JFlix™ Movie Rental Application

A Technical Overview

By

Smita Joshi

January 31, 2007

Abstract

JFlix™ is a Struts-based movie rental application powered by Software Tree's JDX Object-Relational Mapper (OR-Mapper) that simplifies data integration. This application is based on the Model View Controller (MVC) design pattern and uses Sakila, the MySQL sample database. It demonstrates how JDX reduces the amount and complexity of the data access code in the business logic layer by bridging the Object-Relational impedance mismatch seamlessly.

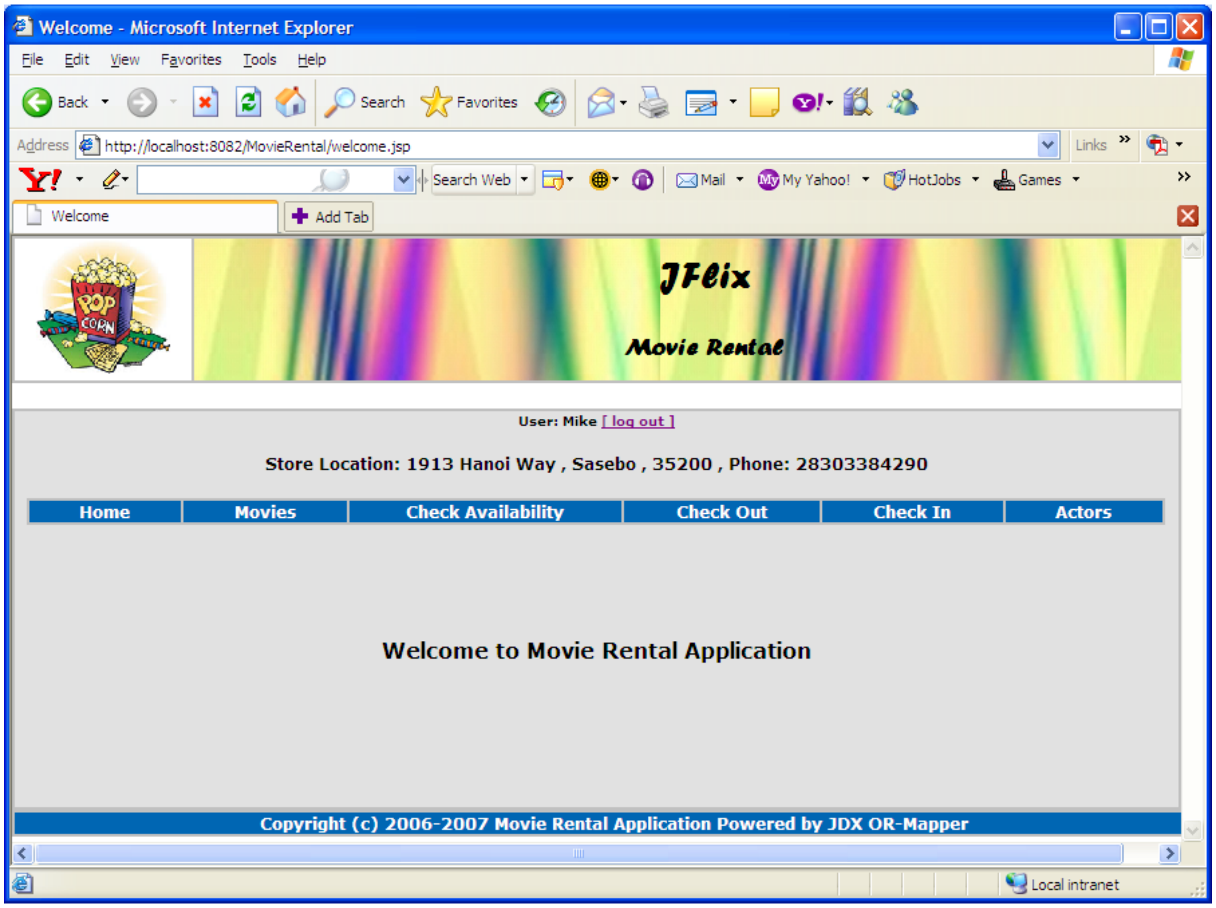
Table Of Contents

Introduction

This article describes the architecture, design, and implementation details of JFlix, a web-based movie rental application. The purpose of this article is to show how an efficient and versatile Object-Relational Mapping (OR-Mapping) product like JDX from Software Tree can simplify the complex and time-consuming task of integrating object-oriented POJO (Plain Old Java Object) data with relational data.

The JFlix application facilitates the typical activities of a staff member working at a movie rental store counter. Using this application, a staff member can check-in, checkout, and check the availability of requested movies. The application can also be used to search for a movie based on movie name or actor name.

This article will be useful for Java developers in general and for web application developers in particular. Some familiarity with Struts and OR-Mapping frameworks will be helpful, although not necessary.


Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8082/MovieRental/welcome.jsp
Links
Search Web Mail My Yahoo! HotJobs Games
Welcome
Add Tab
JFlix
Movie Rental
Home
Movies
Check Availability
Check Out
Check In
Actors
Local intranet

User: Mike [log out]

Store Location: 1913 Hanoi Way , Sasebo , 35200 , Phone: 28303384290

Welcome to Movie Rental Application

Copyright (c) 2006-2007 Movie Rental Application Powered by JDX OR-Mapper

Figure 1: JFlix Welcome Page

JFlix is a Struts-based web application, which implements MVC (Model-View-Controller) architecture. Java Server Pages (JSP) are used to render view; a start-up servlet, action forms, and action classes act as controller; and business objects control the application logic and communicate with a relational database using JDX OR-Mapper for data integration. The application uses the schema and data of

MySQL's sample database, *Sakila.* This article describes the use of the Tomcat web server but any other Java web server or application server can be used to run JFlix.

