Borland®

BDE 애플리케이션을 dbExpress로 마이그레이션하기

provide/resolve 구조를 사용하는 새로운 접근 방식

Bill Todd (The Database Group, Inc. 사장)

볼랜드 Software Coporation

2002년 9월

목차

볼랜드 dbExpress-새로운 비전

여러 데이터베이스를 지원하기 위한 공통 API를 생성하려는 시도는 몇가지의 문제점을 가지고 있습니다. 어떤 방식은 너무 많은 것을 지원하기 때문에 크고 느리고 배포에 어려움이 있었습니다. 다른 방식은 최소한의 공통 요소만을 제공한 때문에 개발자들이 일부 데이터베이스의 특정 기능을 이용할 수 없었습니다. 또 다른 방식은 드라이버 작성의 복잡성 때문에 제한된 기능을 제공하거나 속도가 느리거나 버그를 가지고 있었습니다. 볼랜드 dbExpress는 여러 데이터베이스의 공통 API 제공을 위한 새로운 접근 방식과, 데이터 편집 및 수정 작업을 관리하는 볼랜드의 입증된 provide/resolve 구조를 이용하여 이 문제를 해결했습니다. 이 글에서는 dbExpress 와 provide/resolve 방식의 구조를 살펴보고, dbExpress 컴포넌트를 사용하여 데이터베이스 애플리케이션을 개발하는 방법을 보여주고, BDE(Borland Database Engine)을 사용하는 데이터베이스 애플리케이션을 dbExpress로 변환하는 과정을 설명합니다.

dbExpress 구조

dbExpress는 다음과 같은 6가지 목표를 충족시키도록 설계되었습니다.

- 크기와 시스템 리소스 사용량의 최소화
- 속도 최대화
- 크로스플랫폼 지원 제공
- 더 쉬운 배포
- 쉬운 드라이버 개발
- 메모리 사용과 네트워크 트래픽에 대한 개발자의 제어 능력 확대

white paper

dbExpress 드라이버는 기능이 극히 제한되어 있기 때문에 작고 빠릅니다. 각 드라이버는 윈도우 플랫폼 상의 단일 DLL인 동시에 리눅스 플랫폼 상의 단일 Shared Object 라이브러리로서 구현되어 있습니다. dbExpress 드라이버는 메타데이터 페치, SQL 명령 및 프로시져의 실행, 읽기 전용 단방향 커서 반환을 위한 5개의 인터페이스를 구현하고 있습니다. 그러나, 볼랜드의 provide/resolve데이터 액세스 전략에 따라 DataSetProvider 및 ClientDataSet와 함께 사용하는 경우 dbExpress는 SQL 데이터베이스에 있는 데이터로 작업할 수 있는 모든 기능, 고성능, 높은 병행성에 있어서 최적화된 성능을 발휘 합니다.

provide/resolve구조의 동작 방식

provide/resolve구조는 4개의 컴포넌트를 사용하여 데이터를 액세스하고 편집합니다. 첫번째는 SQLConnection 컴포넌트로서, 사용할 데이터베이스에 대한 dbExpress 드라이버에 연결합니다. 다음은 dbExpress 데이터셋 컴포넌트 가운데 하나로, SQL SELECT 문장을 실행하거나 스토어드 프로시저를 호출한 결과 데이터를 전달합니다. 세번째 컴포넌트는 DataSetProvider이고, 네번째는 ClientDataSet입니다. ClientDataSet를 열면 DataSetProvider에 데이터를 요청하게 됩니다. DataSetProvider는 쿼리 또는 스토어드 프로시저 컴포넌트를 열고 레코드를 읽어온 후에 쿼리 혹은 스토어드 프로시저 컴포넌트를 닫고 필요한 메타데이터와 함께 레코드들을 ClientDataSet에 전달합니다.

ClientDataSet은 레코드들을 조회 또는 수정되는 동안 이들을 메모리에 보유하고 있습니다. 레코드가 코드 또는 유저 인터페이스를 통하여 추가, 삭제 또는 수정되면 ClientDataSet이 모든 변경 사항을 메모리에 기록합니다. 데이터베이스를 갱신하려면 ClientDataSet의 ApplyUpdates 메소드를 호출합니다. ApplyUpdates는 변경 기록을 DataSetProvider로 전송합니다. 프로바이더는 트랜잭션을 시작하고 데이터베이스에 변경 사항을 적용하기 위한 SQL 문을 생성하여 실행합니다. 모든 변경 사항이 성공적으로 적용되면 프로바이더는 트랜잭션을 커밋합니다. 그렇지 않은 경우 트랜잭션은 롤백됩니다. 데이터베이스 갱신은 변경 사항이 비즈니스 규칙을 위반하는 것이거나 다른 사용자가 갱신하려는 레코드를 변경하였거나 하는 경우에는 실행되지 않습니다. 오류가 발생하면 트랜잭션은 취소되고 ClientDataSet의 OnReconcileError 이벤트가 발생하여 사용자가 오류를 처리하도록 합니다.

provide/resolve구조의 이점

짧은 트랜잭션 시간

긴 트랜잭션은 데이터베이스 서버가 락을 유지하도록 하여 동시성을 감소시키고 데이터베이스 서버의 리소스를 소모합니다. provide/resolve구조에서는 갱신의 적용 순간에만 트랜잭션이 존재하므로, 리소스 소모를 현저하게 낮추고 혼잡한 데이터베이스 서버의 동시성을 향상시킵니다.

모든 행의 편집 가능

다중 테이블 조인, 스토어드 프로시저 또는 읽기 전용 뷰에서는 행을 직접 편집할 수 없습니다. 필드 객체의 ProviderFlags 속성을 사용하여 갱신할 필드를 식별하고 DataSetProvider의 OnGetTableName 이벤트를 사용하여 테이블의 이름을 제공하면 많은 읽기 전용 데이터셋을 쉽게 편집할 수 있습니다.

즉시적인 정렬 및 탐색

ClientDataSet은 레코드들을 메모리에 가지고 있으므로 정렬이 빠릅니다. 메모리내의 정렬이 너무 느리면 디자인 타임이나 런타임에서 ClientDataSet 데이터에 인덱스를 만들 수 있습니다. 이런 메모리 인덱스를 이용하면 데이터베이스에 인덱스를 유지하는 오버헤드 없이 즉시 레코드의 보기 순서를 변경하거나 레코드를 찾아갈 수 있습니다.

