|  |  |
| --- | --- |
| 제4편 자바 프로그래밍 워크숍 (PART FOUR Java Programming Workshop)  절 제목(Section Title) | |
| 작성자 (전자우편) | 문서코드(없으면 생략)  대외비 |

문서 정보

요약

내용 (Normal)

키워드

키워드 1, 키워드 2

문서 버전 및 이력

|  |  |  |  |
| --- | --- | --- | --- |
| 버전 | 일자 | 이력사항 | 작성자 |
| 0.8 | 일자 | 재작성 | 작성자 |
| 1.0 | 일자 | 최종 배포 | 작성자 |

목차

[제4편 자바 프로그래밍 워크숍 (PART FOUR Java Programming Workshop) 1](#_Toc330103935)

[1.1 소개(Introduction) 1](#_Toc330103936)

[CHAPTER 1 학사관리 어플리케이션의 소개(Introducing the Learning Management Application) 1](#_Toc330103937)

[CHAPTER 1 DAO의 이해(Core J2EE Patterns - Data Access Object) 1](#_Toc330103938)

[1.2 Context 1](#_Toc330103939)

[1.3 Problem 1](#_Toc330103940)

[1.4 Forces 1](#_Toc330103941)

[1.5 Solution 1](#_Toc330103942)

[1.6 Structure 1](#_Toc330103943)

[1.6.1 Participants and Responsibilities 1](#_Toc330103944)

[1.6.2 Strategies 1](#_Toc330103945)

[1.7 Consequences 1](#_Toc330103946)

[1.8 Sample Code 1](#_Toc330103947)

[1.8.1 Implementing Data Access Object pattern 1](#_Toc330103948)

[1.8.2 Implementing Factory for Data Access Objects Strategy 1](#_Toc330103949)

[1.9 Related Patterns 1](#_Toc330103950)

[CHAPTER 3 Java DB(Apache Derby)의 이해 1](#_Toc330103951)

[CHAPTER 4 Java DB를 이용한 데스크톱 애플리케이션(Using Java DB in Desktop Applications) 1](#_Toc330103952)

[CHAPTER 5 Fly by Night Service 1](#_Toc330103953)

[1. Introduction to Apache Derby 1](#_Toc330103954)

[1.1 The Apache Derby project 1](#_Toc330103955)

[1.2 데이터베이스 시스템 1](#_Toc330103956)

[1.3 데이터베이스 역할 1](#_Toc330103957)

[1.4 Apache Derby 프로젝트란 무엇인가? 1](#_Toc330103958)

[1.5 Apache Derby 데이터베이스 개요 1](#_Toc330103959)

[1.5.1 Apache Derby 다운로드 하기 1](#_Toc330103960)

[1.5.2 Apache Derby 설치하기 1](#_Toc330103961)

[1.5.3 Apache Derby 사용하기 1](#_Toc330103962)

[2. Apache Derby를 이용한 자바 데이터베이스 개발 1](#_Toc330103963)

[2.1 JDBC 소개 1](#_Toc330103964)

[2.1.1 Apache Derby와 JDBC 1](#_Toc330103965)

[2.2 자바 애플리케이션과 Derby 데이터베이스로 연결하기 1](#_Toc330103966)

[2.3 데이터베이스 메타데이터 1](#_Toc330103967)

[2.4 문제 1](#_Toc330103968)

[2.5 연결하기 1](#_Toc330103969)

[2.6 요약 1](#_Toc330103970)

[3. Java database development with Apache Derby 1](#_Toc330103971)

[3.1 JDBC query execution: an overview 1](#_Toc330103972)

[3.2 Executing a query 1](#_Toc330103973)

[제목 1 1](#_Toc330103974)

[3.3 (제목 2) 지원사항 1](#_Toc330103975)

[3.3.1 (제목 3) 배포 방식 1](#_Toc330103976)

[3.3.2 번호매김 테스트용 1](#_Toc330103977)

[3.4 제목 2 1](#_Toc330103978)

[3.4.1 점목록 테스트용 1](#_Toc330103979)

[3.4.2 표 테스트용 1](#_Toc330103980)

[3.5 UI 제어 설정 1](#_Toc330103981)

[3.5.1 목록 및 소스 관련 테스트용 1](#_Toc330103982)

[3.5.2 업데이트 완료 표시 설정 1](#_Toc330103983)

[참고문헌 1](#_Toc330103984)

[용어정리 1](#_Toc330103985)

표 및 그림 목록

표 목록

[표 1-1 전달 인자 설명 1](#_Toc290296826)

그림 목록

[그림 1-1 Install/Update 확인 1](#_Toc290296827)

제4편 자바 프로그래밍 워크숍 (PART FOUR Java Programming Workshop)

## 소개(Introduction)

제3편 자바 프로그래밍에서는

- 이하 편집 후 전체 개요 요약 기술 -

자바 프로그래밍 워크숍은 자바 애플리케이션 프로그램의 개발 방법을 설명합니다.

.Java DB(Apache Derby)  
Derby의 임베디드 Java Database Connectivity (JDBC) 드라이버를 사용하여 임베디드 Derby 데이터베이스를 생성하고 연결하여 필요한 데이터를 조회 및 입력, 수정, 삭제하는 방법을 설명하며 데이터베이스 메타데이터와, Derby에서 생성된 SQL 에러와 경고를 자바 애플리케이션에서 해결하는 방법도 설명합니다.

.클라이언트/서버 프로그래밍  
Standalone 방식의 독립실행프로그램이 Socket과 RMI 통신을 통해 클라이언트와 서버로 구분되어 개발되는 방법을 설명합니다.

.디차인패턴  
DAO Factory MVC

# CHAPTER 1 학사관리 어플리케이션의 소개(Introducing the Learning Management Application)

이 장에서는

- 이하 편집 후 전체 개요 요약 기술 –

<http://tutorials.jenkov.com/java-persistence/index.html>

<http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html>

<http://stackoverflow.com/questions/1612334/difference-between-dto-vo-pojo-javabeans>

# CHAPTER 1 DAO의 이해(Core J2EE Patterns - Data Access Object)

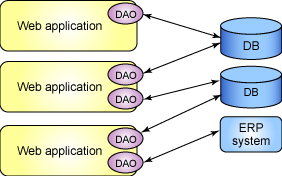
이 장에서는

- 이하 편집 후 전체 개요 요약 기술 -

J2EE 개발자들은 Data Access Object (DAO) 디자인 패턴을 사용하여 저수준의 데이터 액세스 로직과 고급 비즈니스 로직을 분리한다. DAO 패턴을 구현하는 것은 단순히 데이터 액세스 코드를 작성하는 것 이상이다.

지난 18개월 동안, 재능 있는 소프트웨어 엔지니어들과 함께 웹 기반의 공급체인 관리 애플리케이션을 구현했다. 우리가 만든 애플리케이션은 선적 상황, 공급 체인 메트릭스, 창고 재고, 운송장, 프로젝트 관리 데이터, 사용자 프로파일 등의 광범위한 데이터에 접근했다. JDBC API를 사용하여 회사의 다양한 데이터베이스 플랫폼에 연결했고 이 애플리케이션에 DAO 디자인 패턴을 적용했다.

그림 1은 애플리케이션과 데이터 소스의 관계이다:



DAO의 기초

DAO 패턴은 표준 J2EE 디자인 패턴들 중 하나이다. 이 패턴을 사용하여 저수준 데이터 액세스와 고급 비지니스 로직을 분리한다. 전형적인 DAO 구현에는 다음과 같은 요소들이 있다:

* DAO 팩토리 클래스
* DAO 인터페이스
* DAO 인터페이스를 구현하는 구체적 클래스
* 데이터 전송 객체들(밸류(value) 객체)

구체적인 DAO 클래스에는 특정 데이터 소스로 부터 데이터에 액세스하는데 쓰이는 로직이 포함되어 있다.

트랜잭션 경계설정(demarcation)

DAO는 트랜잭션 객체들이라는 것을 반드시 기억해야 한다. DAO에 의해 수행되는 각각의 작동(데이터 구현, 업데이트, 삭제)은 트랜잭션과 관련있다. 따라서 트랜잭션 경계설정(demarcation) 개념은 매우 중요하다

트랜잭션 경계설정은 트랜잭션 영역들이 정의 되는 방식이다. J2EE 스팩은 두 가지 모델의 트랜잭션 경계설정을 설명하고 있다. (표 1):

표 1. 트랜잭션 경계설정

|  |  |
| --- | --- |
| 선언적(Declarative) 트랜잭션 경계설정 | 프로그램에 입각한(Programmatic) 트랜잭션 경계설정 |
| 프로그래머는 EJB 전개 디스크립터를 사용하여 트랜잭션 애트리뷰트를 선언한다. | 프로그래머는 트랜잭션 로직을 코딩해야한다. |
| 런타임 환경(EJB 컨테이너)은 트랜잭션을 자동으로 관리하기 위해 이 애트리뷰트를 사용한다. | 이 애플리케이션은 API를 통해 트랜잭션을 제어한다. |

프로그램에 입각한(Programmatic) 트랜잭션 경계설정을 중점적으로 설명하겠다.

디자인 고려사항

앞서 언급했지만, DAO는 트랜잭션 객체이다. 전형적인 DAO는 구현, 업데이트, 삭제 같은 트랜잭션 작동을 수행한다. DAO를 설계할 때 다음 사항들을 점검한다:

* 트랜잭션은 어떻게 시작하는가?
* 트랜잭션은 어떻게 끝나는가?
* 트랜잭션 시작을 담당하는 객체는 무엇인가?
* 트랜잭션 종료를 담당하는 객체는 무엇인가?
* DAO가 트랜잭션의 시작과 종료를 담당해야 하는가?
* 이 애플리케이션이 다중의 DAO를 통해 데이터에 액세스해야 하는가?
* 트랜잭션에 포함될 DAO의 수는?
* 하나의 DAO가 또 DAO에 대한 메소드를 호출할 수 있는가?

이러한 질문들에 대한 답을 알고있다면 자신의 DAO에 가장 잘 맞는 트랜잭션 경계설정 전략을 선택하는데 도움이 된다. DAO에는 두 가지 주요한 트랜잭션 경계설정 전략이 있다. 하나는 DAO가 트랜잭션의 경계를 설정하도록 하는 것이다. 또 다른 방법은 트랜잭션 경계설정을 DAO의 메소드를 호출하는 객체에 맡기는 것이다. 전자를 선택한다면 DAO 클래스 안에 트랜잭션 코드를 임베딩해야 한다. 후자의 방법을 선택한다면 트랜잭션 경계설정 코드는 DAO 클래스의 외부에 있게 될 것이다. 코드 예제를 보고 이 두 가지 방법을 자세히 보도록 하자.

Listing 1- DAO와 두 개의 데이터 작동;구현 및 업데이트:

Listing 1. DAO 메소드

|  |
| --- |
| public void createWarehouseProfile(WHProfile profile);  public void updateWarehouseStatus(WHIdentifier id, StatusInfo status); |

Listing 2은 간단한 트랜잭션이다. 트랜잭션 경계설정 코드는 DAO 클래스 외부에 있다. 이 예제에서 콜러(caller)가 다중의 DAO 작동들을 이 트랜잭션 내에서 결합하는 방법을 주목해보자.

Listing 2. 콜러(Caller)에 의해 관리되는 트랜잭션

|  |
| --- |
| tx.begin(); // start the transaction  dao.createWarehouseProfile(profile);  dao.updateWarehouseStatus(id1, status1);  dao.updateWarehouseStatus(id2, status2);  tx.commit(); // end the transaction; |

이 트랜잭션 경계설정 전략은 단일 트랜잭션에서 다중의 DAO에 액세스 해야하는 애플리케이션에 특별히 어울린다.

JDBC API 또는 Java Transaction API (JTA)를 사용하여 트랜잭션 경계설정을 구현할 수 있다. JDBC 트랜잭션 경계설정은 JTA 보다 단순하다. 하지만 JTA는 보다 유연하다.

JDBC를 이용한 트랜잭션 경계설정

JDBC 트랜잭션은Connection 객체를 사용하여 제어된다. JDBC Connection 인터페이스 java.sql.Connection)는 두 개의 트랜잭션 모드를 제공한다. (auto-commit과 manual commit). java.sql.Connection은 트랜잭션 제어에 다음의 메소드를 제공한다:

* public void setAutoCommit(boolean)
* public boolean getAutoCommit()
* public void commit()
* public void rollback()

Listing 3은 JDBC API를 사용하여 트랜잭션의 경계설정을 하는 방법이다

Listing 3. JDBC API를 이용한 트랜잭션 경계설정

import java.sql.\*;

import javax.sql.\*;

// ...

DataSource ds = obtainDataSource();

Connection conn = ds.getConnection();

conn.setAutoCommit(false);

// ...

pstmt = conn.prepareStatement("UPDATE MOVIES ...");

pstmt.setString(1, "The Great Escape");

pstmt.executeUpdate();

// ...

conn.commit();

// ...

JDBC 트랜잭션 경계설정을 이용하면 여러 개의 SQL 문장을 하나의 트랜잭션으로 결합할 수 있다. JDBC 트랜잭션의 단점 중 하나는 트랜잭션 범위가 하나의 데이터베이스 연결로 제한되어 있다는 점이다. JDBC 트랜잭션은 다중 데이터베이스로 확장할 수 없다. 다음에는 JTA를 사용한 트랜잭션 경계설정이다.

JDBC와 JTA를 이용한 트랜잭션 경계설정에 대해 이야기했다. 각 접근방식 대로 장점이 있기 때문에 자신의 애플리케이션에 가장 알맞는 것을 선택해야 한다.

최근 많은 프로젝트에서 우리팀은 JDBC API를 사용하여 DAO 클래스를 구현했다. 이 DAO 클래스는 다음과 같이 요약된다:

트랜잭션 경계설정 코드는 DAO 클래스 안으로 임베딩된다.

DAO 클래스는 트랜잭션 경계설정에 JDBC API를 사용한다.

콜러가 트랜잭션 경계를 설정할 방법은 없다.

트랜잭션 범위는 하나의 JDBC Connection으로 제한된다.