This application demonstrates the advantages of using JDX OR-Mapping middleware product for data integration. Adhering to some well thought-out KISS (Keep It Simple and Straight-forward) principles (see Reference [4]), JDX improves developer productivity significantly by presenting an intuitive, object-oriented view of the relational data. The business logic layer uses JDX APIs to communicate with the relational database to persist domain model objects (POJO). The same domain model objects are used for both the business logic tier and the presentation tier, eliminating the need for separate data transfer objects (DTO). Thus, the use of JDX not only simplifies the architecture, but also provides modularity and flexibility to the business logic layer. Similar code written in JDBC and standard SQL would be a lot more cumbersome, error-prone, and difficult to maintain.

Before we delve deep into the implementation of the application, let us first describe the database schema and the domain model used in this application followed by a brief overview of some basic JDX OR-Mapping concepts.

Database Schema

As mentioned before, we are using the existing schema and data provided by *Sakila*, the MySQL sample database,. Although the original schema has many tables, for the purposes of this application, we are using only the following subset of the tables:

- actor
- address
- category
- city
- country
- customer
- film
- film_actor
- film_category
- inventory
- rental
- staff
- store

Additionally, we have defined the view *STOREWITHADDR* to simplify some parts of the application:

```
CREATE VIEW STOREWITHADDR AS SELECT store_id, address, address2, district,
        city, postal_code, phone FROM store, city, address WHERE
        store.address_id = address.address_id AND address.city_id =
        city.city_id;
```

Code Fragment 1: SQL Statement to Create STOREWITHADDR View

Domain Model

The domain model below depicts all the business object (POJO) classes for the JFlix application. These classes belong to the package *movierental*.

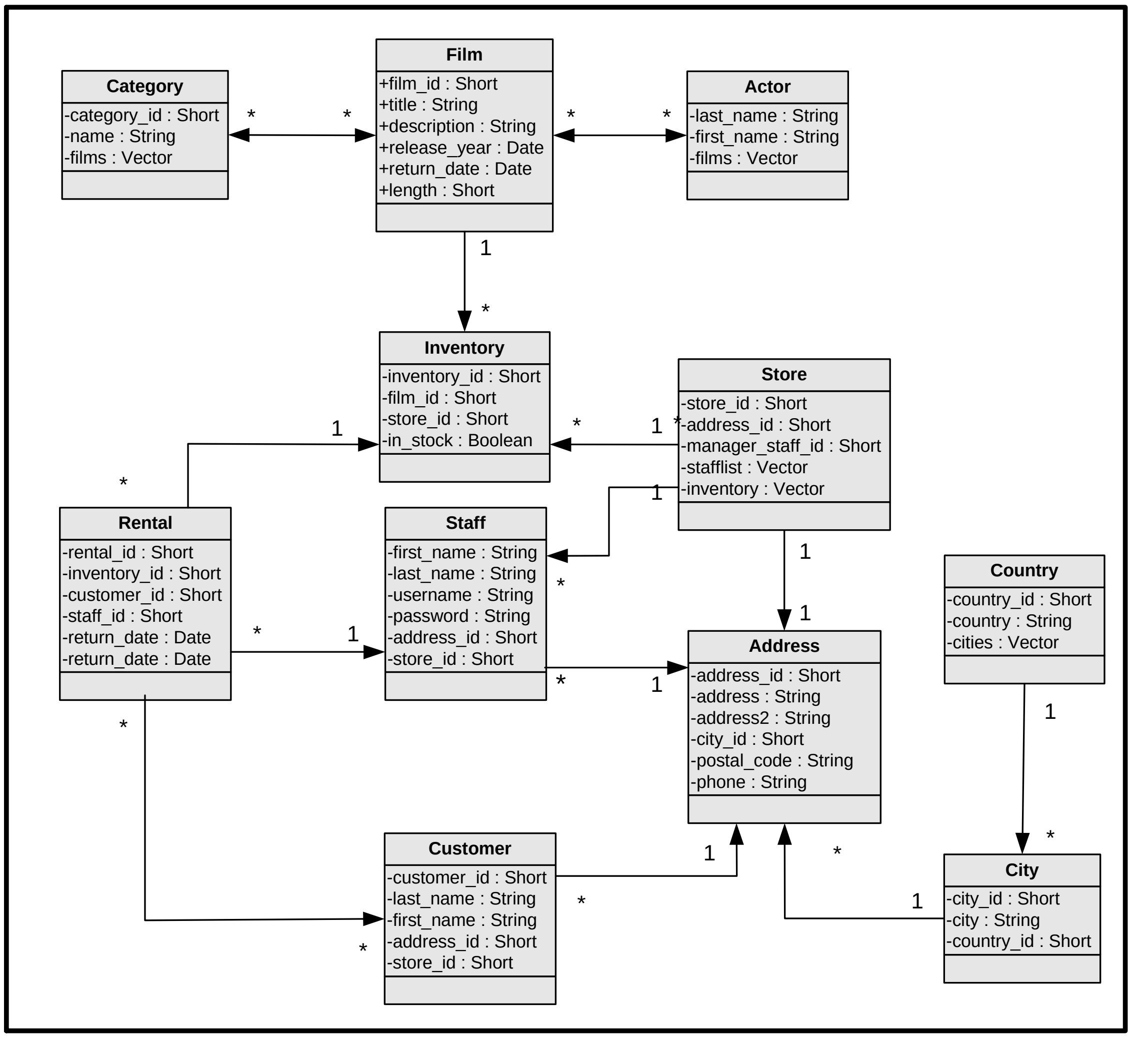


Figure 2: Domain Model

The above domain model shows all the attributes of the classes and the multiplicity between different classes. For example,

- Classes *Film* and *Actor* have a many-to-many relationship between them, i.e., a movie can have many actors in it and an actor can act in multiple movies.
- Classes *Inventory* and *Store* have a one-to-many relationship between them, i.e., a store can have multiple copies of a movie (inventory) but an inventory item can belong to only one store.
- Classes *Store* and *Address* have a one-to-one relationship between them. A store can have only one address and an address can correspond to only one store.

JDX OR-Mapper Concepts

Following are some important JDX concepts relevant to better understand the data integration layer of this application:

Object-Relational Mapping File (ORMFile)

An object-relational mapping (OR-Mapping) specification contains mapping information for all the persistent classes belonging to an application domain. The specification includes, among other things, table names, primary key attributes, and object relationships. JDX provides an innovative, declarative way of specifying human-readable and user-friendly object-relational mapping information based on a simple grammar. A text file (ORMFile) containing this mapping specification can be generated using a text editor, JDXStudio, a modeling tool, or even programmatically.