자동 요약 정보

ClientDataSet은 Sum(Price)-Sum(Cost)와 같이 사용자가 정의한 복잡한 합 계산을 자동으로 보존합니다. 합 계산을 필드별 또는 필드의 결합으로 그룹하여 그룹 합계를 구할 수 있습니다. 또한, Min, Max, Count 및 Avg (평균) 합계를 사용할 수 있습니다.

데이터 서브셋 뷰

SQL WHERE 구문에서 필터 표현을 사용하면 데이터베이스 서버에서 추가로 쿼리를 실행하지 않고도 ClientDataSet에 있는 레코드의 서브셋을 쉽게 표시할 수 있습니다.

다중 동시 데이터 뷰

ClientDataSet 커서를 복제하는 기능으로 ClientDataSet에 있는 데이터의 여러 서브셋을 동시에 볼 수 있습니다. 또한 동일 데이터를 다르게 정렬하여 볼 수도 있습니다.

서버에 부하를 주지 않는 계산 필드

디자인타임에 ClientDataSet에 계산 필드를 추가하여 계산 필드를 메모리 데이터셋의 부분으로 만들 수 있습니다. 계산은 컴파일된 델파이 또는 C++ 언어 코드로 실행되므로, SQL 문 안에 있는 계산된 컬럼이나 스토어드 프로시저 안의 계산보다 더 빠르고 훨씬 더 복잡해질 수 있으면서도 데이터베이스 서버에 저장 공간이나 계산 부담을 주지 않습니다.

실질적으로 제한이 없음

메모리에 레코드들을 보유함으로 해서 작업할 수 있는 레코드 수에 제한이 따를 것처럼 느껴질 수 있습니다. 그러나, 전통적인 클라이언트/서버 애플리케이션 디자인에서는 네트워크 트래픽과 데이터베이스 서버의 부하를 최소화 하기 위해 작은 레코드 셋만을 셀렉트해왔다는 점을 상기해보십시오. 예외적으로 많은 레코드들을 작업해야 하는 경우라도, 각각 100바이트를 가지는 10,000개의 레코드의 경우 단지 1MB의 메모리만을 필요로 합니다. 대단히 많은 수의 레코드로 작업해야 하는 드문 경우라도, ClientDataSet와 DataSetProvider에는 레코드의 일부를 가져와서 편집하고 메모리에서 제거한 후에 다음 레코드 그룹을 가져오는 속성 및 이벤트들이 포함되어 있습니다.

더욱 용이한 배포

dbExpress를 사용하는 애플리케이션은 단지 두 개의 DLL만을 필요로 합니다. 하나는 dbExpress 드라이버이며 (예를 들어 볼랜드 InterBase의 경우 DBEXPINT.DLL), 다른 하나는 ClientDataSet 지원 라이브러리인 MIDAS.DLL입니다. 이 두 개의 DLL은 합쳐도 크기가 0.5 MB 미만입니다. 따라서 애플리케이션의 크기를 최소화할 수 있고 설치가 간단합니다. 이들 DLL들을 배포하는 것이 마음에 들지 않는다면 EXE에 직접 컴파일하여 포함시킬 수도 있습니다. 리눅스에서의 배포도 DLL 대신에 두 개의 shared object 라이브러리를 사용한다는 점을 제외하고는 동일합니다.

더 쉬운 드라이버 개발

dbExpress 드라이버는 온라인 헬프에 설명된 5개의 인터페이스를 구현해야 합니다. 볼랜드는 모델로서 MySQL 드라이버의 소스 코드를 제공합니다.이를 참조하면 데이터베이스 판매회사는 강력한 고성능 드라이버를 쉽게 만들 수 있습니다. 잘 사용되어지지 않거나 오래된 데이터베이스로 작업하는 경우 상업적으로 구입할 수 있는 드라이버가 없는 경우, 드라이버를 직접 만들어 사용할 수도 있습니다.

dbExpress 애플리케이션 개발

기존의 BDE 애플리케이션을 dbExpress로 변환하기 이전에 dbExpress 컴포넌트의 사용 방법을 잘 알아야 합니다. 여기에서는 dbExpress 애플리케이션을 단계별로 생성하면서 각 컴포넌트에 대하여 설명하겠습니다. 이 예는 윈도우 플랫폼에서 볼랜드 Delphi를 사용하여 개발하는 것이지만 윈도우에서 C++Builder로 개발하는 경우, 그리고 리눅스 플랫폼에서 볼랜드 Kylix를 사용하는 절차도 이와 동일합니다.

샘플 애플리케이션은 InterBase 샘플 EMPLOYEE.GDB 데이터베이스를 사용하며, Employee 테이블과 Salary History 테이블 사이에는 일대다 관계가 있습니다. 샘플 애플리케이션은 다음과 같은 dbExpress의 기능을 보여줍니다.

- 마스터 테이블의 필드에 디테일 테이블을 포함시키기
- SQLQuery, DataSetProvider 및 ClientDataSet 컴포넌트를 이용하여 데이터 편집하기
- DataSetProvider와 ClientDataSet 없이 SQLQuery를 읽기 전용 데이터셋으로 사용하기
- ClientDataSet의 수정사항을 데이터베이스에 적용하기
- 수정사항을 데이터베이스에 적용할 때 발생하는 오류 처리하기

SQLConnection 컴포넌트

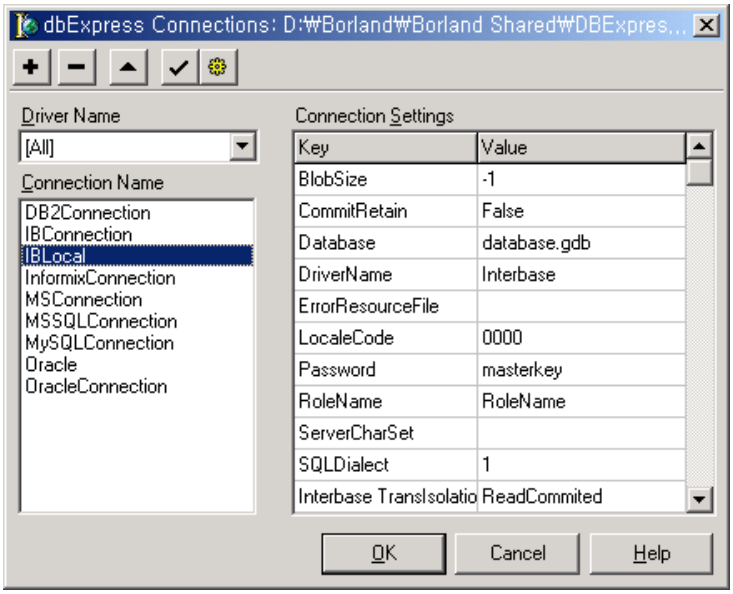
간단한 dbExpress 애플리케이션을 생성하려면 다음 절차에 따릅니다.