로깅과 DAO

잘 구현된 DAO 클래스는 런타임 작동에 대한 세부사항을 파악하기 위해 로깅(logging)을 사용한다. 예외, 설정 정보, 커넥션 상태, JDBC 드라이버 메타데이터, 쿼리 매개변수 중 어떤 것이든 선택하여 기록해야 한다. 기록은 모든 개발 단계에 유용하다.

로깅 라이브러리 선택하기

많은 개발자들은 단순한 형식의 로깅을 사용한다: System.out.println과 System.err.println. Println 문장은 빠르고 편리하지만 완벽한 로깅 시스템은 제공하지 않는다. 표 2는 자바 플랫폼을 위한 로깅 라이브러리이다:

표 2.자바 플랫폼을 위한 로깅 라이브러리

|  |  |  |
| --- | --- | --- |
| 로깅 라이브러리 | 오픈소스여부 | URL |
| java.util.logging | No | http://java.sun.com/j2se/ |
| Jakarta Log4j | Yes | http://jakarta.apache.org/log4j/ |
| Jakarta Commons Logging | Yes | http://jakarta.apache.org/commons/logging.html |

java.util.logging은 J2SE 1.4 플랫폼을 위한 표준 API이다. Jakarta Log4j가 더 많은 기능성과 유연성을 제공한다는 것에는 많은 개발자들이 동의하고 있다. java.util.logging의 이점 중 하나는 J2SE 1.3과 J2SE 1.4 플랫폼을 지원한다는 것이다.

Jakarta Commons Logging은 java.util.logging 과의 연결 또는 Jakarta Log4j에 사용될 수 있다. Commons Logging은 로깅 추상 레이어로서 애플리케이션을 기저의 로깅 구현에서 고립시킬 수 있다. Commons Logging을 사용하여 설정 파일을 변경하여 기저의 로깅 구현을 바꿀 수 있다. Commons Logging은 Jakarta Struts 1.1과 Jakarta HttpClient 2.0에 사용된다.

로깅예제

Listing 7은 DAO 클래스에서 Jakarta Commons Logging을 사용하는 방법이다:

Listing 7. Jakarta Commons Logging

import org.apache.commons.logging.\*;

class DocumentDAOImpl implements DocumentDAO

{

static private final Log log = LogFactory.getLog(DocumentDAOImpl.class);

public void deleteDocument(String id)

{

// ...

log.debug("deleting document: " + id);

// ...

try

{

// ... data operations ...

}

catch (SomeException ex)

{

log.error("Unable to delete document", ex);

// ... handle the exception ...

}

}

}

로깅은 미션 수행에 중요한 애플리케이션의 중요한 일부이다. DAO에서 오류가 생기면 무엇이 잘못되었는지를 이해할 수 있게끔 최고의 정보를 로그가 제공한다. 로깅과 DAO를 결합하면 디버깅과 문제해결은 확실하다.

DAO에서의 예외 핸들링

DAO 패턴을 구현할 때 다음 사항을 자문해보라:

* DAO의 퍼블릭 인터페이스의 메소드들이 검사된 예외를 던지는가?
* 그렇다면 어떤 예외들인가?
* DAO 구현 클래스에서 예외는 어떻게 처리되는가?

DAO 패턴을 작업하는 과정에서 우리 팀은 예외 핸들링에 대한 가이드라인을 개발했다:

* DAO 메소드는 java.lang.Exception을 던져서는 안된다. java.lang.Exception은 너무 일반적이다. 근본 문제에 대한 정보를 제공하지 않을 것이다
* DAO 메소드는 java.sql.SQLException 메소드를 던져서는 안된다. SQLException은 저급 JDBC 예외이다. DAO는 JDBC를 나머지 애플리케이션에 노출하는 것이 아니라 JDBC를 캡슐화 해야한다.
* DAO 인터페이스의 메소드들은 콜러가 예외를 처리할 수 있다고 합리적으로 판단될 때 에만 검사된예외를 던져야 한다. 콜러가 예외를 핸들할 수 없다면 검사되지 않은(런타임) 예외를 던질 것을 고려하라
* 데이터 액세스 코드가 예외를 잡으면 이를 무시하지 말아라. 잡힌 예외를 무시하는 DAO는 문제해결에 애를 먹는다.
* 연쇄 예외를 사용하여 저급 예외를 고급 예외로 트랜슬레이팅 한다.
* 표준 DAO 예외 클래스를 정의하라. Spring Framework (참고자료)은 사전 정의된 DAO 예외 클래스를 제공한다.

구현 예제: MovieDAO

MovieDAO는 이 글에 논의된 모든 기술을 보여주는 DAO이다: 트랜잭션 경계설정, 로깅, 예외 핸들링. 코드는 세 개의 패키지로 나뉜다. (참고자료):

* daoexamples.exception
* daoexamples.movie
* daoexamples.moviedemo

DAO 패턴 구현은 클래스와 인터페이스로 구성된다:

* daoexamples.movie.MovieDAOFactory
* daoexamples.movie.MovieDAO
* daoexamples.movie.MovieDAOImpl
* daoexamples.movie.MovieDAOImplJTA
* daoexamples.movie.Movie
* daoexamples.movie.MovieImpl
* daoexamples.movie.MovieNotFoundException
* daoexamples.movie.MovieUtil

MovieDAO 인터페이스는 DAO의 데이터 작동을 정의한다. 이 인터페이스는 다섯 개의 메소드를 갖고 있다:

* public Movie findMovieById(String id)
* public java.util.Collection findMoviesByYear(String year)
* public void deleteMovie(String id)
* public Movie createMovie(String rating, String year, String, title)
* public void updateMovie(String id, String rating, String year, String title)

daoexamples.movie 패키지에는 두 개의 MovieDAO 인터페이스 구현이 포함되어 있다. 각 구현은 트랜잭션 경계설정에 다른 접근방식을 사용한다. (표 3):

표 3. MovieDAO 구현

|  |  |  |
| --- | --- | --- |
|  | **MovieDAOImpl** | **MovieDAOImplJTA** |
| MovieDAO 인퍼페이스 구현 | Yes | Yes |
| JNDI를 통한 DataSourc 획득 | Yes | Yes |
| DataSource에서 java.sql.Connection 객체 획득 | Yes | Yes |
| DAO가 트랜잭션을 내부적으로 경계설정하는가? | Yes | No |
| JDBC 트랜잭션 사용 | Yes | No |
| XA DataSource 사용 | No | Yes |
| JTA 트랜잭션 참여 | No | Yes |

MovieDAO 데모 애플리케이

이 데모 애플리케이션 이름은 daoexamples.moviedemo.DemoServlet 이다. DemoServlet은 Movie DAO를 사용하여 무비 데이터를 쿼리 및 업데이트 한다.

Listing 8은 싱글 트랜잭션 에서의 JTA-aware MovieDAO와 Java Message Service의 결합 방법이다.

Listing 8. MovieDAO와 JMS 코드 결합

UserTransaction utx = MovieUtil.getUserTransaction();

utx.begin();

batman = dao.createMovie("R",

"2008",

"Batman Reloaded");

publisher = new MessagePublisher();

publisher.publishTextMessage("I'll be back");

dao.updateMovie(topgun.getId(),

"PG-13",

topgun.getReleaseYear(),

topgun.getTitle());

dao.deleteMovie(legallyblonde.getId());

utx.commit();

)

데모를 실행하려면 XA 데이터소스와 비 XA 데이터소스를 애플리케이션 서버에 설정해야 한다. 그런다음 daoexamples.ear 파일을 전개한다.

http://www.ibm.com/developerworks/kr/library/j-dao/

## Context

Access to data varies depending on the source of the data. Access to persistent storage, such as to a database, varies greatly depending on the type of storage (relational databases, object-oriented databases, flat files, and so forth) and the vendor implementation.

## Problem

Many real-world Java 2 Platform, Enterprise Edition (J2EE) applications need to use persistent data at some point. For many applications, persistent storage is implemented with different mechanisms, and there are marked differences in the APIs used to access these different persistent storage mechanisms. Other applications may need to access data that resides on separate systems. For example, the data may reside in mainframe systems, Lightweight Directory Access Protocol (LDAP) repositories, and so forth. Another example is where data is provided by services through external systems such as business-to-business (B2B) integration systems, credit card bureau service, and so forth.

Typically, applications use shared distributed components such as entity beans to represent persistent data. An application is considered to employ bean-managed persistence (BMP) for its entity beans when these entity beans explicitly access the persistent storage-the entity bean includes code to directly access the persistent storage. An application with simpler requirements may forego using entity beans and instead use session beans or servlets to directly access the persistent storage to retrieve and modify the data. Or, the application could use entity beans with container-managed persistence, and thus let the container handle the transaction and persistent details.

Applications can use the JDBC API to access data residing in a relational database management system (RDBMS). The JDBC API enables standard access and manipulation of data in persistent storage, such as a relational database. The JDBC API enables J2EE applications to use SQL statements, which are the standard means for accessing RDBMS tables. However, even within an RDBMS environment, the actual syntax and format of the SQL statements may vary depending on the particular database product.

There is even greater variation with different types of persistent storage. Access mechanisms, supported APIs, and features vary between different types of persistent stores such as RDBMS, object-oriented databases, flat files, and so forth. Applications that need to access data from a legacy or disparate system (such as a mainframe, or B2B service) are often required to use APIs that may be proprietary. Such disparate data sources offer challenges to the application and can potentially create a direct dependency between application code and data access code. When business components-entity beans, session beans, and even presentation components like servlets and helper objects for JavaServer Pages (JSP) pages --need to access a data source, they can use the appropriate API to achieve connectivity and manipulate the data source. But including the connectivity and data access code within these components introduces a tight coupling between the components and the data source implementation. Such code dependencies in components make it difficult and tedious to migrate the application from one type of data source to another. When the data source changes, the components need to be changed to handle the new type of data source.

## Forces

* Components such as bean-managed entity beans, session beans, servlets, and other objects like helpers for JSP pages need to retrieve and store information from persistent stores and other data sources like legacy systems, B2B, LDAP, and so forth.
* Persistent storage APIs vary depending on the product vendor. Other data sources may have APIs that are nonstandard and/or proprietary. These APIs and their capabilities also vary depending on the type of storage-RDBMS, object-oriented database management system (OODBMS), XML documents, flat files, and so forth. There is a lack of uniform APIs to address the requirements to access such disparate systems.
* Components typically use proprietary APIs to access external and/or legacy systems to retrieve and store data.
* Portability of the components is directly affected when specific access mechanisms and APIs are included in the components.
* Components need to be transparent to the actual persistent store or data source implementation to provide easy migration to different vendor products, different storage types, and different data source types.

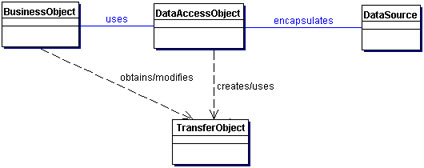
## Solution

Use a Data Access Object (DAO) to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data.

The DAO implements the access mechanism required to work with the data source. The data source could be a persistent store like an RDBMS, an external service like a B2B exchange, a repository like an LDAP database, or a business service accessed via CORBA Internet Inter-ORB Protocol (IIOP) or low-level sockets. The business component that relies on the DAO uses the simpler interface exposed by the DAO for its clients. The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the component and the data source.

## Structure

Figure 9.1 shows the class diagram representing the relationships for the DAO pattern.

   
**Figure 9.1 Data Access Object**

### Participants and Responsibilities

Figure 9.2 contains the sequence diagram that shows the interaction between the various participants in this pattern.

   
**Figure 9.2 Data Access Object sequence diagram**

BusinessObject

The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

DataAccessObject

The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

DataSource

This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

TransferObject

This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

### Strategies

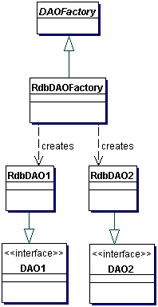
Automatic DAO Code Generation Strategy

Since each BusinessObject corresponds to a specific DAO, it is possible to establish relationships between the BusinessObject, DAO, and underlying implementations (such as the tables in an RDBMS). Once the relationships are established, it is possible to write a simple application-specific code-generation utility that generates the code for all DAOs required by the application. The metadata to generate the DAO can come from a developer-defined descriptor file. Alternatively, the code generator can automatically introspect the database and provide the necessary DAOs to access the database. If the requirements for DAOs are sufficiently complex, consider using third-party tools that provide object-to-relational mapping for RDBMS databases. These tools typically include GUI tools to map the business objects to the persistent storage objects and thereby define the intermediary DAOs. The tools automatically generate the code once the mapping is complete, and may provide other value-added features such as results caching, query caching, integration with application servers, integration with other third-party products (e.g., distributed caching), and so forth.

Factory for Data Access Objects Strategy

The DAO pattern can be made highly flexible by adopting the Abstract Factory [GoF] and the Factory Method [GoF] patterns (see "Related Patterns" in this chapter).

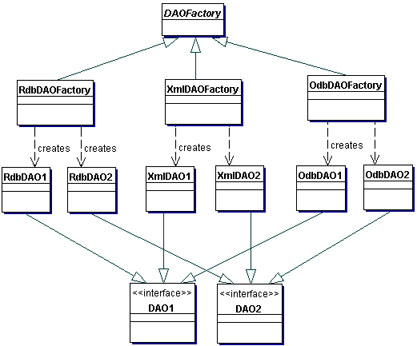
When the underlying storage is not subject to change from one implementation to another, this strategy can be implemented using the Factory Method pattern to produce a number of DAOs needed by the application. The class diagram for this case is shown in Figure 9.3.