A mapping specification can correspond to only one database. While a mapping specification cannot span multiple databases, multiple mapping specifications can be defined for the same database.

Mapping Unit

A mapping specification (identified by an ORMFile) defines a mapping unit such that all interactions corresponding to the mapped classes and sequences in that specification share the same transaction manager and pool of database connections. Multiple threads of an application may share a mapping unit. An application can work with multiple mapping units.

JXResource

A JXResource provides the facilities to work with a mapping unit. A JXResource contains handles for JXSession and JDXS objects. JXSession provides transactional methods (like tx_begin and tx_commit) and JDXS provides, among other things, methods (like query, insert, update, and delete) to store and retrieve domain model objects.

JXResourcePool

A JXResourcePool provides a pool of JXResource(s) for a particular mapping unit based on an OR-Mapping specification. A JXResourcePool allows you to create multiple JXResource components in an extensible pool and provides you with methods for thread-safe sharing of these components. For each object-relational mapping file, there is a corresponding mapping unit and a JXResourcePool.

JFlix OR-Mapping File (*jflix.jdx*)

The JFlix OR-Mapping file, *jflix.jdx*, is reversed engineered from the existing *Sakila* database using the JDXStudio GUI tool. An excerpt of the mapping specification is shown in Appendix C. As you can see, the mapping specification is based on a simple grammar and is easy to create, modify, and comprehend. There are no XML complexities. A lot of the mapping information can be derived by JDX using intelligent defaults (for example, table names, column names, data types, and nullability). Some of the keywords used in the mapping file are explained below:

- **CLASS** specification encapsulates all the Object-Relational Mapping information for one class. The class name may include a namespace (e.g., *movierental.Film*).

- **PRIMARY_KEY** specification identifies the attribute (property) names whose combined values uniquely identify a particular object.

- **SQLMAP** specification allows one to refine the mapping of a class attribute to the SQL column in one of the following ways:
    - Using a column name different from the attribute name,
    - Using an SQL data type different from the default SQL data type for the attribute type, and
    - Allowing the column to be nullable (default is not nullable).

- **COLLECTION_CLASS** specification encapsulates all the Object-Relational Mapping information about a collection class. A collection is actually a pseudo-class; there may not be an actual class by that name in the program.

- **PRIMARY_KEY** specification for a collection class specifies names of those attributes whose values are the same for all the objects in a collection.

- **RELATIONSHIP** specification defines the mapping for a complex attribute referencing an object or a collection of objects. It keeps the application developer from worrying about explicitly initializing the "primary or foreign key" attribute values. BYVALUE keyword is used to indicate that, by default, the related objects should also be inserted, updated, or deleted with the containing object (CASCADE semantics).

JFlix Use Cases

The JFlix application facilitates the typical activities of a staff member working at a movie rental store counter. The following use cases are implemented in this application:

Use Case	Description	Assumptions/Limitations
Login/Logout	A staff member can login to the application using his/her username and password. The username and store location is saved for the session and is displayed on all pages until he/she logs out.	
Search Movie	A staff member can list all the movies, or search for a movie based on the name of the movie entered in full or in part. If multiple movies exist in the inventory matching the search criteria, all those movies are displayed. Clicking on the corresponding “Details” tab shows the details of a particular movie.	
Search Movie By Actor Name	Staff member can list all the actors in the database, or can search a movie based on the name of the actor entered in full or in part. If multiple actors exist in the database matching the search criteria, all those actors are displayed. Clicking on the corresponding “Movies” tab shows a list of movies for that particular actor.	
Check Availability	A staff member can check the availability of a movie by entering its name. The result shows the details for that movie and number of copies available in the inventory of that store.	
Check Out	A Staff member can check out movies based on the inventory ids of the movies and a customer id.	**Limitation**: The UI currently limits checkout of maximum of five movies at a time. **Assumption**: The customer physically brings the movies to the counter.
Check In	A Staff member can check in movies based on the inventory ids of the movies and a customer id.	**Limitation**: The UI currently limits check-in of maximum of five movies at a time. **Assumptions**: The customer returns the movies at the same store location from where he/she has rented those movies. Also, the customer physically brings the movies to the counter.

Table 1: JFlix Use Cases

JFlix Architecture and Implementation

This section describes the architecture and implementation of the JFlix application in detail. An application server (e.g. TOMCAT) is required to deploy the application war file and run the project. Once the war file is deployed you need to restart the application server in order to run the project. This is not needed if your application server uses hot deploy. Following are the different frameworks/software used in the project:

- JDX OR-Mapper – Version 4.7
- Struts – Version 1.2.9
- Struts-Layout - Version 2.0
- The Tomcat 4.1.x Servlet/JSP Container
- MySQL Server 5.0
- Sample Database Sakila – Version 0.8

The application is invoked by using the following URL in a web browser: http://<host>:<portnumber>/MovieRental. The host name and port number should be changed based on the configuration.

Application Control Flow

As mentioned before, JFlix uses the Struts framework based on the MVC (Model View Controller) pattern. Java Server Pages (JSP) are used to render view; the start-up servlets, action forms, and action classes act as controller; and the action classes communicate with the model component consisting of business objects and domain model objects to execute application logic. Business objects communicate with a relational database using JDX OR-Mapper, which provides data integration for domain model (POJO) objects with relational data. The figure below explains the control flow for the application:

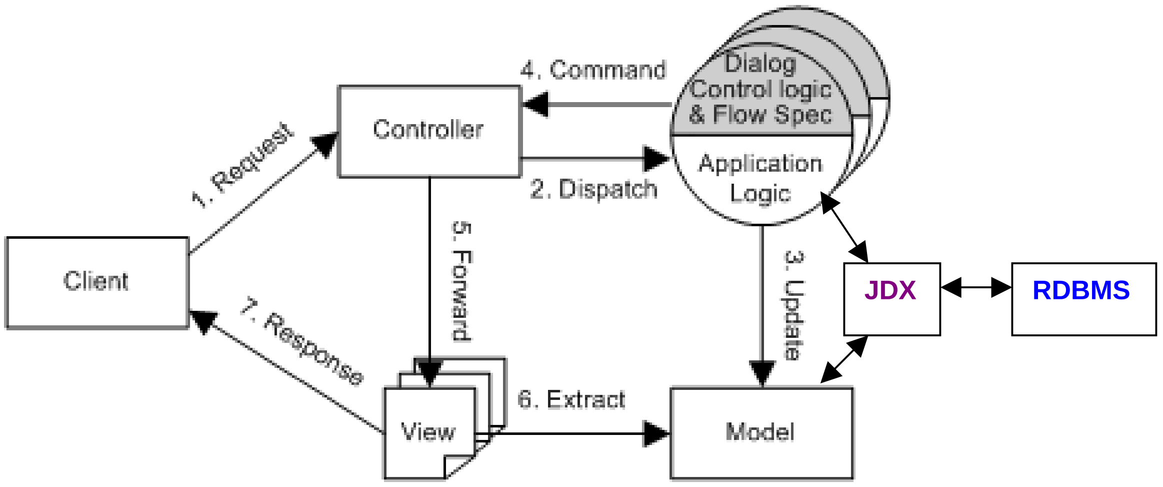


Figure 3: MVC Architecture (See Reference [1])

Now we will discuss the database tier, the presentation tier, and the business logic tier.

Database Tier

The implementation requires the MySQL sample database, *Sakila*, which can be downloaded from the following link: http://forge.mysql.com/wiki/SakilaSampleDB

The *Sakila* download contains two files:

- *sakila-schema.sql* : script to create schema (16 tables and 8 views)
- *sakila-data.sql* : script to populate data

The file *viewsforjdx.sql* is added for the purpose of this project to create an extra view *STOREWITHADDRESS* as described earlier.

If you already have the *Sakila* database installed on your machine, just add the extra view definition in the database as per the *viewsforjdx.sql* script.

Also, in the inventory table, add a Boolean field *"in_stock"* with a default value of *true*. Use the following command:

```
ALTER table inventory ADD in_stock BOOLEAN DEFAULT true;
```

Code Fragment 2: SQL statement to add in_stock field in inventory table

Presentation Tier

The presentation tier or the view component of application mainly comprises of JSP files.
The view components employed in JFlix application are:

- HTML
- Java server pages
- Custom tags
- Layout tags
- Java resource bundles

Struts provide a large number of JSP Custom Tags also called as Struts Tags. JFlix also uses Layout tags in combination with Struts tags, which extend the normal capabilities of JSP and simplify the development of view component.

Other than the Struts framework JSP files used in the application workflow (see Table [2]), some JSP files are used entirely for the purpose of display and navigation. These files are *header.jsp, footer.jsp,* and *menu.jsp*.

Start-up Servlets

Other than the standard struts action servlet, the web configuration file *web.xml* specifies the start-up servlet InitMovieRentalServlet to initialize the JXResourcePool. As soon as the web server is started, the servlet engine calls the *init()* method of the *InitMovieRentalServlet* class. This method initializes the JXResourcePool and registers it with a configured name (ORMPoolName), such that other modules can lookup and share this pool of JXResources using the configured name. The name of the pool and other parameters are provided through the <*init-param*> elements for this servlet in *web.xml* (see Appendix A).

Configuration File

The Struts application framework uses *struts-config.xml* (see Appendix B) as configuration file to initialize its own resources. These resources include:

- Action Forms: to collect user inputs
- Action Mappings: to direct input to server side Actions
- Action Forwards: to select output pages

Struts Framework Files and Work Flow

The following table summarizes different artifacts of the JFlix application interacting with each other to execute different use cases. In a following section, a particular use case (Movie Check Out) is described in more detail:

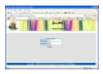
Use Case	JSP	Action Form	Action	Business Logic API (see Appendix D)	Screen Shots (click for details)
Login /Logout	login.jsp logout.jsp	UserForm	LoginAction	getObjects	
Search Movie	searchFilm.jsp filmList.jsp filmDetails.jsp noFilmsFound.jsp	FilmForm	FilmAction FilmDetailsAction	getObjects GetObjectById	
Check Availability	checkAvailability.jsp notAvailable.jsp	FilmForm	CheckAvailabilityAction	getObjectCount	
Search Movie By Actor Name	byActorSearch.jsp actorList.jsp actorDetails.jsp noActorsFound.jsp	ActorForm	ActorAction ActorDetailsAction	getObjects getObjectById	
Check Out	checkOut.jsp checkOutSuccess.jsp	RentalForm	CheckOutAction	checkOutFilms	
Check In	checkIn.jsp checkInSuccess.jsp	RentalForm	CheckInAction	checkInFilms	

Table 2: Struts Framework Files and Work Flow

Some files are used as global forwards, which are accessible from all the pages. For example,

- welcome.jsp
- login.jsp and logout.jsp
- menu.jsp
- error.jsp

Properties File

MessageResources.properties file is used to display messages to the user. It is stored in an application's *WEB-INF/classes* folder and is specified in *struts-config.xml*. At run-time, it helps replace portions of a message string with arguments specified at run time.

Business Logic Tier

The business logic is implemented in the business object class *MovieRentalBOImpl*, which implements the *MovieRentalBO* interface. *MovieRentalBOImpl* works as a stateless singleton business object and is responsible for data access and storage of various domain model objects (POJOs), using the JDX OR-Mapper methods. To make the application thread-safe, eager instantiation of the singleton business object is done. The static method *getInstance()* return the same previously initialized instance.

```
public class MovieRentalBOImpl implements MovieRentalBO {
  private static JXResourcePool jxResourcePool_ = null;
  // Eager instantiation of Singleton movieRentalBoImplInstance_
  private static MovieRentalBOImpl movieRentalBoImplInstance_ = new MovieRentalBOImpl();

  // private constructor
  private MovieRentalBOImpl () {
       jxResourcePool_ = InitMovieRentalServlet.getJXResourcePool();
    }

  public static MovieRentalBOImpl getInstance() throws Exception {
      if (jxResourcePool_ == null) {
                    throw new Exception(". . . ");
      }
      return movieRentalBoImplInstance_;
  }
  ////// other methods ///////
}
```

Code Fragment 3: MovieRentalBOImpl Class in Singleton Pattern

As discussed earlier, the JXResourcePool is initialized by the start-up servlet when the web server engine is started. This resource pool helps create multiple JXResource handles and provides methods to share these handles in a thread-safe way.