1. **새 애플리케이션**을 생성한 후에 **데이터 모듈**을 추가합니다. 데이터 모듈의 이름을 **MainDm**로 정합니다.
2. **Project Options** 다이얼로그를 사용하여 메인 폼이 생성되기 전에 데이터 모듈이 자동으로 생성되도록 합니다.
3. **SQLConnection 컴포넌트**(컴포넌트 팔레트의 dbExpress 페이지)를 **데이터 모듈**에 놓습니다.
4. SQLConnection 컴포넌트의 이름을 **EmployeeConnection**이라고 정하고 DriverName 속성을 **InterBase**로 설정합니다.
5. Params 속성의 **속성 에디터**를 열고 **Database 파라미터**를 샘플 InterBase 데이터베이스인 **EMPLOYEE.GDB**의 경로로 설정합니다. 디폴트로 설치 했다면 `c:\program files\borland \interbase\examples\database\employee.gdb` 일 것입니다.
6. InterBase 서버에 연결하기 위하여 다른 값이 필요하다면, **UserName**과 **Password** 매개 변수를 변경합니다.
7. **LoginPrompt** 속성을 **false**로 설정하여 프로그램을 실행할 때마다 사용자 이름과 비밀번호를 입력하지 않게 합니다.
8. **Connected** 속성을 **true**로 설정하여 연결 상태를 테스트한 후에 다시 **false**로 설정합니다.

SQLConnection 컴포넌트는 여러 개의 데이터셋 컴포넌트에 대해서 데이터베이스 연결을 제공합니다. 많은 데이터베이스에 동시에 연결할 경우에는 여러 개의 SQLConnection 컴포넌트를 사용할 수 있습니다. 데이터베이스에 대한 연결을 정의하는 방법은 세 가지가 있습니다. 이전에 이름을 붙였던 연결을 사용하거나, 새로운 이름의 연결을 생성하거나, SQLConnection 컴포넌트의 Params 속성에 연결 파라미터를 넣는 방법이 있습니다. 기존의 연결 이름을 사용하려면 ConnectionName 속성에 지정하면 됩니다.

dbExpress 커넥션 에디터

새로운 연결 이름을 생성하려면 **SQLConnection 컴포넌트**를 더블 클릭하여 dbExpress 커넥션 에디터를 엽니다. 왼쪽에 있는 Connection Name 리스트박스에 이미 정의된 연결들이 나열됩니다. Driver 드롭다운 리스트에서 **Connection Names**를 **필터링**하여 선택한 드라이버에 대한 연결만 표시할 수도 있습니다. 오른쪽에 있는 Connection Settings 그리드는 선택한 연결에 대한 연결 설정을 표시합니다. 생성한 모든 연결은 `dbxconnections.ini` 파일에 저장됩니다.



커넥션 에디터

연결 파일

연결 이름을 생성한 후에는 SQLConnection의 ConnectionName 속성에 이를 지정할 수 있습니다. 연결 이름을 사용하는 경우 배포할 때 애플리케이션과 함께 연결 파일을 배포하거나, 대상 컴퓨터에서 기존의 연결 파일을 찾아서 연결 이름을 추가해야 합니다.

SQLConnection 컴포넌트의 Params 속성

SQLConnection 컴포넌트의 DriverName 속성을 설정하는 또 다른 방법이 있습니다. DriverName 속성의 드롭다운 리스트에는 시스템에 설치되어 있는 모든 드라이버를 보여줍니다. 드라이버 정보는 `dbxdrivers.ini` 파일에 들어 있습니다. DriverName 속성을 설정하면 드라이버 파일에 들어 있는 정보를 이용하여 LibraryName와 VendorLib 속성도 같이 설정됩니다. LibraryName에는 dbExpress 드라이버 DLL의 이름이 들어 있고, VendorLib에는 데이터베이스 판매회사의 클라이언트 라이브러리 파일이 들어 있습니다.

앞의 화면과 같이 SQLConnection 컴포넌트의 Params 속성에 커넥션 에디터의 연결 매개변수를 입력합니다. 이 방법을 사용하면 연결 정보가 모두 애플리케이션에 포함됩니다. 애플리케이션 사용자가 연결 파라미터를 변경할 수 있도록 하려면 애플리케이션 자체의 설정 파일에 이를 저장하고 애플리케이션 내부에서 또는 별도의 설정 프로그램을 이용하여 파일을 편집하는 방법을 알려줄 수 있습니다. 이렇게 하면 각 애플리케이션이 모든 것을 완전하게 갖추게 됩니다.

또한, SQLConnection 컴포넌트는 트랜잭션 제어를 위한 StartTransaction, Commit 및 Rollback 메소드를 제공합니다. 결과를 반환하지 않는 SQL 문장을 실행하는 경우에는 SQLConnection 컴포넌트의 Execute 또는 ExecuteDirect 메소드를 사용할 수 있으며, 데이터셋 컴포넌트는 필요하지 않습니다. 작업하고 있는 데이터베이스의 메타데이터를 액세스해야 하는 경우 SQLConnection의 GetTableNames, GetFieldNames 및 GetIndexNames 메소드를 이용할 수 있습니다.

데이터셋 컴포넌트

dbExpress는 SQLDataSet, SQLQuery, SQLStoredProc 및 SQLTable의 네 가지 데이터셋 컴포넌트를 제공합니다. SQLDataSet은 새로운 애플리케이션을 작성할 때 선택하는 컴포넌트입니다. CommandType 속성을 설정하면 SQL 문장을 실행하고, 스토어드 프로시저를 호출하거나 테이블에 있는 모든 행과 열에 액세스할 수 있습니다. 나머지 데이터셋 컴포넌트들은 각각의 BDE 해당 컴포넌트들과 가급적 비슷하게 디자인되어 있습니다. 이러한 컴포넌트를 사용하면 BDE 애플리케이션을 dbExpress로 쉽게 변환할 수 있습니다.