   
**Figure 9.3 Factory for Data Access Object strategy using Factory Method**

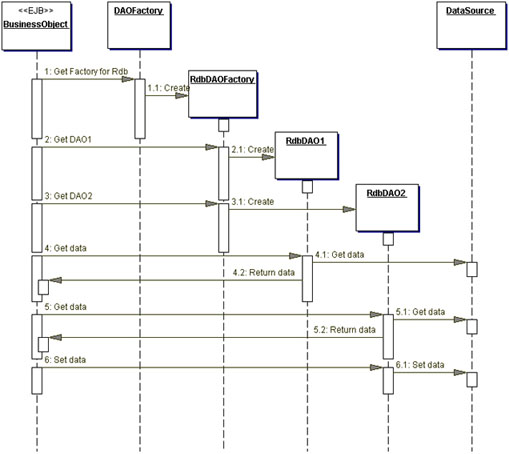
When the underlying storage is subject to change from one implementation to another, this strategy may be implemented using the Abstract Factory pattern. The Abstract Factory can in turn build on and use the Factory Method implementation, as suggested in Design Patterns: Elements of Reusable Object-Oriented Software [GoF]. In this case, this strategy provides an abstract DAO factory object (Abstract Factory) that can construct various types of concrete DAO factories, each factory supporting a different type of persistent storage implementation. Once you obtain the concrete DAO factory for a specific implementation, you use it to produce DAOs supported and implemented in that implementation.

The class diagram for this strategy is shown in Figure 9.4. This class diagram shows a base DAO factory, which is an abstract class that is inherited and implemented by different concrete DAO factories to support storage implementation-specific access. The client can obtain a concrete DAO factory implementation such as RdbDAOFactory and use it to obtain concrete DAOs that work with that specific storage implementation. For example, the data client can obtain an RdbDAOFactory and use it to get specific DAOs such as RdbCustomerDAO, RdbAccountDAO, and so forth. The DAOs can extend and implement a generic base class (shown as DAO1 and DAO2) that specifically describe the DAO requirements for the business object it supports. Each concrete DAO is responsible for connecting to the data source and obtaining and manipulating data for the business object it supports.

The sample implementation for the DAO pattern and its strategies is shown in the "Sample Code" section of this chapter.

   
**Figure 9.4 Factory for Data Access Object strategy using Abstract Factory**

The sequence diagram describing the interactions for this strategy is shown in Figure 9.5.

   
**Figure 9.5 Factory for Data Access Objects using Abstract Factory sequence diagram**

## Consequences

**Enables Transparency**  
Business objects can use the data source without knowing the specific details of the data source's implementation. Access is transparent because the implementation details are hidden inside the DAO.

**Enables Easier Migration**  
A layer of DAOs makes it easier for an application to migrate to a different database implementation. The business objects have no knowledge of the underlying data implementation. Thus, the migration involves changes only to the DAO layer. Further, if employing a factory strategy, it is possible to provide a concrete factory implementation for each underlying storage implementation. In this case, migrating to a different storage implementation means providing a new factory implementation to the application.

**Reduces Code Complexity in Business Objects**

Because the DAOs manage all the data access complexities, it simplifies the code in the business objects and other data clients that use the DAOs. All implementation-related code (such as SQL statements) is contained in the DAO and not in the business object. This improves code readability and development productivity.

**Centralizes All Data Access into a Separate Layer**  
Because all data access operations are now delegated to the DAOs, the separate data access layer can be viewed as the layer that can isolate the rest of the application from the data access implementation. This centralization makes the application easier to maintain and manage.

**Not Useful for Container-Managed Persistence**  
Because the EJB container manages entity beans with container-managed persistence (CMP), the container automatically services all persistent storage access. Applications using container-managed entity beans do not need a DAO layer, since the application server transparently provides this functionality. However, DAOs are still useful when a combination of CMP (for entity beans) and BMP (for session beans, servlets) is required.

**Adds Extra Layer**  
The DAOs create an additional layer of objects between the data client and the data source that need to be designed and implemented to leverage the benefits of this pattern. But the benefit realized by choosing this approach pays off for the additional effort.

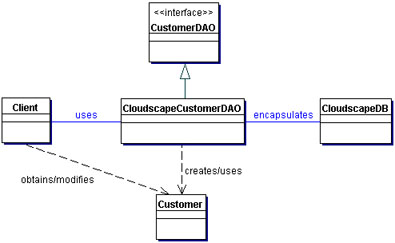
**Needs Class Hierarchy Design**  
When using a factory strategy, the hierarchy of concrete factories and the hierarchy of concrete products produced by the factories need to be designed and implemented. This additional effort needs to be considered if there is sufficient justification warranting such flexibility. This increases the complexity of the design. However, you can choose to implement the factory strategy starting with the Factory Method pattern first, and then move towards the Abstract Factory if necessary.

## Sample Code

### Implementing Data Access Object pattern

An example DAO code for a persistent object that represents Customer information is shown in Example 9.4. The CloudscapeCustomerDAO creates a Customer Transfer Object when thefindCustomer() method is invoked.

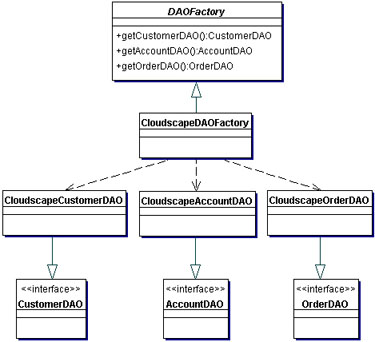
The sample code to use the DAO is shown in Example 9.6. The class diagram for this example is shown in Figure 9.6.

   
**Figure 9.6 Implementing the DAO pattern**

### Implementing Factory for Data Access Objects Strategy

Using Factory Method Pattern

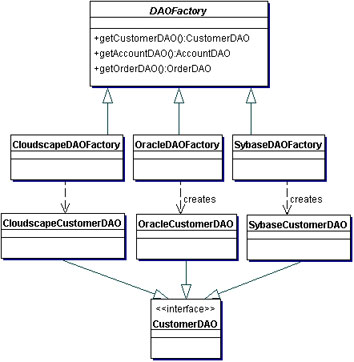
Consider an example where we are implementing this strategy in which a DAO factory produces many DAOs for a single database implementation (e.g., Oracle). The factory produces DAOs such as CustomerDAO, AccountDAO, OrderDAO, and so forth. The class diagram for this example is shown in Figure 9.7.

   
**Figure 9.7 Implementing the Factory for DAO strategy using Factory Method**

The example code for the DAO factory (CloudscapeDAOFactory) is listed in Example 9.2.

#### Using Abstract Factory Pattern

Consider an example where we are considering implementing this strategy for three different databases. In this case, the Abstract Factory pattern can be employed. The class diagram for this example is shown in Figure 9.8. The sample code in Example 9.1 shows code excerpt for the abstract DAOFactory class. This factory produces DAOs such as CustomerDAO, AccountDAO, OrderDAO, and so forth. This strategy uses the Factory Method implementation in the factories produced by the Abstract Factory.

   
**Figure 9.8 Implementing the Factory for DAO strategy using Abstract Factory** **Example 9.1 Abstract DAOFactory Class**

// Abstract class DAO Factory

public abstract class DAOFactory {

// List of DAO types supported by the factory

public static final int CLOUDSCAPE = 1;

public static final int ORACLE = 2;

public static final int SYBASE = 3;

...

// There will be a method for each DAO that can be

// created. The concrete factories will have to

// implement these methods.

public abstract CustomerDAO getCustomerDAO();

public abstract AccountDAO getAccountDAO();

public abstract OrderDAO getOrderDAO();

...

public static DAOFactory getDAOFactory(

int whichFactory) {

switch (whichFactory) {

case CLOUDSCAPE:

return new CloudscapeDAOFactory();

case ORACLE :

return new OracleDAOFactory();

case SYBASE :

return new SybaseDAOFactory();

...

default :

return null;

}

}

}

The sample code for CloudscapeDAOFactory is shown in Example 9.2. The implementation for OracleDAOFactory and SybaseDAOFactory are similar except for specifics of each implementation, such as JDBC driver, database URL, and differences in SQL syntax, if any.

**Example 9.2 Concrete DAOFactory Implementation for Cloudscape**

// Cloudscape concrete DAO Factory implementation

import java.sql.\*;

public class CloudscapeDAOFactory extends DAOFactory {

public static final String DRIVER=

"COM.cloudscape.core.RmiJdbcDriver";

public static final String DBURL=

"jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";

// method to create Cloudscape connections

public static Connection createConnection() {

// Use DRIVER and DBURL to create a connection

// Recommend connection pool implementation/usage

}

public CustomerDAO getCustomerDAO() {

// CloudscapeCustomerDAO implements CustomerDAO

return new CloudscapeCustomerDAO();

}

public AccountDAO getAccountDAO() {

// CloudscapeAccountDAO implements AccountDAO

return new CloudscapeAccountDAO();

}

public OrderDAO getOrderDAO() {

// CloudscapeOrderDAO implements OrderDAO

return new CloudscapeOrderDAO();

}

...

}

The CustomerDAO interface shown in Example 9.3 defines the DAO methods for Customer persistent object that are implemented by all concrete DAO implementations, such as CloudscapeCustomerDAO, OracleCustomerDAO, and SybaseCustomerDAO. Similar, but not listed here, are AccountDAO and OrderDAO interfaces that define the DAO methods for Account and Order business objects respectively.

**Example 9.3 Base DAO Interface for Customer**

// Interface that all CustomerDAOs must support

public interface CustomerDAO {

public int insertCustomer(...);

public boolean deleteCustomer(...);

public Customer findCustomer(...);

public boolean updateCustomer(...);

public RowSet selectCustomersRS(...);

public Collection selectCustomersTO(...);

...

}

The CloudscapeCustomerDAO implements the CustomerDAO as shown in Example 9.4. The implementation of other DAOs, such as CloudscapeAccountDAO, CloudscapeOrderDAO, OracleCustomerDAO, OracleAccountDAO, and so forth, are similar.

**Example 9.4 Cloudscape DAO Implementation for Customer**

// CloudscapeCustomerDAO implementation of the

// CustomerDAO interface. This class can contain all

// Cloudscape specific code and SQL statements.

// The client is thus shielded from knowing

// these implementation details.

import java.sql.\*;

public class CloudscapeCustomerDAO implements

CustomerDAO {

public CloudscapeCustomerDAO() {

// initialization

}

// The following methods can use

// CloudscapeDAOFactory.createConnection()

// to get a connection as required

public int insertCustomer(...) {

// Implement insert customer here.

// Return newly created customer number

// or a -1 on error

}

public boolean deleteCustomer(...) {

// Implement delete customer here

// Return true on success, false on failure

}

public Customer findCustomer(...) {

// Implement find a customer here using supplied

// argument values as search criteria

// Return a Transfer Object if found,

// return null on error or if not found

}

public boolean updateCustomer(...) {

// implement update record here using data

// from the customerData Transfer Object

// Return true on success, false on failure or

// error

}

public RowSet selectCustomersRS(...) {

// implement search customers here using the

// supplied criteria.

// Return a RowSet.

}

public Collection selectCustomersTO(...) {

// implement search customers here using the

// supplied criteria.

// Alternatively, implement to return a Collection

// of Transfer Objects.

}

...

}

The Customer Transfer Object class is shown in Example 9.5. This is used by the DAOs to send and receive data from the clients. The usage of Transfer Objects is discussed in detail in the Transfer Object pattern.

**Example 9.5 Customer Transfer Object**

public class Customer implements java.io.Serializable {

// member variables

int CustomerNumber;

String name;

String streetAddress;

String city;

...

// getter and setter methods...

...

}

Example 9.6 shows the usage of the DAO factory and the DAO. If the implementation changes from Cloudscape to another product, the only required change is the getDAOFactory() method call to the DAO factory to obtain a different factory.

**Example 9.6 Using a DAO and DAO Factory - Client Code**

...

// create the required DAO Factory

DAOFactory cloudscapeFactory =

DAOFactory.getDAOFactory(DAOFactory.DAOCLOUDSCAPE);

// Create a DAO

CustomerDAO custDAO =

cloudscapeFactory.getCustomerDAO();

// create a new customer

int newCustNo = custDAO.insertCustomer(...);

// Find a customer object. Get the Transfer Object.

Customer cust = custDAO.findCustomer(...);

// modify the values in the Transfer Object.

cust.setAddress(...);

cust.setEmail(...);

// update the customer object using the DAO

custDAO.updateCustomer(cust);

// delete a customer object

custDAO.deleteCustomer(...);

// select all customers in the same city

Customer criteria=new Customer();

criteria.setCity("New York");

Collection customersList =

custDAO.selectCustomersTO(criteria);

// returns customersList - collection of Customer

// Transfer Objects. iterate through this collection to

// get values.

...

## Related Patterns

Transfer Object

A DAO uses Transfer Objects to transport data to and from its clients.

Factory Method [GoF] and Abstract Factory [GoF]

The Factory for Data Access Objects Strategy uses the Factory Method pattern to implement the concrete factories and its products (DAOs). For added flexibility, the Abstract Factory pattern may be employed as discussed in the strategies.

Broker [POSA1]

The DAO pattern is related to the Broker pattern, which describes approaches for decoupling clients and servers in distributed systems. The DAO pattern more specifically applies this pattern to decouple the resource tier from clients in another tier, such as the business or presentation tier

# CHAPTER 3 Java DB(Apache Derby)의 이해

이 장에서는

- 이하 편집 후 전체 개요 요약 기술 -

<http://db.apache.org/derby/papers/DerbyTut/install_software.html>

<http://www.ibm.com/developerworks/kr/library/os-ad-trifecta9/index.html>

http://www.ibm.com/developerworks/kr/library/os-ad-trifecta1/index.html

<http://www.ibm.com/developerworks/kr/library/os-ad-trifecta9/index.html>

http://blog.naver.com/PostView.nhn?blogId=syban75&logNo=150030570555

<http://www.test104.com/kr/tech/2948.html>

<http://www.zdnet.co.kr/news/news_view.asp?artice_id=00000039166085&type=det>

<http://www.ibm.com/developerworks/views/opensource/libraryview.jsp?search_by=Developing+Apache+Derby+trifecta>

어드레스북 예제 링크

C:\Program Files\Java\jdk1.7.0\db\lib>set CLASSPATH=C:\Program Files\Java\jdk1.7

.0\db\lib\derbytools.jar

C:\Program Files\Java\jdk1.7.0\db\lib>java org.apache.derby.tools.sysinfo

------------------ Java 정보 ------------------

Java 버전: 1.7.0

Java 벤더: Oracle Corporation

Java 홈: C:\Program Files\Java\jre7

Java 클래스 경로: C:\Program Files\Java\jdk1.7.0\db\lib\derbytools.jar

OS 이름: Windows 7

OS 구조: x86

OS 버전: 6.1

Java 사용자 이름: nhn

Java 사용자 홈: C:\Users\nhn

Java 사용자 디렉토리: C:\Program Files\Java\jdk1.7.0\db\lib

java.specification.name: Java Platform API Specification

java.specification.version: 1.7

java.runtime.version: 1.7.0-b147

--------- Derby 정보 --------

JRE - JDBC: Java SE 6 - JDBC 4.0

[C:\Program Files\Java\jdk1.7.0\db\lib\derbytools.jar] 10.8.1.2 - (1095077)

------------------------------------------------------

----------------- 로케일 정보 -----------------

현재 로케일 : [한국어/대한민국 [ko\_KR]]

[cs] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[de\_DE] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[es] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[fr] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[hu] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[it] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[ja\_JP] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[ko\_KR] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[pl] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[pt\_BR] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[ru] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[zh\_CN] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

[zh\_TW] 로케일 지원을 찾았습니다.