The method *checkoutJXResource()* provided in the business object (*MovieRentalBOImpl*) gets a free JXResource from the pre-initialized pool of JXResources:

```
private JXResource checkoutJXResource() throws Exception {

       JXResource jxResource = (JXResource) jxResourcePool_.getResource();
       if (jxResource == null) {
             throw new Exception ("JXResources exhausted.");
       }
       return  jxResource;
}
```

Code Fragment 4: Method checkOutJXResource() (File: MovieRentalBOImpl.java)

Similarly, the method *checkinJXResource()* releases a JXResource back into the pool:

```
private void checkinJXResource(JXResource jxResource) {

       if (jxResource != null) {
             jxResourcePool_.releaseResource( jxResource );
      }
 }
```

Code Fragment 5: Method checkinJXResource() (File: MovieRentalBOImpl.java)

Other methods in the *MovieRentalBOImpl* class are for data access and storage to execute part of the business logic (see Appendix D). Some methods are generalized (e.g., getObjects, getObjectById) while others are application specific (e.g., checkOutFilms, checkInFilms). All methods have the following pattern:

- Checkout JXResource, get the JXSession handle (for transactional methods) and the JDXS handle (for methods like query, insert, update, delete, etc.),
- Optionally start a transaction
- Use JDX APIs to interact with the database
- Commit the transaction, if started earlier, and
- Check-in JXResource.

Here is an example:

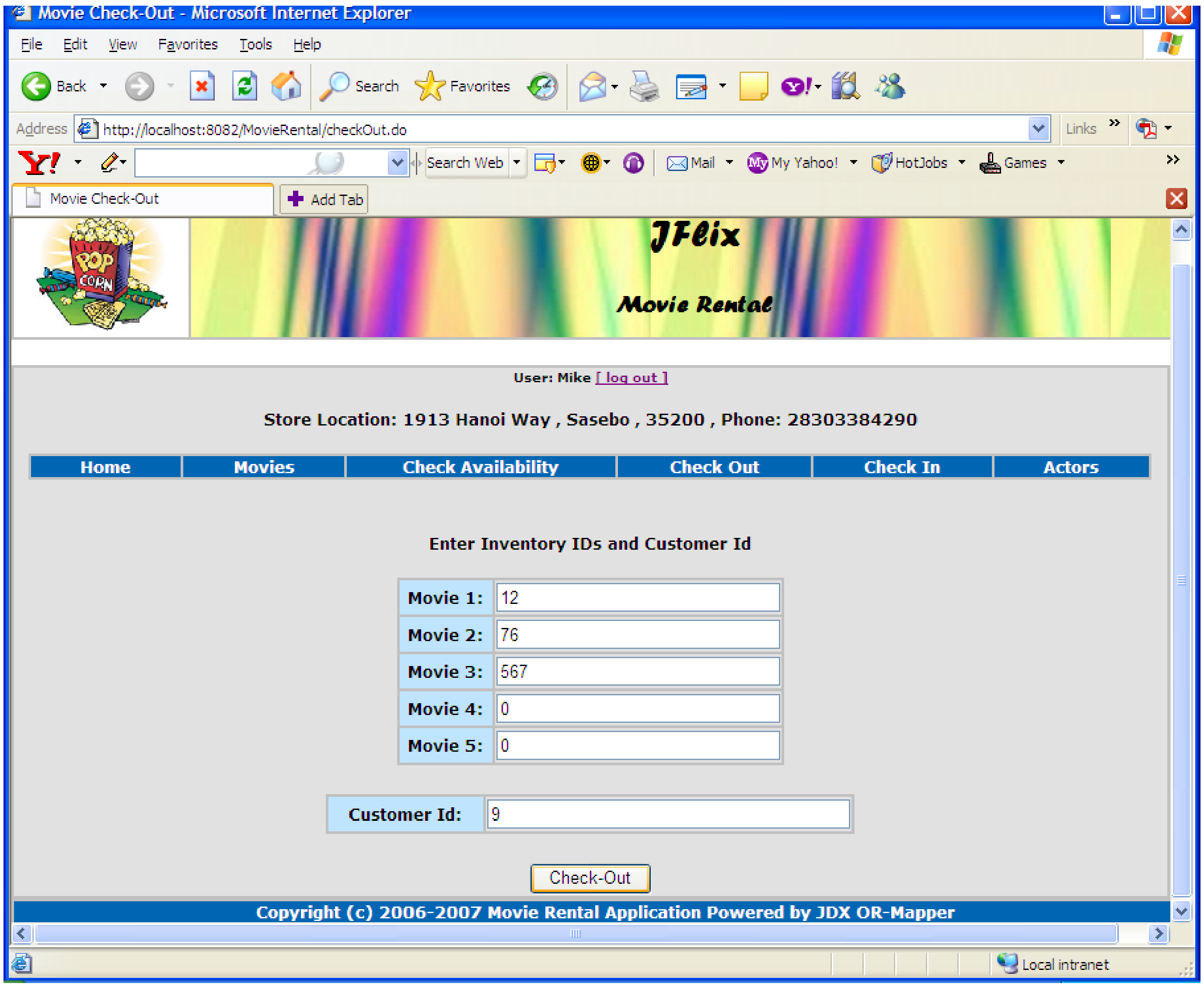
```
public Object getObjectById(. . . .) throws Exception {

       JXResource jxResource = checkoutJXResource();
       JXSession jxSessionHandle = jxResource.getJXSessionHandle();
       JDXS jdxHandle = jxResource.getJDXHandle();

       try {
             // Using jdxHandle, retrieve an object based on its id…
       }
       catch (Exception ex)  {
             // Throw exception
       }
       finally {
             checkinJXResource(jxResource);
      }
       return object;
}
```

Code Fragment 6: Method getObjectById() (File: MovieRentalBOImpl.java)

Movie Check Out: Use Case Details

This section describes the entire process of checking out movies.