SQLQuery

SQLQuery 컴포넌트의 속성과 메소드는 BDE Query 컴포넌트와 아주 흡사합니다. SQLQuery는 읽기 전용의 단방향 결과 세트를 반환하므로 BDE에 관련된 메소드/속성이나 편집에 관련된 속성/메소드는 없습니다. SQLQuery는 DML과 DDL 양쪽 모두의 SQL을 실행하기 위해 사용될 수 있습니다. 결과 셋에 커서를 반환하는 문장의 경우에는 SQLQuery에서 Open 메소드를 호출하거나 Active 속성을 true로 설정합니다. 커서를 반환하지 않는 문장의 경우에는 ExecSQL 메소드를 호출합니다. 다음과 같이 샘플 애플리케이션의 구축을 계속합니다.

1. **3개의 SQLQuery 컴포넌트**를 **데이터 모듈**에 끌어다 놓습니다.
2. 세 컴포넌트의 **SQLConnection** 속성을 모두 **EmployeeConnection**로 설정합니다.
3. 각각의 이름을 **EmployeeQry**, **HistoryQry** 및 **DeptQry**로 정합니다.
4. **EmployeeQry**의 SQL 속성을 다음과 같이 설정합니다.

```
SELECT * FROM EMPLOYEE
WHERE DEPT_NO = :DEPT_NO
ORDER BY LAST_NAME
```

5. **HistoryQry**의 SQL 속성을 다음과 같이 설정합니다.

```
SELECT * FROM SALARY_HISTORY
WHERE EMP_NO = :EMP_NO
```

6. **DeptQry**의 SQL 속성을 다음과 같이 설정합니다.

```
SELECT DEPT_NO, DEPARTMENT FROM
DEPARTMENT
ORDER BY DEPARTMENT
```

7. **DataSource** 하나를 **데이터 모듈**에 떨어뜨리고, 이름을 **EmpLinkSrc**로 정한 후에 DataSet 속성을 **EmployeeQry**로 설정합니다.
8. **HistoryQry**의 DataSource 속성을 **EmpLinkSrc**로 설정합니다.
9. **EmployeeQry**로 되돌아가서 **Params** 속성 편집기를 열고 DEPT_NO 매개변수의 값을 **XXX**로 설정합니다. 이것은 잘못된 값이므로 사용자가 올바른 부서 번호를 입력할 때까지 아무 레코드도 표시되지 않습니다.
10. **EmployeeQry**를 더블 클릭하여 **필드 에디터**를 열고 모든 필드에 대하여 **필드 객체**를 추가합니다.
11. **EMP_NO** 필드를 선택하여 **ProviderFlags** 속성을 확장한 후에 **pfInKey**를 **true**로 설정합니다. pfInKey

플래그를 true로 설정하면 EMP_NO 필드가 기본키로 인식됩니다. SQL 문장이 변경 사항을 데이터베이스에 적용하도록 구성하려면 DataSetProvider(나중에 추가)가 이 정보를 알아야 합니다.

12. **FULL_NAME** 필드를 선택하여 **ProfiderFlags** 속성을 확장한 후에 **pfInUpdate**와 **pfInWhere**를 **false**로 설정합니다. FULL_NAME은 계산 필드이므로 DataSetProvider에 의하여 생성되는 SQL 문장의 WHERE 절에 포함되거나 갱신되지 않아야 합니다.
13. **EmployeeQry**의 **Active** 속성을 **true**로 설정합니다.
14. **HistoryQry**를 더블 클릭하여 **Fields Editor**를 열고 모든 **필드 객체**를 추가합니다.
15. Fields Editor에서 **EMP_NO** 필드를 선택한 후에 ProviderFlags 속성에서 **pfInKey**를 **true**로 설정합니다. 이 과정을 주키의 일부인 CHANGE_DATE와 UPDATER_ID 필드에 대하여 반복합니다.
16. **NEW_SALARY** 필드도 계산되는 필드이므로 이것을 선택하여 **pfInUpdate**와 **pfInWhere** 프로바이더 플래그를 **false**로 설정합니다.
17. **EmployeeQry**와 **HistoryQry**의 **Active** 속성을 **false**로 설정합니다.
18. **EmployeeConnection**의 Connected 속성을 **false**로 설정합니다.

SQLTable

SQLTable 컴포넌트는 BDE의 Table 컴포넌트와 비슷합니다. BDE에서와 같이 테이블에 있는 모든 행과 열을 반환합니다.그렇기 때문에 클라이언트/서버 애플리케이션에 알맞은 방법은 아닙니다. 이것의 유일한 장점은 BDE Table 컴포넌트를 사용하는 로컬 데이터베이스 애플리케이션을 dbExpress와 데이터베이스 서버로 빠르게 변환하는 데에 있습니다.

SQLTable을 사용하려면 **SQLConnection** 속성을 **SQLConnection 컴포넌트**로 설정합니다. **TableName** 속성을 액세스하고자 하는 **테이블의 이름**으로 설정합니다. **Active** 속성을 **true**로 설정하거나 **Open 메소드**를 **호출**하여 테이블을 엽니다.

SQLStoredProc

BDE 애플리케이션에서 BDE StoredProc 컴포넌트를 사용한 것과 같은 방법으로 SQLStoredProc 컴포넌트를 사용하여 스토어드 프로시저를 호출합니다. SQLDataSet 컴포넌트를 사용하여 스토어드 프로시저를 호출할 수도 있으나, SQLStoredProc 컴포넌트는 BDE StoredProc 컴포넌트와 거의 같기 때문에 이것을 사용하여 BDE 애플리케이션을 전환하는 것이 더 쉽습니다.

SQLStoredProc를 사용하려면 **SQLConnection** 속성을 **SQLConnection 컴포넌트**로 설정하고, **StoredProcName** 속성을 실행하고자 하는 **스토어드 프로시저의 이름**으로 설정합니다. 스토어드 프로시저가 레코드들에 대한 커서를 반환하는 경우에는 **SQLStoredProc의 Active** 속성을 **true**로 설정하거나 **Open** 메소드를 호출하여 실행합니다. 스토어드 프로시저가 결과 셋에 대한 커서를 반송하지 않으면 ExecProc 메소드를 호출하여 실행합니다.