버전: 10.8.1.2 - (1095077)

------------------------------------------------------

C:\Program Files\Java\jdk1.7.0\db\lib>java org.apache.derby.tools.ij

ij 버전 10.8

ij>

# CHAPTER 4 Java DB를 이용한 데스크톱 애플리케이션(Using Java DB in Desktop Applications)

<http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javadb/>

Sun Microsystems recently announced that it is distributing and supporting Java DB based on the 100 percent Java technology, open-source Apache Derby database. Derby was previously available under its earlier name, Cloudscape, from its former owners: Cloudscape, Informix, and IBM. IBM donated the Derby product source code to the Apache Foundation as an open-source project. Sun, IBM, other companies, and individuals have been actively involved in development of the relational database as part of the Apache Derby community. Sun distributes Java DB in many of its products, including the Sun Java Enterprise System and the Sun Java System Application Server. The NetBeans integrated development environment (IDE) 5.0 also supports Java DB.

Java DB is lightweight at 2 megabytes and embeddable within desktop Java technology applications. Desktop applications can now access powerful database storage with triggers, stored procedures, and support for SQL, Java DataBase Connectivity (JDBC) software, and Java Platform, Enterprise Edition (Java EE, formerly referred to as J2EE), all embedded within the same Java virtual machine (JVM).\*

This article describes how to download, install, integrate, and deploy Java DB within desktop Java technology applications. A demo application called Address Book demonstrates how to work with Java DB as an embedded database.

Contents

* - Creating the Address Book Demo
* - Installing Java DB
* - Integrating Java DB With NetBeans IDE 5.0
* - Loading the Database Driver
* - Connecting to the Java DB Database
* - Creating the Database
* - Using the Database
* - Deploying Your Application
* - Summary

Creating the Address Book Demo

The Address Book demo uses Java DB to store address information. This demo stores names, phone numbers, email addresses, and postal addresses. It allows you to create new address entries and to save, edit, and delete them. The application creates its database in the user's home directory within an .addressbook subdirectory. The database is embedded with the application, so there is no need to set up or manage a separate server or system. To deploy this embedded database application, we need only the application JAR file and the database library JAR file. Figure 1 shows the demo's user interface (UI).

|  |  |
| --- | --- |
| |  | | --- | | Figure 1: Address Book uses Java DB as an embedded database. | |

Figure 1: Address Book uses Java DB as an embedded database.

Address Book's main frame window is an AddressFrame class that extends a Java Foundation Classes/Swing (JFC/Swing) JFrame. The AddressFrame is a container for other graphical components and also acts as a controller by handling various events generated by the child components. The child components are JPanel subclasses, each with a different responsibility:

* AddressPanel represents an address record. It also provides the UI for editing existing records and creating new records. It contains text fields for all the major properties of an Address object.
* AddressActionPanel provides buttons for all the major use cases that the application supports. This panel generates events that AddressFrame must handle. For example, when the user clicks Save, this panel generates an event. AddressFrame listens to and handles all important events from this panel.
* AddressListPanel provides a scrollable list of names on the far left of the AddressFrame. The list holds ListEntry objects. A ListEntry stores a database record's unique identifier. The record identifier (ID) allows the application to retrieve an entire record's contents into the AddressPanel.

The application uses a Data Access Object (DAO) to isolate the database-specific code. The DAO encapsulates database connections and statements. A DAO is a useful design pattern that allows loose coupling between an application and the underlying persistence-storage mechanism. The application'sAddressDao class is an example of a DAO. When the AddressFrame edits, saves, or deletes an Address object, it always uses an instance of the AddressDao class. Although the Address Book application uses Java DB, you could change it to use an entirely different database just by modifying this one class.

Installing Java DB

The easiest way to get Java DB is to download a copy from the Sun Developer Network's Java DB site. The binary distribution provides everything you need to begin working with embedded database applications. After you download the binary distribution, you'll find a Java DB directory structure that contains the following subdirectories:

The demo subdirectory has two demonstration programs. One example shows how to create a trivial embedded application. The other shows how to use Java DB in a client-server environment.

The frameworks subdirectory contains utilities for setting up environment variables and for creating and working with databases. This directory is not useful for our demo because our application will be entirely self-contained. No outside utilities will be used for the Address Book application.

The javadoc subdirectory contains API documentation. This directory is particularly useful if you configure your IDE to point to it as the Java DB API Javadoc.

The docs subdirectory contains documentation about the Java DB product itself: setup, administrator, and reference guides.

Finally, the lib subdirectory contains the Java DB libraries packaged as JAR files. Read the Java DB documentation to find out about the various libraries. For an embedded database application, we will use only the derby.jar library file.

Installing Java DB for development requires only that you make the derby.jar file part of your application classpath. It's that simple. You can set the CLASSPATH variable of your Solaris, Linux, Windows, or other host environment to include the JAR file, or you can include the file as part of your command-line options when compiling or running. The Address Book demo's ANT script shows you how to include this JAR file during project development if you use ANT. Alternatively, some IDEs, including NetBeans IDE 5.0, let you configure classpaths.

Integrating Java DB With NetBeans IDE 5.0

Most IDEs provide a way to add libraries to the development classpath. Follow these instructions to add the Java DB libraries to NetBeans IDE 5.0:

From the Tools menu, select Library Manager, as shown in Figure 2.

Figure 2: The library manager lets you add third-party libraries to your project.

In the Library Manager window, create a new library named JavaDBEmbedded, as shown in Figure 3. Click OK.

Figure 3: Name the set of libraries that your project will need.

To add the derby.jar file to the JavaDBEmbedded library, click on Add JAR/Folder... in the Library Manager window. Navigate the file chooser to the derby.jar file and select it as shown in Figure 4.

Figure 4: Add the derby.jar file to the JavaDBEmbedded library.

In the same Library Manager window for the JavaDBEmbedded library, select the Javadoc tab. Add the javadoc subdirectory from your Java DB installation. Now, the Java DB API Javadoc is available when you use the JavaDBEmbedded library in your NetBeans IDE projects.

You can now add the JavaDBEmbedded library to your NetBeans IDE 5.0 project by using the project's property settings. When you compile, debug, and run the application within the IDE, the IDE will be able to find the needed derby.jar file.

To accommodate users of other IDEs, I have placed the derby.jar library within the lib subdirectory of the downloadable Address Book project. The accompanying ANT script can build and run the demo without any dependencies on the NetBeans IDE.

Loading the Database Driver

Loading the JDBC technology driver starts the database management system. Java DB's drivers come with the derby.jar file, so you don't need to download anything else. Load the JDBC driver by referencing it using the Class.forName method. The embedded driver name is org.apache.derby.jdbc.EmbeddedDriver, and you can load it as you would other JDBC drivers.

Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

The Address Book demo reads the driver name from a configuration property file and passes the name to a loadDriver method. Additionally, as mentioned earlier, Address Book encapsulates all database functionality into a Data Access Object (DAO), a core Java EE design pattern used to access data from a variety of sources. The DAO pattern works equally well for Java SE applications like Address Book. The following code snippet shows how AddressDao reads the driver name and loads the driver:

private Properties bProperties = null;

public AddressDao(String addressBookName) {

this.dbName = addressBookName;

setDBSystemDir();

dbProperties = loadDBProperties();

String driverName = dbProperties.getProperty("derby.driver");

loadDatabaseDriver(driverName);

...

}

private Properties loadDBProperties() {

InputStream dbPropInputStream = null;

dbPropInputStream =

AddressDao.class.getResourceAsStream("Configuration.properties");

dbProperties = new Properties();

try {

dbProperties.load(dbPropInputStream);

} catch (IOException ex) {

ex.printStackTrace();

}

return dbProperties;

}

private void loadDatabaseDriver(String driverName) {

// Load the Java DB driver.

try {

Class.forName(driverName);

} catch (ClassNotFoundException ex) {

ex.printStackTrace();

}

}

Connecting to the Java DB Database

A JDBC technology connection identifies a specific database and allows you to perform administrative tasks. Tasks include starting, stopping, copying, and even deleting a database. The driver manager provides all database connections.

Retrieve a connection from the driver manager by providing a URL string that identifies the database and a set of properties that influence the connection's interaction with the database. A very common use of properties is to associate a user name and password with a connection.

All connection URLs have the following form:

jdbc:derby:<dbName>[propertyList]

The dbName portion of the URL identifies a specific database. A database can be in one of many locations: in the current working directory, on the classpath, in a JAR file, in a specific Java DB database home directory, or in an absolute location on your file system. The easiest way to manage your database location in an embedded environment is to set the derby.system.home system property. This property tells Java DB the default home location of all databases. By setting this property, the Address Book demo ensures that Java DB always finds the correct application database. The application database is named DefaultAddressBook, and it will exist within the directory indicated by the derby.system.home property. The connection URL for this database would look like this:

jdbc:derby:DefaultAddressBook

The optional propertyList is a set of properties that you can pass to the database system. You can pass properties into the Java DB system either on the URL itself or as a separate Properties object. If properties are part of the URL, a semicolon (;) should precede each property. The most common properties are these:

* create=true
* databaseName=nameOfDatabase
* user=userName
* password=userPassword
* shutdown=true

To connect to the DefaultAddressBook database, the demo must first set the derby.system.home system property. The demo uses the .addressbook subdirectory of the user's home directory. Use the System class to find out the user's home directory. Then use the class again to set the derby.system.home property:

private void setDBSystemDir() {

// Decide on the db system directory: <userhome>/.addressbook/

String userHomeDir = System.getProperty("user.home", ".");

String systemDir = userHomeDir + "/.addressbook";

// Set the db system directory.

System.setProperty("derby.system.home", systemDir);

}

Once the application has specified where all databases will exist, it can retrieve a database connection. In this example, notice that I have appended connection properties to the database URL.

Connection dbConnection = null;

String strUrl = "jdbc:derby:DefaultAddressBook;user=dbuser;password=dbuserpwd";

try {

dbConnection = DriverManager.getConnection(strUrl);

} catch (SQLException sqle) {

sqle.printStackTrace();

}

Alternatively, you can put those properties into a Properties object. Pass the Properties object as an argument when retrieving a connection:

Connection dbConnection = null;

String strUrl = "jdbc:derby:DefaultAddressBook";

Properties props = new Properties();

props.put("user", "dbuser");

props.put("password", "dbuserpwd");

try {

dbConnection = DriverManager.getConnection(strUrl, props);

} catch(SQLException sqle) {

sqle.printStackTrace();

}

Creating the Database

The Address Book demo application does not have a ready-made database. Instead, the application must create the database when it starts. One of the biggest advantages of using an embedded database is that the application -- not the user -- takes care of the details of setting up a database. The application can control where the database exists, what tables exist, and how permissions are handled.

Address Book creates a database called DefaultAddressBook in a subdirectory of the user's home directory, and it does so without asking the user for any additional information. You can create a new database in Java DB by using the create=true property when retrieving a connection to a database. Because our application will use the DefaultAddressBook database, we should first create this database. Assuming that the application has already set the derby.system.home property as discussed earlier, the application creates the database and connects to it like this:

Connection dbConnection = null;

String strUrl = "jdbc:derby:DefaultAddressBook;create=true";

try {

dbConnection = DriverManager.getConnection(strUrl);

} catch (SQLException ex) {

ex.printStackTrace();

}

Because the create=true property is included, Java DB will attempt to create the database files for the first time. Creating the database doesn't actually create any application tables. However, you should now be able to find a new subdirectory named .addressbook/DefaultAddressBook in your home directory.

After it has created the database, the application creates the tables. The demo uses only one ADDRESS table in the default application APP schema. The following SQL code creates the ADDRESS table:

CREATE table APP.ADDRESS (

ID INTEGER NOT NULL

PRIMARY KEY GENERATED ALWAYS AS IDENTITY

(START WITH 1, INCREMENT BY 1),

LASTNAME VARCHAR(30),

FIRSTNAME VARCHAR(30),

MIDDLENAME VARCHAR(30),

PHONE VARCHAR(20),

EMAIL VARCHAR(30),

ADDRESS1 VARCHAR(30),

ADDRESS2 VARCHAR(30),

CITY VARCHAR(30),

STATE VARCHAR(30),

POSTALCODE VARCHAR(20),

COUNTRY VARCHAR(30) )

Each record has a record identifier or ID field. Java DB generates this value for each new record that it adds to the database. The ID field is the primary key for each address record.

All remaining address record fields contain varchar elements of various lengths. For example, the LASTNAME field can contain a maximum of 30 varchar characters. The varchar type is equivalent to a UTF-16 Java char code unit.

The Java technology code that uses the above SQL statement to create the ADDRESS table looks like the following code. The dbConnection is the same as the one shown in the previous code. We simply pass it into createTables, create a new Statement, and call the execute method to run the SQL code on the newly formed database. The strCreateAddressTable instance variable holds the SQL statement text.

private boolean createTables(Connection dbConnection) {

boolean bCreatedTables = false;

Statement statement = null;

try {

statement = dbConnection.createStatement();

statement.execute(strCreateAddressTable);

bCreatedTables = true;

} catch (SQLException ex) {

ex.printStackTrace();

}

return bCreatedTables;

}

At this point, the database and the ADDRESS table exist in a subdirectory named .addressbook/DefaultAddressBook in your home directory. Although you can browse this subdirectory, avoid modifying any files. If you edit or delete any of these database files directly, you can destroy the integrity of your database.