The user interface starts with the *checkOut.jsp* page displaying the form to enter inventory ids of the movies to be checked out, and the customer id. As per the current UI design, a maximum of five movies can be checked out at one time, although that number is not a technical limitation.


Movie Check-Out - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
http://localhost:8082/MovieRental/checkOut.do
JFlix
Movie Rental
Home Movies Check Availability Check Out Check In Actors
Check-Out

User: Mike [log out]

Store Location: 1913 Hanoi Way , Sasebo , 35200 , Phone: 28303384290

Enter Inventory IDs and Customer Id

Movie 1:	12
Movie 2:	76
Movie 3:	567
Movie 4:	0
Movie 5:	0
Customer Id:	9

Copyright (c) 2006-2007 Movie Rental Application Powered by JDX OR-Mapper

Figure 4: Check Out Form

After the user enters inventory ids and customer id and hits the "Check Out" button, the struts framework invokes the *execute()* method of the CheckOutAction class with the user input encapsulated in the request parameter:

```
<layout:submit reqCode=”/checkOut.do”
      onclick=”document.forms[0].opCOdeOut.value=’checkOut’”>
            <layout:message key=”Check Out”/>
</layout:submit>
```

Code Fragment 7: Code for Check Out Button from (File: checkOut.jsp)

```
<action>
      path=”/checkOut”
      type=”movierental.action.CheckOutAction”
      name=”RentalForm”
      scope=”request”
      input=”/error.jsp”>

      <forward name=”checkedOut” path=”/checkOutSuccess.jsp”/>
</action>
```

Code Fragment 8: Code for ‘checkOut’ action forwards and mapping (File: struts-config.xml)

Within the *execute()* method of the CheckOutAction class, the method *checkOutFilms()* of the MovieRentalBO interface is invoked by passing inventory ids, customer id, and staff member id, as parameters. *CheckOutFilms()* method returns a list of movies successfully checked out by *MovieRentalBOImpl* class.

```
public ActionForward execute(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response) throws Exception, ServletException {
      . . .

      // get the singleton instance of MovieRentalBOImpl
      MovieRentalBO movierentalbo = MovieRentalBOImpl.getInstance();

      Vectors films = movierentalbo.checkOutFilms(invIds, customerId, staff_Id);
      request.setAttribute("moviesCheckedOut", films);
      return (mapping.findForward("checkedOut"));

      . . .
}
```

Code Fragment 9: Code from action method execute() (File: CheckOutAction.java)

As shown in the code fragment below, the *checkOutFilms()* method of the *MovieRentalBOImp*l class executes the following business logic:

- Checkout JXResource
- Start a transaction
- Check validity of inventory ids and customer id
- For each inventory ID, invoke JDX *insert* method to insert a new record in the *rental* table
- Invoke JDX *update2* method to update the *inventory* table by setting the value of the *in_stock* column to *false* for all the rows corresponding to the given inventory ids.
- Get the list of checked out movies
- Commit the transaction
- Return the list of checked out movies to the *execute()* method in action class.

```
public Vector checkOutFilms(ArrayList inventory_ids, short customer_id,
                                  short staff_id) throws Exception {

  JXResource jxResource = checkoutJXResource();
  JXSession jxSesSsionHandle = jxResource.getJXSessionHandle();
  JDXS jdxHandle = jxResource.getJDXHandle();

  String invPredicate = validateCheckInCheckOutParameters
                          (inventory_ids, customer_id, jdxHandle);
  jxSessionHandle.tx_begin();
  Vector films = null; // List of successfully checked out films
  try {
    int countIds = inventory_ids.size();
    Short inv_id;

    // Do the checkout processing
    // First create and insert new Rental objects for each inventory item
    Timestamp ts = new Timestamp((new java.util.Date()).getTime());
    for (int i = 0; i < inventory_ids.size(); i++) {
      inv_id = (Short) inventory_ids.get(i);
      Rental rental = new Rental(ts, inv_id, customer_id, staff_id);
      jdxHandle.insert(rental, JDXS.FLAG_SHALLOW, null);
    }

    // Update in_stock value of all the inventory items to false
    String invStockPredicate = invPredicate + " AND in_stock=1";
    int count = updateInventoryStatus(invStockPredicate, false, jdxHandle);
    if (count < countIds) {
       throw new Exception((countIds-count)+ "Inventory Id(s) out of stock.”);
    }

    // Now get the vector of the checked out films to be returned
    // back to the caller
    films = getFilmsForInventoryIds(invPredicate, countIds, jdxHandle);
    jxSessionHandle.tx_commit();
  } catch (Exception ex)  {
    jxSessionHandle.tx_rollback();
    throw ex;
  }
  finally {
    checkinJXResource(jxResource);
  }
  return films;
}
```