SQLDataSet

SQLDataSet를 사용하려면 **SQLConnection** 속성을 사용하고자 하는 **SQLConnection 컴포넌트**로 설정합니다. 다음으로, **CommandType** 속성을 ctQuery, ctStoredProc 또는 ctTable에 설정합니다. 대개의 경우 기본값인 ctQuery을 사용하게 됩니다. CommandText 속성의 값은 CommandType의 값에 따라 다릅니다. CommandType이 ctQuery인 경우, CommandText는 실행할 SQL 문을 포함합니다. CommandType이 ctStoredProc인 경우, CommandText는 스토어드 프로시저의 이름입니다. CommandType이 ctTable인 경우, CommandText는 테이블의 이름입니다. Params 속성을 사용하여 "파라미터화된" 쿼리 또는 스토어드 프로시저에 대한 파라미터를 제공하고, DataSource 속성을 사용하여 SQLDataSet을 마스터/디테일 관계에 있는 다른 데이터셋 컴포넌트에 연결합니다. SQLDataSet가 레코드셋에 대한 커서를 반환하는 경우에는 Open 메소드를 호출하거나 Active 속성을 true로 설정하여 엽니다. 만일 결과 셋을 반송하지 않으면 ExecSQL 메소드를 호출합니다.

SQLDataSet는 읽기 전용의 단방향 커서만을 제공합니다. 리포트 인쇄 등에서처럼 이것이 필요한 유일한 액세스라면

레포트 툴의 요구 사항에 따라 SQLDataSet만을 사용하거나 DataSource 컴포넌트와 함께 사용할 수 있습니다. 레코드 안을 앞 혹은 뒤로 이동하거나 데이터를 편집해야 하는 경우에는 DataSetProvider와 ClientDataSet를 추가하거나 이 문서에서 다음에 설명할 SimpleDataSet 컴포넌트를 사용합니다.

SQLConnection의 메소드들이 제공하는 정보 이상의 자세한 메타데이터 정보가 필요한 경우에는 SQLDataSet 컴포넌트의 SetSchemaInfo 메소드를 이용합니다. SetSchemaInfo는 SchemaType, SchemaObject 및 SchemaPattern의 세가지 파라미터를 가집니다. SchemaType은 stNone, stTables, stSysTables, stProcedures, stColumns, stProcedureParams, stIndexes 가운데 하나일 수 있습니다. 이 파라미터는 SQLDataSet가 열렸을 때 포함되는 정보의 종류를 표시합니다. SQL 문장이나 스토어드 프로시저를 사용하여 테이블에서 정보를 추출할 때 스키마 종류는 stNone로 설정됩니다. 다른 스키마 종류들은 각각 반환되는 정보에 적절한 구조에 맞춰 데이터셋을 생성합니다. SchemaObject는 스토어드 프로시저 또는 테이블 이름이 필요한 경우의 그 이름입니다. SchemaPattern은 결과 셋을 필터링할 SQL 패턴을 제공합니다. 예를 들어, SchemaType이 테이블이고 SchemaPattern이 'EMP%'인 경우 데이터셋은 EMP로 시작하는 테이블만 포함합니다.

SimpleDataSet

BDE을 사용하여 레코드를 보고 편집하려면 TQuery를 데이터 모듈에 끌어다 놓고 두가지 속성을 설정하면 됩니다. dbExpress의 경우에는 SQLQuery 또는 SQLDataSet와 DataSetProvider, ClientDataSet를 데이터 모듈에 끌어다 놓고 속성을 설정하여 이들을 함께 연결시킵니다. 세가지 컴포넌트를 추가하고 속성을 설정하는 추가적인 시간을 절약하는 두 가지 방법이 있습니다.

볼랜드 Delphi 7 Studio에는 SimpleDataSet 컴포넌트가 도입되었습니다. 이 SimpleDataSet은 SQLDataSet, DataSetProvider 및 ClientDataSet를 하나의 컴포넌트로 결합한 것입니다. 만일 단일 테이블 또는 스토어드 프로시저로부터 레코드를 보고 편집하는 작업만 필요한 경우 **SimpleDataSet**이 빠른 방법입니다. 단지 이것을 **데이터 모듈**에 추가하고 **Connection** 속성을 SQLConnection 컴포넌트로 설정한 후 내장된 SQLDataSet 컴포넌트의 **CommandType**과 **CommandText** 속성을 설정하면 끝입니다. SimpleDataSet 컴포넌트는 약간의 제한 사항을 가지고 있습니다.

- 멀티티어 애플리케이션에서는 사용할 수 없습니다. 앞으로 애플리케이션을 멀티티어로 변환해야 할 가능성이 있다면 분리된 컴포넌트들을 사용하십시오.
- 자식 데이터셋을 데이터셋 필드로 SimpleDataSet에 연결할 수 없습니다.
- 내부 DataSetProvider의 이벤트들이 모두 노출되지 않습니다.
- 내부 DataSetProvider의 Options 속성이 노출되지 않습니다. 프로바이더 옵션을 디자인타임 또는 런타임에 설정할 수 없습니다.
- 디자인타임에 내부 데이터셋에 대한 필드 객체를 생성할 수 없습니다. 이것은 디자인타임에 필드 객체의 ProviderFlags 속성을 설정할 수 없다는 것을 의미합니다. 이것은 코드내에서 할 수 밖에 없습니다.
- ClientDataSet 컴포넌트의 내부 데이터베이스인 볼랜드 MyBase만을 사용하고 데이터베이스 서버에 연결하지 않는 경우에는 ClientDataSet만을 사용하게 되면 애플리케이션의 리소스 요구를 줄일 수 있습니다.
- 내부에 포함된 SQLDataSet의 속성 및 메소드는 SQLQuery의 속성 및 메소드만큼 BDE Query 컴포넌트의 속성과 비슷하지 않습니다. 그러므로, 프로젝트를 변환하는 과정에서 SimpleDataSet을 사용하면 코드 변경이 더 많이 필요하게 됩니다.