Using the Database

Once the database and its tables have been created, your application can create new connections and statements to add, edit, delete, or retrieve records. In Address Book, these actions are controlled by buttons within the AddressActionPanel. Figure 5 shows the available options:

* New. Create a new address record.
* Delete. Delete the displayed address record.
* Edit. Edit the displayed address record.
* Save. Save the new or edited address record that is displayed.
* Cancel. Cancel any edits or any attempt to create a new record.

Figure 5: Address Book has several options for interacting with records.

The main window of the application is AddressFrame, which acts as a controller and as a view at the same time. It registers itself with the AddressActionPanel to receive notification when a user clicks anywhere on the action bar.

The New command clears the address entry panel and enables the user to edit all fields. No SQL commands are issued at this point, but the UI should allow you to enter a new address.

The Delete command attempts to delete the currently selected address record. AddressFrame retrieves the currently selected Address identifier from the AddressPanel and uses AddressDao to delete the record. The panel calls its own deleteAddress method, which calls the DAO's deleteRecord method with the correct ID. After deleting the record from the database, the application must delete the ListEntry from the AddressListPanel too.

private void deleteAddress() {

int id = addressPanel.getId();

if (id != -1) {

db.deleteRecord(id);

int selectedIndex = addressListPanel.deleteSelectedEntry();

...

}

...

}

In the AddressDao, the deleteRecord method handles the actual deletion of the record from the database. The AddressDao creates a PreparedStatement when it first connects to the database.

stmtDeleteAddress = dbConnection.prepareStatement(

"DELETE FROM APP.ADDRESS " +

"WHERE ID = ?");

The PreparedStatement can be used multiple times, and this one uses a parameter to determine which record to delete. The deleteRecord method executes the update after setting the ID parameter:

public boolean deleteRecord(int id) {

boolean bDeleted = false;

try {

stmtDeleteAddress.clearParameters();

stmtDeleteAddress.setInt(1, id);

stmtDeleteAddress.executeUpdate();

bDeleted = true;

} catch (SQLException sqle) {

sqle.printStackTrace();

}

return bDeleted;

}

The Edit command allows the user to edit the currently selected Address record in the AddressPanel. For example, you can change the name, city, or phone number of a saved record.

The Save command retrieves either the newly created or edited Address from the AddressPanel and attempts to either update the existing record or create a new record. If the user has been editing a record, Save will update that record with the new information. If the user has created a new record, Save will insert a new record in the database. New records have not yet been saved. At this point, their ID field is still set to the default -1 value. This value changes to an autogenerated, unique record identifier once you save the record.

The following code in AddressFrame will save edited and newly created address records by calling the DAO's editRecord or saveRecord method, respectively. Of course, when you create a new record, the application must also update the AddressListPanel.

private void saveAddress() {

if (addressPanel.isEditable()) {

Address address = addressPanel.getAddress();

int id = address.getId();

if (id == -1) {

id = db.saveRecord(address);

address.setId(id);

String lname = address.getLastName();

String fname = address.getFirstName();

String mname = address.getMiddleName();

ListEntry entry = new ListEntry(lname, fname, mname, id);

addressListPanel.addListEntry(entry);

} else {

db.editRecord(address);

}

addressPanel.setEditable(false);

}

}

The DAO's editRecord method must update the fields that change in the Address record. Because the demo application doesn't distinguish between edited and unedited fields, it simply updates all fields in the record. Following are the PreparedStatement object and the editRecord method:

stmtUpdateExistingRecord = dbConnection.prepareStatement(

"UPDATE APP.ADDRESS " +

"SET LASTNAME = ?, " +

" FIRSTNAME = ?, " +

" MIDDLENAME = ?, " +

" PHONE = ?, " +

" EMAIL = ?, " +

" ADDRESS1 = ?, " +

" ADDRESS2 = ?, " +

" CITY = ?, " +

" STATE = ?, " +

" POSTALCODE = ?, " +

" COUNTRY = ? " +

"WHERE ID = ?");

...

public boolean editRecord(Address record) {

boolean bEdited = false;

try {

stmtUpdateExistingRecord.clearParameters();

stmtUpdateExistingRecord.setString(1, record.getLastName());

stmtUpdateExistingRecord.setString(2, record.getFirstName());

stmtUpdateExistingRecord.setString(3, record.getMiddleName());

stmtUpdateExistingRecord.setString(4, record.getPhone());

stmtUpdateExistingRecord.setString(5, record.getEmail());

stmtUpdateExistingRecord.setString(6, record.getAddress1());

stmtUpdateExistingRecord.setString(7, record.getAddress2());

stmtUpdateExistingRecord.setString(8, record.getCity());

stmtUpdateExistingRecord.setString(9, record.getState());

stmtUpdateExistingRecord.setString(10, record.getPostalCode());

stmtUpdateExistingRecord.setString(11, record.getCountry());

stmtUpdateExistingRecord.setInt(12, record.getId());

stmtUpdateExistingRecord.executeUpdate();

bEdited = true;

} catch(SQLException sqle) {

sqle.printStackTrace();

}

return bEdited;

}

Saving a new Address creates a new database record, and that new record has a new primary key or record identifier. When we create the PreparedStatement, we can tell the database that we want to know the generated keys. Notice how we can retrieve a ResultSet of generated keys after inserting the record. The saveRecord method returns the primary key of the newly created record.

stmtSaveNewRecord = dbConnection.prepareStatement(

"INSERT INTO APP.ADDRESS " +

" (LASTNAME, FIRSTNAME, MIDDLENAME, " +

" PHONE, EMAIL, ADDRESS1, ADDRESS2, " +

" CITY, STATE, POSTALCODE, COUNTRY) " +

"VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)",

Statement.RETURN\_GENERATED\_KEYS);

...

public int saveRecord(Address record) {

int id = -1;

try {

stmtSaveNewRecord.clearParameters();

stmtSaveNewRecord.setString(1, record.getLastName());

stmtSaveNewRecord.setString(2, record.getFirstName());

stmtSaveNewRecord.setString(3, record.getMiddleName());

stmtSaveNewRecord.setString(4, record.getPhone());

stmtSaveNewRecord.setString(5, record.getEmail());

stmtSaveNewRecord.setString(6, record.getAddress1());

stmtSaveNewRecord.setString(7, record.getAddress2());

stmtSaveNewRecord.setString(8, record.getCity());

stmtSaveNewRecord.setString(9, record.getState());

stmtSaveNewRecord.setString(10, record.getPostalCode());

stmtSaveNewRecord.setString(11, record.getCountry());

int rowCount = stmtSaveNewRecord.executeUpdate();

ResultSet results = stmtSaveNewRecord.getGeneratedKeys();

if (results.next()) {

id = results.getInt(1);

}

} catch(SQLException sqle) {

sqle.printStackTrace();

}

return id;

}

Deploying Your Application

Now that you have written the application, you must deploy it to users. Java technology applications can use a variety of deployment strategies, including Java Web Start software, applets, and stand-alone JAR files. I distribute the Address Book application as a stand-alone application with JAR files.

The ANT build file, build.xml, uses a dist target to create AddressBook.jar. It also places the database JAR file in the lib subdirectory directly under the AddressBook.jar location. The final distribution structure for the application looks like this:

AddressBook.jar

lib/derby.jar

In many cases, applications that use third-party libraries such as derby.jar require an external script to run. The script usually places the third-party JAR files on the classpath and then executes the application from its JAR. This method is troublesome, however, because it requires multiple scripts, typically one for each supported host operating system. For example, to support distribution on Windows, Solaris, and Linux platforms, I would create a run.bat batch file for the Windows platform and a run.csh script for a Solaris or Linux platform. Instead, we can avoid these clumsy execution scripts.

If our build process includes classpath information in the AddressBook.jar manifest file, you can run the application by simply passing the AddressBook.jar file on the execution command line. On most platforms, you can also just double-click on the JAR file name in a graphical window. On a command line, you can use this simple execution command:

java -jar AddressBook.jar

This simple deployment and execution scenario can be accomplished by creating a manifest.mf file that becomes part of the AddressBook.jar file. You can include information in the manifest that tells the Java programming language interpreter which class contains the main method and what other JAR files should become part of the classpath. The following manifest does both, and we can include it when building AddressBook.jar.

Manifest-Version: 1.0

Main-Class: com.sun.demo.addressbook.AddressFrame

Class-Path: lib/derby.jar

Once your build process generates the application distribution structure shown previously, you can simply distribute this structure as a ZIP file. Users can simply unzip the file into any location and run the AddressBook.jar file. The AddressBook.jar file will contain the manifest file mentioned earlier and will tell your runtime environment what JAR files should also be on the classpath. Of course, because Java DB is embedded into this application, it must be able to find lib/derby.jar in order to run correctly.

Summary

Working with Java DB is easy and fun. Java DB makes it possible to create and distribute an embedded database with minimal effort. Just remember a few tips to make your work with Java DB successful:

* + - * 1. Place the derby.jar file in your development environment's classpath so that your Java technology compiler and runtime environment can find the libraries to compile and run the application.
        2. Set the derby.system.home system property to tell Java DB where to find your database. You can set this property programmatically or on the command line.
        3. Create a build process that places the derby.jar file in a lib subdirectory immediately below your application's own directory.

Add derby.jar to the application classpath by including a Class-Path property in your application JAR's manifest file.

# CHAPTER 5 Fly by Night Service

# Introduction to Apache Derby

**요약：** 모든 애플리케이션 개발자들은 데이터를 저장하는 문제에 직면해 있습니다. 인터넷 또는 웹 기반 애플리케이션의 성장과 함께 이러한 필요성은 보다 명확해 졌습니다. 이번 칼럼에서는 표준 기반의 오픈 소스 자바 데이터베이스 시스템인 Apache Derby를 소개합니다. 이것을 다른 데이터베이스 시스템들과 비교하고 다운로드와 설치와 관련한 문제들을 논의합니다.

<http://www.ibm.com/developerworks/kr/library/os-ad-trifecta1/>

<http://www.ibm.com/developerworks/opensource/library/os-ad-trifecta1/#fig1>

## The Apache Derby project

Apache Derby 프로젝트에서 개발된 소프트웨어 기술을 연구하고자 기획된 새로운 시리즈 "Apache Derby로 개발하기 -- Hitting the Trifecta"의 첫 번째 글이다. Apache Derby 프로젝트에서 내놓은 소프트웨어는 IBM이 Apache Software Foundation에 기부한 기술을 기반으로 하는 오픈 소스 데이터베이스 시스템이다. Apache Derby 데이터베이스 소프트웨어는 자바로 작성되었기 때문에 이식성이 뛰어나고 작은 패키지에서 놀라운 퍼포먼스를 보인다.

Derby 데이터베이스는 많은 데이터베이스 표준들을 구현하기 때문에, 데이터베이스 사용 경험이 있을 경우 Derby를 사용해도 되고, 기존의 Derby 데이터베이스 애플리케이션을 다른 표준 순응의 데이터베이스 시스템으로 이식할 수도 있다. Derby는 공식적으로 릴리스 된지 일년도 안됐기 때문에 유용한 정보가 부족하다. IBM developerWorks에서 많은 기술자료와 튜토리얼을 제공하고 있다. 본 시리즈는 데이터베이스 시스템을 사용한 경험이 별로 없는 사용자들을 위해 기획되었다. developerWorks 웹 사이트의 다른 기술자료들에도 Apache Derby 데이터베이스 소프트웨어를 소개한 것이 많이 있다. 자바 엔터프라이즈 소프트웨어 스택으로 통합되는 방법을 설명한 자료도 있다.

이 글에서는 먼저 일반적인 데이터베이스 시스템에 대해 설명한 다음, Apache Derby를 설명하도록 하겠다.

## 데이터베이스 시스템

여러분이 인식하는지는 잘 모르겠지만, 인터텟 서핑을 하다 보면 다양한 데이터베이스 기반 웹 애플리케이션과 인터랙팅 하게 된다. 생소하게 들릴 수 있지만, 여러분이 방문하는 웹 사이트는 데이터베이스에 저장된 데이터를 사용하여 동적으로 생성되는 것이다. 다음과 같은 유형의 웹 사이트를 여러분이 방문했다고 생각해 보자.

* developerWorks 오픈 소스 프로젝트(그림 1) 같은 정보 포탈.
* 로컬 뉴스나 스포츠 기사를 다루는 뉴스 웹사이트.
* 은행 또는 투자 기관 같은 금융 웹 사이트.
* 도로 지도 웹사이트.
* 특정 주제에 대한 정보를 담고 있는 웹 사이트를 찾아주는 검색 엔진.



그림 - 인터넷 검색 포털 Naver

위 예제들 모두 데이터베이스를 사용하여 정보를 저장, 배치, 검색한다. 이러한 애플리케이션에서 웹 사이트는 사용자에게서 필요한 정보를 모으고(예를 들어, 주소), 애플리케이션 데이터베이스를 쿼리하고, 요청을 받은 데이터를 시각적인 결과로 만들어 낸다.

대부분의 데이터베이스 시스템들은 크고 복잡하다. 정확한 주행 방향을 그림과 함께 제공하기 위해 필요한 지도 정보를 갖고 있다고 생각해 보라. 데이터를 저장하고 이것을 애플리케이션에 사용하는 것은 큰 일임에 분명하다. IBM DB2 ®와 Microsoft SQL Server 등의 많은 벤더들이 개입되었다. 이러한 상용 데이터베이스 시스템들은 완전히 엔터프라이즈급의 기능을 제공한다. 결국 엄청난 데이터 양을 보유할 수 있고 많은 사용자들과 동시에 인터랙팅 할 수 있으며 여러 대형 전산 환경으로 나아갈 수 있다.

## 데이터베이스 역할

알고 있겠지만, 이러한 시스템을 작동시키는 것은 간단한 일이 아니다. 전통적으로, 데이터베이스를 작동할 때 개입되는 태스크들은 세 가지 범주로 나뉜다. 역할이 가끔은 겹치지만, 개별 책임들은 이해하기 쉽다.