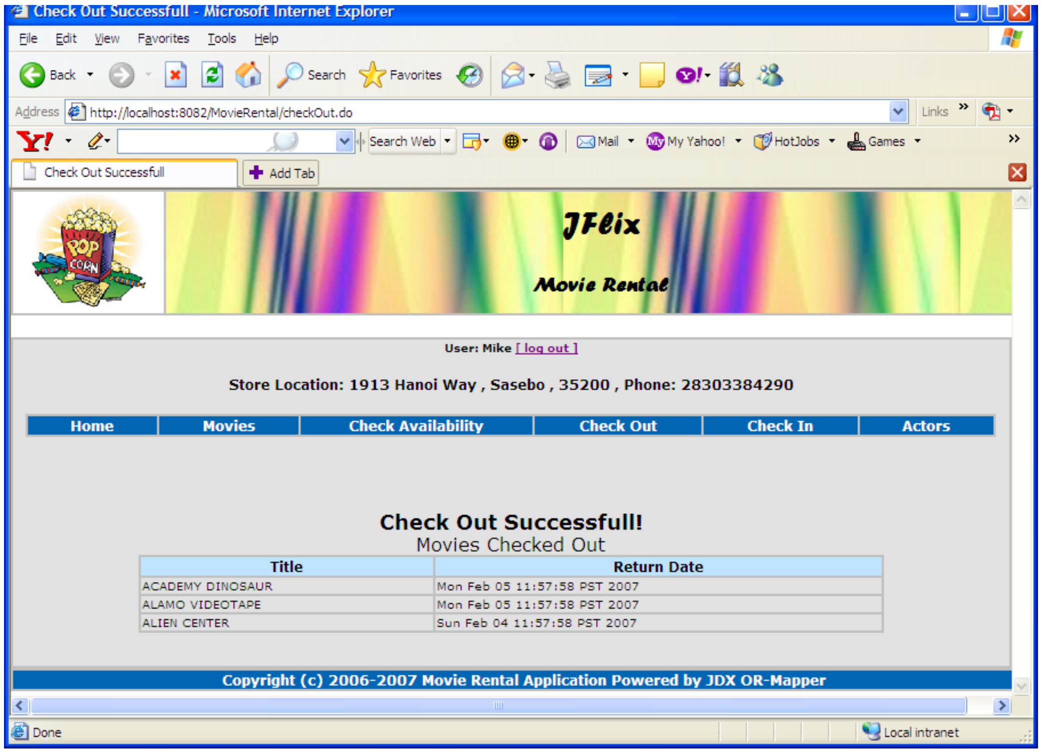
Code Fragment 10: Code from method checkOutFilms() (file: MovieRentalBOImpl.java)

The *getFilmsForInventoryIds()* method first retrieves the list of inventory objects corresponding to the checked-out or checked-in inventory items. Then, employing the getObjectById() method, it retrieves the film objects one-by-one for each of the inventory objects using the corresponding film_id value.

```
static private Vector getFilmsForInventoryIds(String inventoryPredicate, int
                                  countIds, JDXS jdxHandle) throws Exception {

  // First get the list of inventory items to get the film ids
  Vector InventoryList = jdxHandle.query("movierental.Inventory",
                            inventoryPredicate, countIds, JDXS.FLAG_SHALLOW, null);
  // Now retrieve the films
  Inventory inventory;
  Vector films = new Vector();
  Object film = null;

  for (int i = 0; i < InventoryList.size(); i++) {
    inventory = (Inventory) InventoryList.elementAt(i);
    short filmId = inventory.getFilm_id();
    Object id = ObjectId.createObjectId("movierental.Film;film_id="+filmId);
    film = jdxHandle.getObjectById (id, true, JDXS.FLAG_SHALLOW, null);
    films.add(film);
  }
  return films;
}
```

Code Fragment 11: Code from method getFilmsForInventoryIds() (File: MovieRentalBOImlp.java)

Finally, the *execute()* method initializes the request context with movie list and passes control to *checkOutSuccess.jsp* using the action forward “checkedOut”. The page displays names of the movies checked out and their return dates.



User: Mike [log out]

Store Location: 1913 Hanoi Way , Sasebo , 35200 , Phone: 28303384290

Check Out Successfull!

Movies Checked Out

Title	Return Date
ACADEMY DINOSAUR	Mon Feb 05 11:57:58 PST 2007
ALAMO VIDEOTAPE	Mon Feb 05 11:57:58 PST 2007
ALIEN CENTER	Sun Feb 04 11:57:58 PST 2007

Copyright (c) 2006-2007 Movie Rental Application Powered by JDX OR-Mapper

Figure 5: Check Out Success Screen

Summary

The JFlix movie rental application demonstrates how JDX OR-Mapper can be used in combination with the Struts framework to develop a sophisticated web-application. This application also demonstrates that JDX can easily work with an existing database schema like MySQL *Sakila*. JDX provides an efficient way of communication between a Java object model (POJO) and a SQL relational model. This approach also eliminates the need to create separate Data Transfer Objects (DTOs) for each domain model object for passing data between the presentation tier and the business logic tier.

Exploiting JDX's intuitive, flexible, and powerful APIs, makes the business logic code simple, concise, and easy to maintain The same logic, if implemented using JDBC, would be much more complex, error-prone, and difficult to maintain. Using JXResourcePool facility makes it very easy to efficiently share the underlying mapping and database resources in a thread-safe manner. All in all, JDX OR-Mapper improves developer productivity and helps create high-performance applications quickly.

JFlix Movie Rental application is shipped with JDX OR-Mapper software. A free evaluation version of JDX is available from Software Tree's web site at http://www.softwaretree.com.

Acknowledgements

I would like to thank Mr. Damodar Periwal, the architect of JDX, for his valuable guidance and help throughout the design and implementation of this application. I would also like to thank Mr. Nikhil Samdani for reviewing this article and offering valuable feedback to improve the contents and the presentation of the material.

References

1. MVC Application Control Flow Diagram: http://www.matthiasbook.de/papers/dialogcontrol-it2002/2.html

2. Software Tree Website: www.softwaretree.com

3. Beyond JDBC: A White Paper: http://www.softwaretree.com/products/jdx/whitepaper/BeyondJDBC.pdf

4. The KISS Principles for OR-Mapping Products: A White Paper: http://www.softwaretree.com/products/njdx/whitepaper/KISSPrinciples.pdf

Trademarks

Appendix

Appendix A: Start-up Servlet *web.xml*

Here is an excerpt from *web.xml* to show start-up servlet configuration for the application. One is the standard struts action servlet and the other one (`MovieRentalStartupServlet`) is the servlet for JXResourcePool initialization:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
       <!-- Standard Action Servlet Definition -->
       <servlet>
              <servlet-name>action</servlet-name>
              <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>

              <init-param>
                     <param-name>config</param-name>
                     <param-value>/WEB-INF/struts-config.xml</param-value>
              </init-param>
              <load-on-startup>2</load-on-startup>
       </servlet>