SimpleDataSet가 제공하는 것보다 더욱 유연성이 필요한 경우에는 **SQLQuery, DataSetProvider, ClientDataSet**를 **폼** 또는 데이터 모듈에 끌어다 놓습니다. SQLQuery의 **Connection** 속성을 설정하고, DataProvider의 **DataSet** 속성을 SQLQuery에 연결하도록 설정합니다. ClientDataSet의 **ProviderName** 속성을 DataSetProvider에 연결하도록 설정하고, **3개의 컴포넌트** 모두 선택한 후에 메인 메뉴에서 **Component | Create Component Template**를 선택합니다. 새로운 템플릿을 위한 **클래스 이름**과 **팔레트** 페이지를

지정합니다. 이제, 하나의 컴포넌트를 끌어다 놓는 것만큼 쉽게 세 개의 별개의 컴포넌트를 데이터 모듈에 끌어다 놓을 수 있습니다.

SQLMonitor

DbExpress의 마지막 컴포넌트인 SQLMonitor는 애플리케이션의 성능 튜닝을 도와주기 위한 컴포넌트입니다. SQLMonitor는 SQLConnection 컴포넌트와 이것과 연결된 데이터베이스 서버 사이를 겨쳐가는 모든 SQL 문을 체크합니다. SQL 문장은 파일에 기록되거나, 메모 컴포넌트에 표시되거나, 기타 원하는 방법으로 처리될 수 있습니다.

데이터 액세스 컴포넌트

양방향 스크롤과 데이터를 갱신하는 기능이 필요한 경우, DataSetProvider 및 ClientDataSet 컴포넌트를 사용해야 합니다.

DataSetProvider

DataSetProvider는 DataSet 속성을 통하여 dbExpress 데이터셋 중 하나와 연결됩니다. DataSetProvider는 요청에 따라 ClientDataSet에 데이터를 제공하고, ClientDataSet이 제공하는 변경 로그를 이용하여 데이터베이스 갱신을 위한 SQL DML 문장을 생성합니다.

샘플 애플리케이션을 위한 데이터 모듈로 돌아가서, 다음의 단계를 수행합니다.

1. **DataSetProvider**(컴포넌트 팔레트의 Data Access 페이지)를 추가합니다.
2. **DataSet** 속성을 **EmployeeQry**로 설정하고, **Name** 속성을 **EmployeeProv**로 설정합니다.

DataSetProvider의 UpdateMode 속성은 사용자가 갱신하려는 레코드를 다른 사용자가 갱신하였는지에 대해 프로바이더가 어떻게 판단할지를 제어할 수 있습니다. 프로바이더가 데이터베이스를 갱신하는 SQL 문장을 발생시킬 때 각 `UPDATE`와 `DELETE` 문장에는 레코드를 식별하기 위해 `WHERE` 절이 포함됩니다. 만약 UpdateMode를 `upWhereAll`로 설정하면, 그 레코드에 속한 필드들 중 Blob 필드가 아니고 pfInWhere 프로바이더 플래그가 false가 아닌 필드의 원래 값이 WHERE 절에 포함됩니다. 이것은 다른 사람이 이 필드들 중 하나라도 변경하면 UPDATE 또는 DELETE가 실패한다는 것을 의미합니다. UpdateMode를 upWhereChanged로 설정하면 변경된 필드만이 WHERE 절에 포함됩니다. UpdateMode를 upWhereKeyOnly로 설정하면 기본키 필드만을 포함하는 WHERE 절이 생성됩니다.

프로바이더의 Options 속성에는 provide/resolve 과정을 제어하는 여러 플래그가 포함됩니다. `poCascadeDeletes` 옵션이 true이면 삭제한 마스터 레코드에 대한 디테일 레코드를 삭제하기 위한 SQL 문장을 생성하지 않습니다. 프로바이더는 데이터베이스 서버가 연쇄 삭제를 지원하여 자동으로 디테일 레코드를 삭제하는 것으로 가정합니다. `poCascadeUpdates` 옵션은 마스터 테이블의 기본키에 대한 변경 사항에 대하여 같은 기능을 제공합니다.

프로바이더에 데이터를 제공하는 SQLQuery가 SQL 문장에 `ORDER BY` 절을 가지는 경우, 레코드들이 ClientDataSet에서의 순서를 유지하도록 하려면 `poRetainServerOrder` 플래그를 true로 설정합니다. 만일 ClientDataSet의 CommandText 속성을 변경하여 SQLQuery의 SQL 문장을 변경할 수 있도록 하려면 **`poAllowCommandText`** 옵션을 **true**로 설정합니다.

DataSetProvider의 BeforeUpdateRecord 이벤트 핸들러를 생성한 경우, 이벤트 핸들러가 데이터베이스가 갱신되기 전에 레코드에 있는 필드의 값을 변경할 수 있다면 **`poPropogateChanges`** 옵션을 **true**로 설정합니다. 그러면 프로바이더는 모든 변경 사항을 ClientDataSet로 돌려보내 메모리에 들어있는 레코드를 갱신합니다.

DataSetProvider에는 유용한 이벤트가 많지만, 가장 중요한 두 가지는 OnGetTableName와 BeforeUpdateRecord입니다. 여러 테이블의 조인, 스토어드 프로시저 또는 읽기 전용 뷰에서 리턴된 행은 데이터베이스 애플리케이션에서 직접 편집할 수 없습니다.

DataSetProvider에는 이러한 상황을 처리할 수 있는 세 가지 도구가 있습니다. 스토어드 프로시저가 리턴한 레코드들처럼 레코드들이 단일 테이블의 필드를 포함하고

있는 경우, 유일한 문제는 DataSetProvider가 갱신해야 할 테이블의 이름을 알아낼 방법이 없다는 것입니다. 이에 대한 해법은 테이블의 이름을 리턴하는 DataSetProvider의 OnGetTableName 이벤트 핸들러를 생성하는 것입니다.

두 번째 가능성은 여러 테이블의 조인에서 단일 테이블의 필드가 갱신되어야 하는 경우입니다. 먼저, 갱신될 **필드들을 식별하도록** 개별 필드의 **ProviderFlags** 속성을 설정합니다. 다음으로 **테이블 이름**을 리턴할 **OnGetTableName** 이벤트 핸들러를 생성하면 프로바이더가 자동으로 SQL 문장을 생성합니다. 각 레코드에 대하여 여러 테이블을 갱신해야 한다면 **BeforeUpdateRecord** 이벤트 핸들러를 **DataSetProvider**에 추가합니다. 이벤트 핸들러에서는 각 테이블에 대한 SQL 문장을 발생시켜 실행할 수 있습니다.