* + - * 1. 데이터베이스 관리자 (DBA) -- 데이터베이스 시스템의 전체적인 운영을 책임지고 있다. 기반 하드웨어의 선택과 레이아웃, 데이터베이스 서버의 설치 및 최적화, 데이터베이스 서버의 일반적인 작동(데이터 백업과 복구) 등을 관리한다.
        2. 데이터베이스 개발자 -- 데이터베이스 작동에 대한 실질적인 책임을 맡고 있다. 데이터베이스 설계, 스키마, 테이블, 테이블 관계, 인덱스, 쿼리 최적화 등이 포함된다.
        3. 데이터베이스 애플리케이션 개발자 -- Java Database Connectivity (JDBC)나 Open Database Connectivity (ODBC) 같은 데이터베이스 애플리케이션 프로그래밍 인터페이스(API) 애플리케이션 코드와 기반 데이터베이스를 통합하여 필요할 때 마다 데이터를 저장 및 검색한다.

아직도 이해가 되지 않는다 해도 괜찮다. 원래 데이터베이스는 어려운 법이다. 온라인 뱅킹 예제를 들어보겠다. 은행 웹 사이트로 연결할 때 자신의 정보(사용자이름과 패스워드)를 주고 액세스 할 수 있다. 데이터를 보고, 대금을 지불하고, 자금을 이체할 수 있다. 여러분의 은행이 사용하는 데이터베이스는 관련 정보를 빠르게 배치하고, 트랜잭션을 안전하게 관리하고, 사용자와 보안 상태에서 인터랙팅 하고, 무엇보다도 중요한 것은 데이터를 소실하지 않아야 한다. 은행은 많은 사용자들을 대상으로 이러한 일을 동시에 처리해야 한다.

하지만 모든 애플리케이션들이 꼭 이래야 하는 것은 아니다. 데이터베이스 작동을 그저 배우려고 하거나 데이터베이스 애플리케이션을 빠르게 프로토타이핑 하려는 것이었다면 대부분의 상용 데이터베이스 시스템들이 오히려 거추장스럽다. 다행히도 Apache Derby 데이터베이스를 사용하여 데이터베이스 기반 애플리케이션을 개발하는 것은 생각 보다 쉽다. 앞으로의 시리즈에서는 Apache Derby 데이터베이스를 사용하여 데이터베이스 애플리케이션을 구현하는 방법을 설명하도록 하겠다.

## Apache Derby 프로젝트란 무엇인가?

Apache Derby 프로젝트는 완전히 자바 프로그래밍 언어로 작성된 오픈 소스 데이터베이스를 구현하는 프로젝트이다. 데이터베이스를 개발하는 일은 간단한 일이 아니고 Apache Derby 데이터베이스도 예외는 아니다. 하지만 Derby 프로젝트는 기초부터 시작하지 않았다. 1996년 Cloudscape, Inc.가 자바 언어로 작성된 데이터베이스 서버를 구현할 목표를 가지고 설립되었다. 이 회사는 그 다음 해에 첫 번째 릴리스를 내놓았고 결국 제품의 이름도 Cloudscape로 바뀌었다. 1999년에 Cloudscape, Inc.는 대형 데이터베이스 벤더인 Informix Software, Inc.에 인수되었다. Informix Software는 2001년 IBM이 인수했고, IBM Cloudscape™ 데이터베이스 시스템은 많은 IBM 제품들에서 임베디드 데이터베이스 엔진으로 사용되었다. 2004년 4월, IBM은 Cloudscape 데이터베이스 소프트웨어를 Apache Software Foundation에 기부했고 Apache Derby 프로젝트가 생겨났다.



그림 -2 Apache Derby 프로젝트 웹사이트

Cloudscape 데이터베이스는 거의 오십만 줄에 육박하는 자바 코드였고 이것을 Apache Derby 프로젝트로 변환하는데 시간이 걸렸다. 인큐베이션 기간이 지난 후에 Derby는 2005년 7월에 공식적으로 릴리스 되었다. 새로운 것 처럼 보이겠지만 Derby는 거의 10년이나 되는 개발 역사를 갖고 있다.

IBM은 계속해서 Cloudscape 데이터베이스를 관리하고 있다. IBM은 Cloudscape 데이터베이스를 무료 다운로드로 제공하고 있고 유료 컨설팅 서비스도 병행하고 있다. 게다가 Sun Microsystems는 Apache Derby의 패치 버전을 Java DB 제품으로서 포함시킬 것이라고 발표했다. IBM과 Sun의 강력한 지원으로 Apache Derby 데이터베이스의 미래는 밝다. Derby 데이터베이스는 SQL-92와 JDBC, Version 3.0 같은 많은 데이터베이스 표준에 순응한다. 따라서 Derby 데이터베이스 시스템을 사용하여 초기에 개발된 애플리케이션은 IBM DB2 Universal Database™ 같은 데이터베이스 시스템으로 쉽게 이식될 수 있다.

## Apache Derby 데이터베이스 개요

Apache Derby는 자바로 작성되었기 때문에 자바 가상 머신(JVM)이 있는 어느 곳에서나 실행될 수 있다. 다시 말해서 Microsoft Windows®, Macintosh, Linux®, UNIX® 플랫폼 같은 어떤 운영 체계에서도 실행될 수 있다는 의미가 된다. Derby는 세 개의 자바 플랫폼- Java 2 Platform, Micro Edition (J2ME), Java 2 Platform, Standard Edition (J2SE), Java 2 Platform, Enterprise Edition (J2EE)-에서도 실행된다. Derby 소프트웨어는 Java Archive (JAR) 파일에 번들된다. (크기는 2MB 정도이다.) Derby 데이터베이스는 풋프린트도 작고 애플리케이션과 함께 쉽게 번들 될 수 있다.

Derby 데이터베이스를 두 가지 방법으로 사용할 수 있다.

* 임베디드 데이터베이스(embedded database): 사용자는 데이터베이스의 존재를 인식하지 못한다. 애플리케이션은 데이터베이스를 사용한다. 같은 JVM에서 실행되고 데이터베이스는 로컬 파일 시스템상에 데이터를 저장한다. 임베디드 모델에서 데이터베이스는 같은 JVM에서 실행되는 애플리케이션만 통신한다.
* 클라이언트-서버 연결(client-server connection): 많은 상용 벤더들이 사용하는 전통적인 모델이다. 이 모델에서, 애플리케이션은 네트워크 연결을 통해 데이터베이스와 통신하고, 애플리케이션과 데이터베이스는 개별 JVM에서 작동한다. 데이터베이스 서버는 다중 클라이언트 애플리케이션들과 통신할 수 있다.

### Apache Derby 다운로드 하기

Apache Derby의 작동의 단순함을 제대로 이해하는 한 가지 방법은 직접 실행해 보는 것이다. 이 나머지 섹션에서는 Apache Derby의 버전을 다운로드 하여 확인하는 과정을 설명하도록 하겠다.(참고자료) 그 전에 Java Runtime Environment (JRE)가 설치되어 있어야 한다. 1.3 이후 버전의 JRE면 충분하지만 이 글에서는 Java 1.4.2 이후 버전을 사용한다.

가장 먼저, Apache Derby를 다운로드 한다. 그림 3에서 처럼, 세 개의 다른 버전들-소스, 라이브러리, 바이너리를 다운로드 한다. 소스 버전은 소스 코드를 말한다. 이 버전을 사용하려면 소스 코드를 컴파일 하여 .jar 파일을 구현해야 한다. 라이브러리 버전에는 Derby 데이터베이스용 필수 .jar 파일만 포함되어 있다. 바이너리 버전에는 .jar 파일과 Derby 문서가 포함되어 있다.

간단히 하기 위해, 바이너리 버전을 다운로드 한다. 다운로드의 무결성을 확인하라. 공식 버전임을 보장하는 PGP (Pretty Good Privacy) 서명을 확인하고 다운로드 파일의 무결성을 보증하는 MD5 (Message-Digest algorithm 5) 서명도 확인하라.

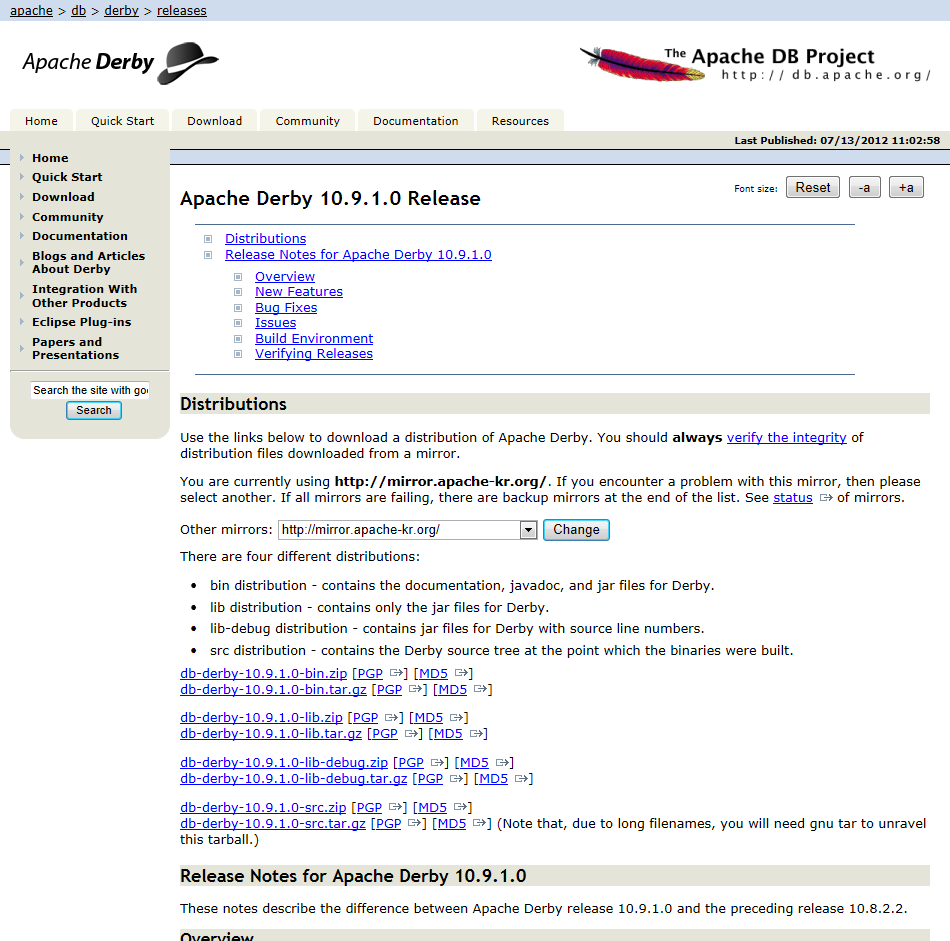


그림 -3 Apache Derby 데이터베이스 다운로드

### Apache Derby 설치하기

Derby 데이터베이스 파일들의 다운로드 및 무결성 검사가 끝났다면 설치는 간단하다.(물론 플랫폼에 따라 다르다.)

알맞은 위치를 선택한다. C:\Apache(Windows) 또는 /opt/Apache (UNIX 기반 시스템)

터미널 윈도우를 열고(또는 Windows의 경우 명령어 프롬프트), 새로운 디렉토리로 변경하고, Derby 데이터베이스를 포함하고 있는 아카이브를 확장한다. 이렇게 하면 여러분이 설치했던 Derby 데이터베이스 버전으로 새로운 디렉토리가 생성된다. (db-derby-10.1.2.1-bin)

Derby .jar 파일을 CLASSPATH 환경 변수에 추가한다. 명령행 프롬프트가 편하다면 db-derby-10.1.2.1-bin/lib 디렉토리에 있는 derby.jar를 CLASSPATH 변수에 직접 추가한다. 또 다른 방법으로는 db-derby-10.1.2.1-bin\frameworks\embedded\bin 디렉토리에 있는 setEmbeddedCP 스크립트를 실행할 수도 있다. 이 스크립트를 실행하기 전에 DERBY\_INSTALL 환경 변수를 설정하거나 이 스크립트를 변경하여 DERBY\_INSTALL이 해당 디렉토리를 가리키도록 한다.

sysinfo 툴로 설치 상황을 검사한다.(

sysinfo로 Derby 설치 테스트하기

C:\Program Files\Java\jdk1.7.0\db\lib>java -classpath .;derby.jar org.apach

by.tools.sysinfo

------------------ Java 정보 ------------------

Java 버전: 1.7.0\_05

Java 벤더: Oracle Corporation

Java 홈: C:\Program Files\Java\jre7

Java 클래스 경로: .;derby.jar

OS 이름: Windows 7

OS 구조: x86

OS 버전: 6.1

Java 사용자 이름: nhn

Java 사용자 홈: C:\Users\nhn

Java 사용자 디렉토리: C:\Program Files\Java\jdk1.7.0\db\lib

java.specification.name: Java Platform API Specification

java.specification.version: 1.7

java.runtime.version: 1.7.0\_05-b05

--------- Derby 정보 --------

JRE - JDBC: Java SE 6 - JDBC 4.0

[C:\Program Files\Java\jdk1.7.0\db\lib\derby.jar] 10.8.1.2 - (1095077)

------------------------------------------------------

----------------- 로케일 정보 -----------------

현재 로케일 : [한국어/대한민국 [ko\_KR]]

.

.

.

### Apache Derby 사용하기

데이터베이스 시스템 작동은 어려운 일만은 아니다. Apache Derby를 사용하면 완벽한 기능을 갖춘 데이터베이스 시스템을 실행할 수 있다. Apache Derby는 표준 순응형 데이터베이스이기 때문에 이것을 사용하여 개발된 애플리케이션들은 보다 강력한 데이터베이스 시스템으로 쉽게 마이그레이션 될 수 있다.

# Apache Derby를 이용한 자바 데이터베이스 개발

요약： 이 글은 Apache Derby 데이터베이스를 이용한 자바™ 애플리케이션 개발 방법을 설명합니다. Derby의 임베디드 Java Database Connectivity (JDBC) 드라이버를 사용하여 임베디드 Derby 데이터베이스에 연결하는 방법을 설명하며 데이터베이스 메타데이터와, Derby에서 생성된 SQL 에러와 경고를 자바 애플리케이션에서 해결하는 방법도 설명합니다.