       <!-- Start-up servlet definition for JDX -->
       <servlet>
              <servlet-name>MovieRentalStartupServlet</servlet-name>
              <servlet-class>movierental.InitMovieRentalServlet</servlet-class>
              <init-param>
                            <param-name>ORMPoolName</param-name>
                            <param-value>movierentalORMPool</param-value>
              </init-param>
              <init-param>
                     <param-name>JDBCDriver</param-name>
                     <param-value>com.mysql.jdbc.Driver</param-value>
              </init-param>
              <init-param>
                     <param-name>ServiceURL</param-name>
                     <param-value>
       JDX:jdbc:mysql://localhost/sakila;user=root;JDX_ORMFile=jflix.jdx;JDX_FINE_CO
       NN_POOLING=YES; JDX_DBTYPE=MYSQL;DEBUG_LEVEL=5</param-value>
              </init-param>
              <init-param>
                     <param-name>JDX_ORMFILE</param-name>
                     <param-value>/WEB-INF/classes/jflix.jdx</param-value>
              </init-param>
              <load-on-startup>2</load-on-startup>
       </servlet>

<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
       <servlet-name>MovieRentalStartupServlet</servlet-name>
       <url-pattern>*.do</url-pattern>
</servlet-mapping>
</web-app>
```

Appendix B: Struts Configuration File *struts-config.xml*

This is an excerpt from *struts-config.xml* showing form-bean and action-mapping specifications for the check out films use case:

```
<!-- ======================================= Form Bean Definitions -->

    <form-beans>
           <form-bean name="RentalForm"
            type="movierental.actionform.RentalForm"/>
    </form-beans>

<!-- ==================================== Action Mapping Definitions -->

 <action-mappings>

<action
      path="/checkOut"
      type="movierental.action.CheckOutAction"
      name="RentalForm"
      scope="request"
      input="/error.jsp">

      <forward name="checkOut"         path="/checkOut.jsp"/>
      <forward name="checkedOut"       path="/checkOutSuccess.jsp"/>
      <forward name="filmNotAvailable" path="/notAvailable.jsp"/>
</action>

 </action-mappings>

</struts-config>
```

Appendix C: ORMFIle *jflix.jdx*

This is an excerpt from *jflix.jdx* showing OR-Mapping specifications for the classes used in the *Check Out* films use case. Please note that the SQLMAP specification needs to be done for only those attributes whose corresponding columns are nullable. If all the columns had been non-nullable (a default assumed by JDX), the mapping specification would have become more compact.

```
CLASS movierental.Film TABLE film
    PRIMARY_KEY film_id
    RELATIONSHIP actors REFERENCES VectorActors WITH film_id
    RELATIONSHIP categories REFERENCES VectorCategory WITH film_id
    RELATIONSHIP inventory REFERENCES VectorInventory WITH film_id
    SQLMAP FOR release_year NULLABLE
    SQLMAP FOR original_language_id NULLABLE
    SQLMAP FOR length NULLABLE
    SQLMAP FOR rating NULLABLE
    SQLMAP FOR last_update NULLABLE
    SQLMAP FOR description NULLABLE
    SQLMAP FOR special_features NULLABLE
;
CLASS movierental.Inventory TABLE inventory
    PRIMARY_KEY inventory_id
    SQLMAP FOR last_update NULLABLE
;
COLLECTION_CLASS VectorInventory COLLECTION_TYPE VECTOR ELEMENT_CLASS
       movierental.Inventory ELEMENT_TABLE inventory
    PRIMARY_KEY film_id
;
CLASS movierental.Customer TABLE customer
    PRIMARY_KEY customer_id
    SQLMAP FOR last_update NULLABLE
    SQLMAP FOR email NULLABLE
    RELATIONSHIP store REFERENCES movierental.Store WITH store_id
;
CLASS movierental.Staff TABLE staff
    PRIMARY_KEY staff_id
    SQLMAP FOR last_update NULLABLE
    SQLMAP FOR picture NULLABLE
    SQLMAP FOR email NULLABLE
    SQLMAP FOR password NULLABLE
;
CLASS movierental.Rental TABLE rental
    PRIMARY_KEY rental_id
    RDBMS_GENERATED rental_id last_update
;
```

Appendix D: Business Logic Layer Methods in *MovieRentalBO.java*

Here are the signatures of the main methods of the business logic layer interface (*MovieRentalBO*) that uses JDX for data access and storage of domain model objects.

```
public interface MovieRentalBO {

  /**
    * Method getObjects() takes className, predicate, numOfObj, and deep as input
    * and returns a vector list of all the qualifying objects of the given class
    * from the database. If deep parameter is true, all the related objects are
    * also fetched.
   **/
  public  Vector getObjects(String className, String predicate, long numOfObj,
                            boolean deep, Vector details) throws Exception;

  /**
    * Method getObjectById() takes className and a primary key predicate in the
    * form of pkey1=val1[;pkey2=val2]) as input and returns the qualifying
    * object from the database.  It returns null if no qualifying object is found.
    * If deep parameter is true, all the related objects are also fetched.
   **/

   public Object getObjectById(String className, String primaryKeyPredicate,
                               boolean deep, Vector details) throws Exception;

   /**
     * Method getObjectCount() takes className, attrib, predicate, and deep as
     * input and returns the number of qualifying objects of the given class.
    **/
   public int getObjectCount(String className, String attrib, String predicate,
                             boolean deep) throws Exception;

  /**
    * Method checkOutFilms() checks out a set of movies. It takes as input a vector
    * of movie inventory ids, a customer id, and a store clerk id.  It inserts a
    * new Rental object for each checked out movie and sets the in_stock value to
    * false for the appropriate inventory objects.
   **/
   public Vector checkOutFilms(ArrayList inventory_ids, short customer_id,
                               short staff_id) throws Exception;

   /**
     * Method checkInFilms() checks in a set of  movies. It takes as input a vector
     * of inventory ids of the movies to be checked in and a customer id.  It
     * updates the Rental object (sets return_date to today) for each of the
     * checked in movie and sets the in_stock value to true for the appropriate
     * inventory object.
    **/
   public Vector checkInFilms(ArrayList inventory_ids, short customer_id)
                                                                 throws Exception;


}
```