BeforeUpdateRecord 이벤트 핸들러도 갱신하기 전에 각 레코드를 검사하고 필드의 값을 변경할 장소를 제공합니다. 예외를 발생시켜서 갱신을 전체적으로 봉쇄할 수도 있습니다. 이런 이유로 BeforeUpdateRecord는 비즈니스 룰을 적용할 수 있는 좋은 장소입니다.

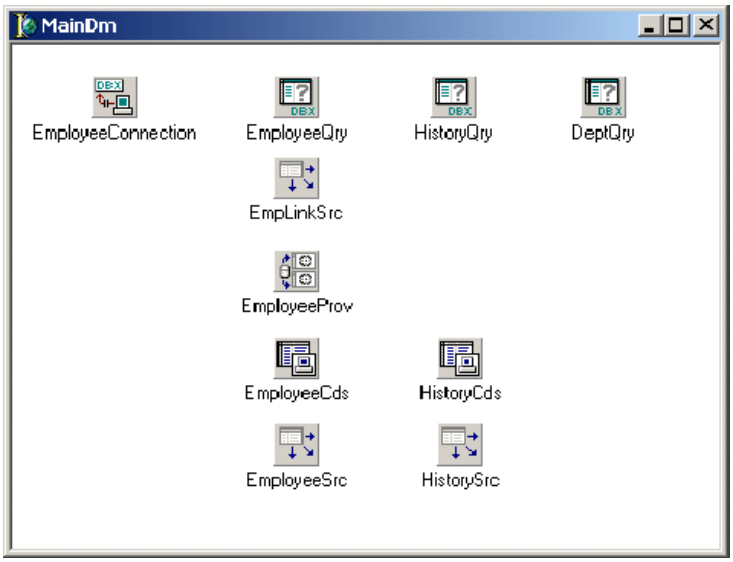
ClientDataSet

ClientDataSet는 ProviderName 속성을 통하여 DataSetProvider에 연결됩니다. 이것은 DataSetProvider로부터 데이터를 받아서 메모리에 데이터를 저장하고, 변경 사항을 기록하고, ClientDataSet ApplyUpdates 메소드가 호출되면 변경 사항을 DataSetProvider에 보냅니다.

다음과 같이 샘플 애플리케이션을 계속합니다.

1. 두 개의 **ClientDataSet**를 샘플 애플리케이션에 있는 **데이터 모듈**에 끌어다 놓습니다.
2. 첫번째 것의 **ProviderName** 속성을 **`EmployeeProv`**로 설정하고, Name 속성을 **`EmployeeCds`**로 설정합니다.
3. **EmployeeCds**를 더블 클릭하여 **Fields Editor**를 연 후에 **모든 필드 객체**를 추가합니다. Fields Editor에 있는 마지막 필드 객체의 이름은 HistoryQry입니다. 이것은 네스트된 데이터셋 필드로, 각 종업원 레코드에 대한 급여 이력 레코드가 들어 있습니다. HistoryQry 컴포넌트가 반환하는 레코드는 EmployeQry 결과 셋 안에 네스트된 데이터셋으로 표시됩니다. 이것은 SQLClientDataSets의 DataSource 속성의 HistoryQry이 EmployeeQry에 연결된 EmpLinkSrc로 설정되어 있으며 EmployeeQry 레코드로부터 SQL 문장에 대한 매개변수 값을 취하기 때문입니다.
4. **EmployeeCds**을 마우스 오른쪽으로 클릭한 후에 팝업 메뉴에서 **Fetch Params**를 선택하면 ClientDataSet가 매개변수 목록을 갱신하여 EmployeeQry 컴포넌트와 일치되도록 합니다.
5. 두번째 **ClientDataSet**를 **HistoryCds**로 설정하고, **DataSetField** 속성을 **EmployeeCdsHistoryQry**로 설정하면 EmployeeCds에 있는 네스트된 데이터셋 필드로부터 데이터를 읽어들입니다.
6. **HistoryCds**를 더블 클릭하여 **모든 필드**를 추가합니다.
7. 두 개의 **DataSource 컴포넌트**를 **데이터 모듈**에 떨어뜨린 후에 이름을 **EmployeeSrc**와 **HistorySrc**로 정합니다.
8. 이들의 **DataSet** 속성을 각각 **EmployeeCds**와 **HistoryCds**로 설정합니다.

이제 EmployeeCds와 HistoryCds 컴포넌트의 Active 속성을 true로 설정할 수 있을 것입니다. 데이터 모듈은 다음 그림과 비슷할 것입니다.



데이터 모듈은 이 그림과 비슷할 것입니다

네스트된 디테일 데이터셋은 마스터와 디테일 데이터셋에 대한 갱신 적용 순서가 혼동되지 않도록 하므로 대단히 강력합니다. 예를 들어, 하나 또는 그 이상의 디테일 레코드가 있는 새로운 마스터 레코드가 추가되는 경우, 마스터 레코드 INSERT는 디테일 레코드에 대한 INSERT 문장보다 앞서서 처리되어야 합니다. 그러나, 마스터에 대한 디테일 레코드를 삭제한 후에 마스터 레코드를 삭제하면, 디테일 레코드에 대한 DELETE 문장이 마스터에 대한 삭제에 앞서서 처리되어야 합니다. 네스트된 데이터셋을 사용하면 DataSetProvider는 데이터베이스 서버에서의 오류를 방지하기 위하여 올바른 순서로 갱신이 처리되도록 보장합니다.

ClientDataSet에는 몇 가지 유용한 속성이 있습니다. Aggregates 속성은 데이터셋에 있는 숫자 필드의 합, 최소, 최대, 개수 또는 평균을 자동으로 계산하는 집계 필드를 제공합니다. 집계는 색인으로 그룹화할 수 있습니다. AggregatesActive 속성을 통해 런타임에 집계 계산을 활성화하거나 비활성화 할 수 있습니다. 세미콜론으로 구분된 필드 이름 목록을 IndexFieldNames 속성에 할당하면 해당 필드별로 오름차순으로 ClientDataSet에 있는 레코드를 정렬합니다. 내림차순으로 정렬하거나 매우 큰 데이터셋에서 더 빠른 성능을 얻으려면 정렬하고자 하는 필드에 색인을 만들어서 색인 이름을 ClientDataSet의 IndexName 속성에 할당합니다.