<http://www.ibm.com/developerworks/kr/library/os-ad-trifecta9/index.html>

<http://www.ibm.com/developerworks/opensource/library/os-ad-trifecta2/>

## JDBC 소개

일전에 본 시리즈 기술자료에서 ij 툴을 사용하여 Apache Derby 데이터베이스에 연결하여 인터랙팅하는 예제를 통해 많은 데이터베이스 개념들을 설명한바 있다. 그 당시에는 분명하지는 않았지만 임베디드 Apache Derby 데이터베이스로 연결하여 인터랙팅 하기 위해 JDBC 애플리케이션 프로그래밍 인터페이스(API)를 사용했던 자바 애플리케이션을 사용하고 있었다. 앞으로의 기술자료에서는 고유의 자바 애플리케이션을 작성하여 ij 툴의 기본 기능을 재생성 하는 방법을 설명할 것이다. 이 글에서는 데이터베이스로 연결하여 데이터베이스 에러와 경고를 다루는 방법에 초점을 맞춰 설명하도록 하겠다.

자바 코드를 보기 전에, JDBC API의 특징을 설명하겠다. JDBC는 공식 Java Database Connectivity API이고, Java Development Kit 1.1 버전부터 있었다. JDBC API는 java.sql 패키지 내에 포함되어 있고, 면밀히 관찰한다면, 이 API는 분명히 인터페이스라는 것을 알 수 있을 것이다. 결국, 데이터베이스 JDBC 드라이버를 생성하는 작업은 그러한 인터페이스를 구현하는 자바 클래스를 제공해야 하는 데이터베이스 벤더(또는 엔터프라이즈 서드 파티)에게 할당된다. JDBC API의 확장으로 보다 고급스러운 기능이 제공되며, javax.sql 패키지에서 사용할 수 있다. 다음 글에서는 표준 JDBC 패키지들을 설명하도록 하겠다. 기본적인 것을 설명한 후에 확장 부분에 대해 설명하겠다.

자바 애플리케이션과 데이터베이스 간 연결은 JDBC 드라이버에 의해 제어된다. 원래, 네 가지 유형의 JDBC 드라이버가 있었고, 이들은 유형 넘버(Type Number) 1,2,3,4로 구분되었다. 이러한 유형들은 자바 애플리케이션이 데이터베이스와 통신하는 다양한 기술들을 나타내었다. 오늘날, Derby 데이터베이스로 연결하는데 사용하는 드라이버를 포함하여, 대부분의 드라이버는 Type 4 드라이버이고, 이는 완전히 자바 언어로 작성되었으며 JDBC API를 벤더 스팩의 데이터베이스 프로토콜로 직접 변환한다. Derby 데이터베이스의 경우, Derby가 자바로 작성되었기 때문에 프로세스가 비교적 단순하다.

### Apache Derby와 JDBC

JDBC의 기초를 익혔다면, 자바 프로그래밍 언어를 사용하여 임베디드 Apache Derby 데이터베이스로 연결하는 방법을 배워보자. 우선, Apache Derby 데이터베이스 소프트웨어를 설치해야 한다. (본 시리즈의 첫 번째 기술자료 참조). 아직 설치 전이라면, 첫 번째 기술자료를 다시 보고 Derby 소프트웨어를 다운로드 및 설치하기 바란다. Derby 데이터베이스 소프트웨어를 설치한 후에, 샘플 코드를 사용하여 Derby 데이터베이스로 연결한다. (Listing 1)

샘플 코드 실행하기

----------------------------------------------------

Database Name = Apache Derby

Database Version = 10.8.1.2 - (1095077)

Driver Name = Apache Derby Embedded JDBC Driver

Driver Version = 10.8.1.2 - (1095077)

Database URL = jdbc:derby:test

----------------------------------------------------

첫 번째 예제에서, 데이터베이스 애플리케이션 코드를 개발 및 실행할 수 있는 워크스페이스를 만든다. 우선, 새로운 디렉토리를 만든다. 여기에는 이 글에서 제공하는 코드 파일을 저장할 것이다. 컴파일 및 실행하기 전에, ls 명령을 실행하여 디렉토리에 이 글에서 설명하는 자바 소스 코드만 포함되어 있다는 것을 증명해야 한다.

다음 단계는 자바 컴파일러를 사용하여 Java Virtual Machine (JVM)에 의해 실행될 자바 bytecode 파일을 생성하는 단계이다. 소스 코드를 컴파일 하면, JVM에 있는 bytecode를 실행한다. FirstConnect 클래스에 대해 main 메소드를 호출하면 아웃풋이 생성된다. 문제가 생겼다면, 사이드바 실행되지 않을 경우를 참조하기 바란다. 이 자바 코드는 테스트 데이터베이스를 만든 다음, Apache Derby 임베디드 JDBC 드라이버를 사용하여 연결한다. 이 코드가 SQL 경고와 에러를 핸들하는 모습을 보이려면, 코드를 재실행 할 수 있다. 데이터베이스가 생성되지 않았다(이미 존재함)는 것을 가리키는 경고와, 이전에 디스플레이 되었던 데이터베이스와 JDBC 드라이버에 대한 표준 정보가 나온다.

이 예제는 실행 디렉토리의 내용을 디스플레이 하고, 컴파일 된 자바 클래스 파일과 새로운 데이터베이스 파일을 보여주면서 끝난다. 마지막 포인트가 중요하다. 임베디드 데이터베이스로서 Apache Derby를 사용할 때, 데이터베이스 파일의 기본 위치는 코드가 있는 디렉토리여야 한다. 이 곳이 싫다면 JDBC URL을 수정하여 데이터베이스 파일을 생성할 곳을 설정해야 한다. 이 부분은 이 글의 범위를 벗어나므로, Derby Developer's Guide를 참조하기 바란다. (참고자료)

실행되지 않을 경우

첫 번째 데이터베이스 연결 예제는 매우 단순하지만, 가끔 문제가 발생할 때도 있다. 자바 프로그래밍 언어로 작업할 때 생기는 문제 대부분은 필수 클래스 파일이 CLASSPATH에 없는 경우이다. 예를 들어, 코드가 컴파일 되지만, 코드를 실행할 때 에러 메시지, JDBC Driver org.apache.derby.jdbc.EmbeddedDriver not found in CLASSPATH가 생긴다면, derby.jar 파일을 CLASSPATH에 추가해야 한다. 이 부분은 첫 번째 기술자료에서 설명했다. 이 외에도 javac 툴을 사용하여 자바 소스 코드를 컴파일 하지 않았거나, 잘못된 디렉토리에 있을 경우에도 에러가 생긴다.

## 자바 애플리케이션과 Derby 데이터베이스로 연결하기

이전 섹션에서 실행 방법을 배웠던 자바 코드는 단순한 것으로서, 이제부터 상세히 설명하도록 하겠다. 실행 환경에서 자바 데이터베이스 애플리케이션을 개발하는 일은 어려운 일이다. 향후 기술자료에서 이 부분을 자세히 설명하도록 하고, 이 글에서는 자바 애플리케이션과 임베디드 Apache Derby 데이터베이스를 연결하는 문제에 초점을 맞추겠다. Listing 2에 나타난 이 기술은 연결 프로토콜을 구현하기 위해 JDBC 드라이버를 사용해야 한다.

JDBC를 사용하여 Derby 데이터베이스로 연결하기

**package** com.abreqadhabra.freelec.java.workshop.sample;

**import** java.sql.Connection;

**import** java.sql.DatabaseMetaData;

**import** java.sql.DriverManager;

**import** java.sql.SQLException;

**public** **class** JDBCConnect {

**private** **static** **final** String *driver* = "org.apache.derby.jdbc.EmbeddedDriver";

**private** **static** **final** String *url* = "jdbc:derby:test;create=true";

**public** **static** **void** main(String[] args) {

Connection con = **null**;

DatabaseMetaData dbmd = **null**;

**try** {

Class.*forName*(*driver*);

con = DriverManager.*getConnection*(*url*);

System.*out*.println(con.toString());

// Use the database connection somehow.

dbmd = con.getMetaData() ;

System.*out*.println("\n----------------------------------------------------") ;

System.*out*.println("Database Name = " + dbmd.getDatabaseProductName()) ;

System.*out*.println("Database Version = " + dbmd.getDatabaseProductVersion()) ;

System.*out*.println("Driver Name = " + dbmd.getDriverName()) ;

System.*out*.println("Driver Version = " + dbmd.getDriverVersion()) ;

System.*out*.println("Database URL = " + dbmd.getURL()) ;

System.*out*.println("----------------------------------------------------") ;

} **catch** (SQLException se) {

*printSQLException*(se);

} **catch** (ClassNotFoundException e) {

System.*out*.println("JDBC Driver " + *driver*

+ " not found in CLASSPATH");

} **finally** {

**if** (con != **null**) {

**try** {

con.close();

} **catch** (SQLException se) {

*printSQLException*(se);

}

}

}

}

**private** **static** **void** printSQLException(SQLException se) {

**while** (se != **null**) {

System.*out*.print("SQLException: State: " + se.getSQLState());

System.*out*.println("Severity: " + se.getErrorCode());

System.*out*.println(se.getMessage());

se = se.getNextException();

}

}

}

이 예제 코드는 Apache Derby의 임베디드 JDBC 드라이버에 대한 자바 클래스 이름과 JDBC 데이터베이스 커넥션 URL을 포함하고 있는 두 개의 상수를 정의한다. ij 툴에서 connect 명령어를 실행할 때 사용했던 것이므로 커넥션을 위한 URL은 익숙할 것이다. driver에 할당된 값은 임베디드 드라이버 클래스에 대한 전체 이름이다.

나머지 코드는 main 메소드에 저장된다. JVM에서 디폴트 클래스 로더를 사용하여 이전에 정의했던 driver 클래스를 찾아 인스턴스로 만드는데, 이 때 Class.forname 메소드를 사용한다. 이 메소드는 Derby 임베디드 드라이버에 대한 클래스 파일을 찾는데, 이것은 CLASSPATH에 존재해야 하고, 이를 JVM으로 로딩한다. 로딩 과정 동안, 클래스의 정적 섹션이 처리되고, 드라이버를 JDBC DriverManager 객체에 등록한다.

DriverManager는 팩토리 객체로서 작동한다. 데이터베이스 URL의 경우, DriverManager는 해당 JDBC 드라이버를 사용하여 데이터베이스 커넥션을 리턴한다. 이 단계는 코드의 다음 행에서 실행되는데, 이곳에서 이전에 정의되었던 URL을 사용하여 DriverManager에서 JDBC connection을 요청한다.

연결 요청이 try ... catch 블록 안에서 래핑되었다는 것을 알 수 있다. 그리고 finally 블록 안에서 연결에 대해 close 메소드를 호출한다는 것도 알 수 있다. 이 두 가지 사안에 대해서는 문제 섹션에서 설명한다. 이 글에서는, JDBC로 데이터베이스 연결을 구축하는 다양한 방법들은 소개할 수 없다. 이를테면, 사용자 이름이나 패스워드 같은 보안 정보를 활용하지 않는다. 향후 기술자료에서 데이터베이스 연결 기술 대안들을 살펴보도록 하겠다. (사이드 바 DataSource란 무엇인가? 참조) 그때까지 기다릴 수 없다면, Derby Developer's Guide(참고자료)를 참조하기 바란다.

DataSource란 무엇인가?

이 글에서는 Derby embedded JDBC 드라이버를 사용하여 임베디드 Apache Derby 데이터베이스로 연결하는 부분만 초점을 맞춰 설명한다. 데이터베이스로 연결하는 JDBC 솔루션 중에는 DataSource도 선호된다. 이것은 데이터베이스 URL, 드라이버 클래스, 사용자 이름, 패스워드, 데이터베이스 이름 같은 데이터베이스 연결 상세로부터 여러분이 자유로워질 수 있다. 구체적인 연결 정보에서 분리되기 때문에 데이터베이스 애플리케이션 코드를 변경할 필요 없이 매개변수들을 쉽게 변경할 수 있다. 향후 기술자료에서는 Derby와 DataSource를 사용하는 방법을 설명하겠다. 하지만 JDBC 드라이버와 DriverManager를 배운 다음, DataSource에 대해 배우는 것이 더 효과적이다.

## 데이터베이스 메타데이터

메타데이터를 한번도 못 봤다면 이상한 개념이라고 생각할 수 있다. 실제로는 매우 단순하다. 메타데이터는 데이터를 설명하는 데이터이다. 데이터베이스라는 정황에서, 메타데이터는 데이터베이스 이름, 버전 넘버, 연결하는 JDBC 드라이버의 이름 같은 특정 데이터베이스들을 설명한다. DatabaseMetaData 객체에서 알맞은 메소드를 호출하여 JDBC에서 데이터베이스 메타데이터에 액세스 할 수 있다. (Listing 3)

실행결과

----------------------------------------------------

Database Name = Apache Derby

Database Version = 10.8.1.2 - (1095077)

Driver Name = Apache Derby Embedded JDBC Driver

Driver Version = 10.8.1.2 - (1095077)

Database URL = jdbc:derby:test

----------------------------------------------------

올바른 메타데이터에 액세스 하려면, 이전 예제에서 보여준 것처럼 현재 JDBC Connection에서 새로운 DatabaseMetaData 객체를 만들어야 한다. 그런 다음, Apache Derby JDBC 드라이버에서 제공하는 많은 메타데이터 함수들 중 하나를 호출할 수 있다. (참고자료 섹션에서 전체 리스트를 참조하라.) 이 예제에서, 여러분이 액세스 하는 데이터베이스에 대한 이름과 제품 버전, JDBC 드라이버 이름과 데이터베이스의 버전 넘버, 연결을 구축했던 데이터베이스를 구분하는 전체 JDBC URL 등을 가져온다. 데이터베이스 메타데이터 같은 경우는 다른 많은 데이터베이스나 JDBC 드라이버로 인터랙팅 해야 하는 데이터베이스 애플리케이션을 개발할 경우 매우 유용하다. 이 경우, 메타데이터를 사용하여 특정 데이터베이스의 기능과 JDBC 드라이버를 결정할 수 있다.

