェースで Entity EJB を保持します。

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Agent extends EJBObject
{
  Long getAgent_id() throws RemoteException;

  void setAgent_id(Long newAgent_id) throws RemoteException;

  String getName() throws RemoteException;

  void setName(String newName) throws RemoteException;

  String getAirline() throws RemoteException;

  void setAirline(String newAirline) throws RemoteException;

  String getAuto_rental() throws RemoteException;

  void setAuto_rental(String newAuto_rental) throws RemoteException;
}
```

以下は、ホームインタフェースです。

```
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.util.Collection;

public interface AgentHome extends EJBHome
```

```
{
  Agent create() throws RemoteException, CreateException;

  Agent findByPrimaryKey(Long primaryKey)
        throws RemoteException, FinderException;

  Collection findAll() throws RemoteException, FinderException;

  Agent create(Long agent_id)
        throws RemoteException, CreateException;
}
```

そのレコードクラスとして、リモートインタフェース (Agent) **を使用するカスタムデータソースを作成することができます。その後、**findAll() **メソッドは簡単になります。**

```
public class Agent__Controller
       extends NXJJavaClassTargetController
    {

       public Iterator findAllRecords() throws Exception {
           return agentHome.findAll().iterator();
       } // findAllRecords()
...
```

agentHome **は、データソースの実装クラスあるいはコントローラクラスのいずれかのためのコンストラクタで、**JNDI **ネームスペースで調べられる** Entity EJB **ホームインタフェースです。**

完全な機能の Entity EJB データソース

前の例では、Entity EJB **に基づく読み取り専用のデータソースを作成する方法を示しました。残念ながら、**2 **つの理由によって完全な追加 / 更新 / 削除機能を持つ** Entity EJB **データソースの作成は、更に難しいです。第一には、**NXJ **は新しい空のレコードを返すことを** prepareInsertRow() **と** prepareUpdateRow() **メソッドに要求します。しかし、**EJB **の仕様は、新しい** Entity EJB **が作成される前に、プライマリキーが提供されることを必要とします。第二に** update() **と** insert() **操作は、まだ原子的でなければなりません。**EJB **の仕様は、プロパティが変更されるとすぐに根本的なデータストアが更新されることを可能にします。これにより、**NXJ **でデータを格納するためのレコードクラスであるデータソース実装のためのヘルパークラス、およびヘルプ機能を含む** “shim” **クラスを定義する必要があります。**

上で示される同じ Agent bean **については、レコードクラスはこのように見えるでしょう。**

```
import javax.ejb.RemoveException;
import java.rmi.RemoteException;

public class AgentRecord
{
    private Long agent_id;
    private String name;
    private String airline;
    private String auto_rental;
    /* package */ Agent bean;

    public void setAgent_id(Long agent_id)
        { this.agent_id = agent_id; }
    public Long getAgent_id() { return this.agent_id; }
    public void setName(String name) { this.name = name; }
    public String getName() { return this.name; }
    public void setAirline(String airline)
        { this.airline = airline; }
    public String getAirline() { return this.airline; }
    public void setAuto_rental(String auto_rental)
        { this.auto_rental = auto_rental; }
    public String getAuto_rental() { return this.auto_rental; }

    public AgentRecord() {}// No-arg constructor
    public AgentRecord(Agent bean) throws RemoteException
    {
       this.bean = bean;
       this.refreshFromBean();
    }

    public void refreshFromBean() throws RemoteException
    {
       if (bean != null) {
           this.agent_id = bean.getAgent_id();
           this.name = bean.getName();
           this.airline = bean.getAirline();
           this.auto_rental = bean.getAuto_rental();
       }
    }

    public void updateBean() throws RemoteException
    {
```

```
       // Note that we do not update primary key!!!
      // We would have to create a new bean for this case!!!
       if (bean != null) {
           bean.setName(this.name);
           bean.setAirline(this.airline);
           bean.setAuto_rental(this.auto_rental);
       }
    }

    public void deleteBean() throws RemoveException
    {
       if (bean != null) {
           bean.remove();
           bean = null;
       }
    }

    public void assignBean(Agent bean)
    {
       this.bean = bean;
    }
}
```

AgentRecord クラスは、EJB 自体を更新するために使用することができる EJB への参照、および EJB から EJB にデータをコピーするためのメソッドと同様に、実際の EJB からデータを保持します。これは、テーブルを表わすために実際にデータソースによって使用されるクラスになります。また、より簡単に実際のデータソース実装を記述する “shim” クラスを作成します。

```
import javax.naming.*;
import java.util.ArrayList;
import java.util.Iterator;
import javax.ejb.FinderException;
import javax.ejb.DuplicateKeyException;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public class AgentShim
{

    private AgentHome home;

    public AgentShim() throws NamingException, RemoteException
    {
       InitialContext jndiContext = new InitialContext();
```

```
       home = (AgentHome)jndiContext.lookup("Agent");
    }

    public Iterator findAll() throws FinderException,
        RemoteException
    {
       ArrayList bag = new ArrayList();

       Iterator iter = home.findAll().iterator();
       while (iter.hasNext())
           bag.add(new AgentRecord((Agent)iter.next()));

       return bag.iterator();
    }

    public Agent createBean(Long agent_id) throws
        CreateException, RemoteException
    {
       return home.create(agent_id);
    }
}
```

これらのクラスが存在すれば、容易にコントローラクラスを作成することができます。

```
public class AgentRecord__Controller
    extends NXJJavaClassTargetController
{
    private AgentShim agentShim;

    public AgentRecord__Controller() throws Exception {
        agentShim = new AgentShim();
    }

    public Iterator findAllRecords() throws Exception {
        return agentShim.findAll();
    } // findAllRecords()

    public Object prepareInsertRow()
        throws Exception
    {
        return new AgentRecord();
    } // prepareInsertRow()

    public Object insert(
```

```
        Object insertRow
        )
        throws Exception
    {
        AgentRecord agentRecord = (AgentRecord)insertRow;
        agentRecord.assignBean(
            agentShim.createBean(agentRecord.getAgent_id()));
        agentRecord.updateBean();
        return agentRecord;
    } // insert()

    public Object prepareUpdateRow(
        Object rowid
        )
        throws Exception
    {
        return rowid;
    } // prepareUpdateRow()

    public Object update(
        Object updateRow,
        Object oldRow
        )
        throws Exception
    {
        ((AgentRecord)updateRow).updateBean();
        return updateRow;
    } // update()

    public void delete(
        Object rowid
        )
        throws Exception
    {
        ((AgentRecord)rowid).deleteBean();
    } // delete()

} // class AgentRecord__Controller
```

AgentShim **クラスが、絶対に必要というわけではない点に注意してください。** AgentShim **クラスの機能性はコントローラクラスに移行することができます。しかし、**AgentShim **クラスの使用は、より簡単なコントローラクラスを記述することができます。また、**AgentRecord **のいくつかの機能は、**AgentShim **クラスまたはコントローラクラスに移行することができます。**