## 문제

Apache Derby 데이터베이스로 작업한다면, 데이터베이스 경고와 데이터베이스 에러도 반드시 경험하게 된다. 데이터베이스에 저장한 정보의 중요성을 감안할 때, 데이터베이스 에러와 경고를 올바르게 다루는 것은 매우 중요하다. 다행히도, 나머지 섹션에서 언급하겠지만, 이 작업은 매우 간단하다. 우선 데이터베이스에서 생긴 에러에 대해 애플리케이션에서 제공한 정보를 보는 방법부터 배워야 한다. (Listing 4)

예외 처리 코드

|  |
| --- |
|  |

private static void printSQLException(SQLException se) {

while(se != null) {

System.out.print("SQLException: State: " + se.getSQLState());

System.out.println("Severity: " + se.getErrorCode());

System.out.println(se.getMessage());

se = se.getNextException();

}

}

private static void printSQLWarning(SQLWarning sw) {

while(sw != null) {

System.out.print("SQLWarning: State=" + sw.getSQLState()) ;

System.out.println(", Severity = " + sw.getErrorCode()) ;

System.out.println(sw.getMessage());

sw = sw.getNextWarning();

}

}

Listing 4에서 보듯, SQL 예외와 SQL 경고를 처리하는 방법은 비슷하다. SQLWarning 클래스는 SQLException 클래스에서 상속을 받는다. 하지만, SQL 경고는 SQL 예외보다 덜 심각하기 때문에, 이들을 따로 분리해서 처리하는 것이 더 낫다. 이 두 개의 함수들은 private static으로 선언되기 때문에 다른 클래스가 아닌 main 메소드에서 호출할 수 있다. 실행 코드에서 프로그램 요구 사항에서 나오는 대로 변경할 수 있다.

이 두 개의 함수에서, 경고와 예외를 루핑한다. 일반적인 에러 핸들링 관점에서 보면 이상하지만, 데이터베이스 프로그래밍의 경우에는 이러한 것도 허용해야 한다. 변경 사유는 간단하다. 이벤트들은 데이터베이스 내에서 종종 커플링된다. 예를 들어, 데이터베이스에 다중 행(row)을 삽입하면 칼럼 datatype 또는 해당 이름이 잘못 기입된 것으로 인해 모두 실패되고 여러 에러들을 만나게 된다. 이 모든 것을 올바르게 설명하려면 예외들을 함께 연결하여 호출 코드에 데이터베이스와 관련한 모든 문제들을 공지해야 한다. 이를 쉽게 하기 위해 Apache Derby는 예외 체인에 가장 중요한 예외를 가장 먼저 배치한다.

루핑하면서 에러 또는 경고 정보를 출력해야만 한다. SQL 상태는 다섯 문자 스트링으로 되어 있는데, 이것은 X/OPEN Common Application Environment (CAE) specification, Data Management: SQL, Version 2 SQL 스팩 또는 SQL99 표준 변환에 순응하는 것으로서 프로그램이 특정 데이터베이스 에러 조건에서 복구할 수 있도록 한다. SQL 상태의 처음 두 문자들은 클래스 값이고, 나머지 세 문자는 서브클래스(subclass) 값이다. SQL 상태 값이 00000이면 성공한 것이고, 클래스 값이 01이면 데이터 절단 같은 경고 조건을 나타낸다. SQL 코드는 데이터베이스 스팩의 값으로 되어있다.

SQL 예외 처리는 단순하다. 표준 Java try ... catch 메커니즘을 사용하여 JDBC 메소드 호출을 래핑한다. (Listing 5)

SQL 예외 잡기

try{

// Execute a JDBC operation

} catch (SQLException se) {

printSQLException(se) ;

}

앞서 언급했던 것처럼, SQL 경고는 printSQLWarning 메소드에 의해서 핸들된다. 메소드를 호출하기 전에, 최소한 한 개 이상의 SQL 경고가 있는지를 체크한다. 만일 해당 경고가 존재할 경우, 첫 번째 SQLWarning 객체를 해당 메소드로 전달한다. 향후 기술자료에서 설명하겠지만, 많은 JDBC 객체들은 SQL 경고를 만들기 때문에, 에러-경고 핸들링 코드를 캡슐화 하면 데이터베이스 애플리케이션 코드의 개발과 관리 작업이 수월해진다.

## 연결하기

지금까지 Apache Derby에 JDBC API를 사용하여 데이터베이스 연결을 구축하는데 필요한 많은 컴포넌트들을 설명했다. 이러한 조각들은 필수 기능으로서, Listing 7에는 완전한 연결 예제가 나와있다.

완전한 연결 예제 코드

package com.abreqadhabra.freelec.java.workshop.sample;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.SQLWarning;

import java.sql.DatabaseMetaData;

public class FirstConnect {

private static final String driver = "org.apache.derby.jdbc.EmbeddedDriver";

private static final String url = "jdbc:derby:test;create=true";

private static void printSQLException(SQLException se) {

while (se != null) {

System.out.print("SQLException: State: " + se.getSQLState());

System.out.println("Severity: " + se.getErrorCode());

System.out.println(se.getMessage());

se = se.getNextException();

}

}

private static void printSQLWarning(SQLWarning sw) {

while (sw != null) {

System.out.print("SQLWarning: State=" + sw.getSQLState());

System.out.println(", Severity = " + sw.getErrorCode());

System.out.println(sw.getMessage());

sw = sw.getNextWarning();

}

}

public static void main(String[] args) {

Connection con = null;

DatabaseMetaData dbmd = null;

try {

Class.forName(driver);

con = DriverManager.getConnection(url);

SQLWarning swarn = con.getWarnings();

if (swarn != null) {

printSQLWarning(swarn);

}

dbmd = con.getMetaData();

System.out

.println("\n----------------------------------------------------");

System.out.println("Database Name = "

+ dbmd.getDatabaseProductName());

System.out.println("Database Version = "

+ dbmd.getDatabaseProductVersion());

System.out.println("Driver Name = " + dbmd.getDriverName());

System.out.println("Driver Version = " + dbmd.getDriverVersion());

System.out.println("Database URL = " + dbmd.getURL());

System.out

.println("----------------------------------------------------");

} catch (SQLException se) {

printSQLException(se);

} catch (ClassNotFoundException e) {

System.out.println("JDBC Driver " + driver

+ " not found in CLASSPATH");

} finally {

if (con != null) {

try {

con.close();

} catch (SQLException se) {

printSQLException(se);

}

}

}

}

}

이 클래스는 JDBC를 통해서 임베디드 Apache Derby 데이터베이스로 연결하는데 필요한 것이고, 이 글 초반에 컴파일 및 실행했던 샘플 코드에서 제공된다. 새로운 코드는 프로그램 코드 상단에 있는 다섯 개의 import 문을 포함한 것이다. import java.sql.\*을 사용할 수 있지만, 각각 import 문을 개별적으로 리스팅하면 애플리케이션에서 사용하는 JDBC 객체들을 보다 정확히 만들 수 있다.

이 애플리케이션에는 [Listing 4](http://www.ibm.com/developerworks/kr/library/os-ad-trifecta9/index.html#list4)에서 언급했던 에러 핸들링 함수가 포함되어 있다. 모든 데이터베이스 연산은 try ... catch 블록 안에 들어있고, 여기에는 [Listing 3](http://www.ibm.com/developerworks/kr/library/os-ad-trifecta9/index.html#list3)의 데이터베이스 메타데이터 연산들도 포함되어 있다. 많은 데이터베이스 애플리케이션들은 데이터베이스 애플리케이션 코드와 데이터베이스 간 강결합을 피하여 애플리케이션과 데이터베이스의 관리를 단순화 시킨다. Apache Derby 데이터베이스의 기능 때문에, 애플리케이션과 데이터베이스 간 라인을 흐리게 처리했고, 이러한 분리는 엄격하게 실행될 필요가 없다.

## 요약

이 글에서, 임베디드 Apache Derby 데이터베이스로 연결하는 자바 애플리케이션의 작성 방법을 배웠다. 데이터베이스로 연결하고, 데이터베이스에서 메타데이터를 추출한 다음, 데이터베이스 연결을 끊이기 위해 Apache Derby 프로젝트에서 Type 4 JDBC 드라이버를 사용했다. Derby에 의해서 생성된 SQL 경고와 SQL 예외를 자바 애플리케이션에서 핸들하는 방법도 배웠다. 향후 기술자료에서는 이 글에서 설명한 기술들을 기반으로 Derby 데이터베이스에 쿼리를 실행하고, 자바 애플리케이션에서 얻은 결과를 처리하는 방법을 설명하겠다.

# Java database development with Apache Derby

**Summary:**  Get a thorough overview on how to issue a simple database query against an Apache Derby database and how to process the selected results. Doing so requires the introduction of three new JDBC classes: Statement, ResultSet, and ResultSetMetaData. Learn how to use these classes with a JDBC database connection to quickly and easily extract data from an Apache Derby database into your own Java™ application

<http://www.ibm.com/developerworks/opensource/library/os-ad-trifecta10/index.html>

## JDBC query execution: an overview

The JDBC API is designed in a hierarchical manner, where an object of one type contains objects of other types. For example, you make a database connection via a JDBC Connection; and to send an SQL query to a database, you create a JDBC Statementobject from the appropriate Connection object. Although it may seem logical, this has an important effect: If a database connection is closed for some reason, any database objects that it *contains* are also closed. This hierarchy extends one more level, because the results of a query are accessed by using a ResultSet object, which is contained within the appropriateStatement object. This hierarchy is demonstrated graphically in Figure 1.

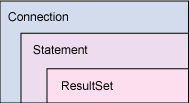


그림 - 인터넷 검색 포털 Naver

Although you won't do so in this article, if you re-execute a JDBC query, any underlying ResultSet object is reused. This means you need to first process a query result completely before reusing a JDBC Statement -- for example, by reissuing a database query -- or you will lose any results from the previous query.

To execute a query against an Apache Derby database, you need to first have a properly initialized database. If you've been following previous articles in this series

, you probably already have a database that you can use for the rest of this article. If not, or if you want to start with a clean slate, you can use the derby.build.sql script file shown in Listing 1.

Initializing your Apache Derby workspace

rb$ mkdir derbyWork

rb$ cd derbyWork/

rb$ unzip ../derby10.zip Archive: ../derby10.zip

inflating: derby.build.sql

inflating: FirstQuery.java

inflating: SecondQuery.java

inflating: ThirdQuery.java

rb$ java org.apache.derby.tools.ij < derby.build.sql > derby.build.out 2> derby.build.err

rb$ javac \*.java

rb$ ls

FirstQuery.class ThirdQuery.class derby.build.sql

FirstQuery.java ThirdQuery.java derby.log

SecondQuery.class derby.build.err test

SecondQuery.java derby.build.out

As shown in Listing 1

, you should first create a new working directory and then extract the source code file that is included with this article (see the Downloads section at the end of this article). The next step is to create an Apache Derby database that is properly initialized, which is easily done by running an SQL script from the Apache Derby ij tool. Finally, compile the three included Java source code files. Although the results aren't explicitly shown (because some of them generate a lengthy output), you can execute each of these Java applications, for example, by entering java FirstQuery at the command prompt. The rest of this article presents the source code for each of these three examples.

## Executing a query

As you learned in the previous section, executing a database query involves three main concepts:

* A database connection
* A query statement
* The query results

To use these objects, you need to first import them into your application, as shown in Listing 2.

# 제목 1

## (제목 2) 지원사항

내용을 적습니다.

### (제목 3) 배포 방식

내용을 적습니다.

### 번호매김 테스트용

번호매김 예제입니다.

* + - * 1. 번호매김
        2. 번호[[1]](#footnote-1)매김
        3. 번호매김

## 제목 2

점목록 예제입니다.

### 점목록 테스트용

먼저 [[2]](#footnote-2)는 다음의 조건을 만족해야 한다.

* 점목록
* 점목록
* 점목록

### 표 테스트용

표에는 표 캡션을 삽입합니다. 표 캡션은 표의 위, 왼쪽에 삽입합니다.

표 -1 전달 인자 설명

|  |  |
| --- | --- |
| 구분 | 인자 설명 |
| 경로 | 일반적으로 exe를 실행시킬 때 기본적으로 넘어가는 인자로 의 경로 정보를 표시 |
| [[3]](#footnote-3) 윈도 핸들 한글은 | 와 통신을 하기 위한 윈도우 핸들 값. (10진수로 표현) |
| 기타 인자 | 런처 실행 시 필요한 추가 정보를 담는다. 개발사가 임의로 설정할 수 있으며, 공백 문자도 사용할 수 있다. |

## UI 제어 설정

### 목록 및 소스 관련 테스트용

가 실행된 후에 발생하는 업데이트 프로세스는 가 담당한다. 는 업데이트 과정을 UI에 표시해야 하며, 이를 위해 간 통신 설정을 해야 한다.

* + - * 1. 먼저 IoutBound.h를 include한다.
        2. 에게 UI 제어 메시지를 전송한다.

UI 제어 메시지 전송 함수는 다음과 같다.

(nCode)

::SendMessage(HWND, WM\_COPYDATA, HWND, PCOPYDATASUCT);

// 전달변수에 대한 설명은 다음과 같다.

// HWND : 에서 제공한 윈도 핸들

// WM\_COPYDATA : 윈도 출력 메시지

### 업데이트 완료 표시 설정

기본적으로 업데이트 진행상황 표시 설정과 동일하다.

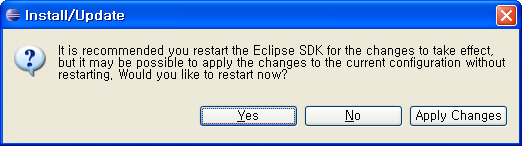


그림 0-1 Install/Update 확인

참고문헌

[1] 참고문헌 1

[2] 참고문헌 2

용어정리

용어1

용어설명

용어2

용어설명

1. [↑](#footnote-ref-1)
2. TR은 Techincal Report의 [↑](#footnote-ref-2)
3. nano는 서버 개발 환경에 필요한 네트워크, DBMS, XML 등 각종 라이브러리로 구성되어 있으며 C++로 구현된 범용 라이브러리이다. (각주텍스트) [↑](#footnote-ref-3)