CommandText 속성을 사용하면 DataSetProvider에 데이터를 제공하는 컴포넌트 내의 SQL 문장을 변경할 수 있습니다. **ClientDataSet**을 닫고 **새로운 SQL 문장**을 **CommandText**에 할당한 후에, **ClientDataSet**를 열어서 새로운 쿼리가 반환한 레코드를 보십시오. ClientDataSet의 Params 속성을 사용하여 소스 데이터셋의 SQL 문장에 있는 매개변수에 새 값을 할당할 수도 있습니다. 소스 데이터셋의 속성을 직접 액세스하는 대신 ClientDataSet의 CommandText와 Params 속성을 사용해야 하는 이유는 무엇일까요? 이것은 장래에 프로그램을 멀티티어 볼랜드 DataSnap 애플리케이션으로 전환하는 경우 코드를 변경하지 않아도 되는 장점이 있기 때문입니다.

PacketRecords 속성은 프로바이더가 서버에서 한 번에 가져오는 레코드의 수를 제어합니다. 기본값인 –1은 소스 데이터셋의 SQL 문장이 반환한 레코드를 프로바이더가 모두 가져오는 것입니다. 일반적으로 이것을 문제가 되지 않지만, 레코드 수가 매우 많은 경우에는 레코드를 나누어서 가져올 필요가 있습니다.

BDE와 dbExpress의 provide/resolve구조의 가장 큰 차이점 가운데 하나는, 변경 로그에 있는 변경 사항을 데이터베이스에 적용하려면 ClientDataSet의 ApplyUpdates 메소드를 호출해야 한다는 것입니다. 변경 로그에 적용되지 않은 변경 사항이 있는지 여부를 알려면 ChangeCount 속성을 사용합니다. ApplyUpdates는 한 개의 파라미터를 가집니다. 이 매개변수는 오류의 갯수인데, 허용치를 초과하였을 경우 프로세스를 롤백하고 종료됩니다. 일반적으로 0으로 설정함으로써 오류가 발생하자마자 갱신 프로세스가 멈추도록 합니다. –1로 설정하면 발생된 오류의 수에 관계없이 변경 로그에 있는 변경 사항을 DataSetProvider가 모두 적용합니다.

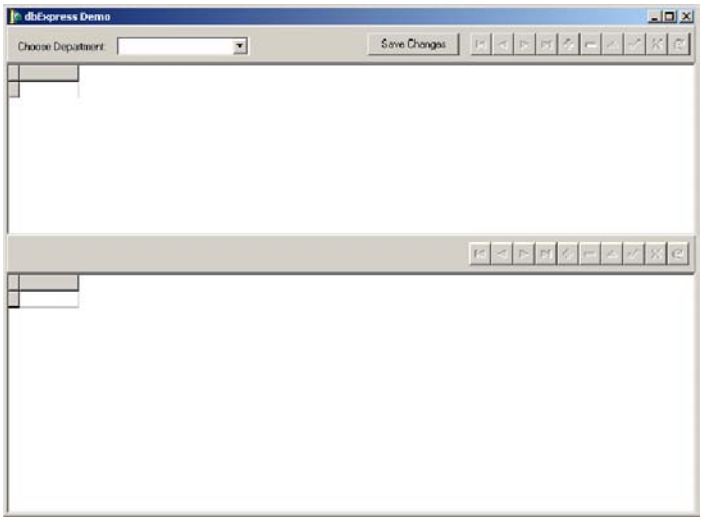
갱신 오류가 발생하면 ClientDataSet의 OnReconcileError 이벤트가 발생합니다. 데모 애플리케이션에서 갱신 오류를 처리하려면 다음 절차를 따릅니다.

1. Object Repository의 **Dialogs** 페이지로부터 **Reconcile Error Dialog**를 프로젝트에 추가합니다.
2. 데이터 모듈의 **uses 절**에 **Reconcile Error Dialog** 유닛을 추가합니다.
3. **Reconcile Error Dialog**가 자동으로 생성되지 않도록 확인합니다.
4. EmployeeCds에 대한 **OnReconcile** 오류 이벤트 핸들러를 생성한 후에 다음 코드를 추가합니다.

```
Action := HandleReconcileError(
      DataSet, UpdateKind, E);
```

업데이트를 적용할 때에 오류가 발생하면 Reconcile Error Dialog가 오류 메시지와 오류를 발생시킨 레코드 및 유저가 처리 방법을 선택할 수 있는 버튼들을 보여줍니다. 샘플 애플리케이션을 마치려면 다음 절차를 따릅니다.

1. **Panel, DBGrids, DBNavigators**를 각각 두개씩 메인 **폼**에 추가하여 아래의 그림과 같이 구성합니다.



2. **데이터 모듈** 유닛을 폼 유닛의 **uses 절**에 추가합니다.
3. 위의 DBGrid와 DBNavigator의 **DataSource** 속성을 **EmployeeSrc DataSource 컴포넌트**로 설정합니다.
4. 아래의 DBGrid와 DBNavigator의 **DataSource** 속성을 **HistorySrc DataSource 컴포넌트**로 설정합니다.
5. **Label** 컴포넌트와 **ComboBox** 컴포넌트를 위의 Panel에 추가합니다.
6. ComboBox에 **OnChange** 이벤트 핸들러를 생성하고 아래의 코드를 추가합니다.

```
procedure TMainForm.DeptComboChange(Sender:
  TObject);
begin
  with MainDm.EmployeeCds do
  begin
    if Active then CheckBrowseMode;
    Close;
    Params.ParamByName('DEPT_NO').AsString
      := Copy(DeptCombo.Text, 1, 3);
    Open;
  end; //with
end;
```

7. **Button** 컴포넌트를 위의 **Panel**에 추가한 후에 캡션을 **Save Changes**로 설정합니다.
8. 다음과 같은 코드를 **Button**의 **OnClick**의 이벤트 핸들러에 추가합니다.

```
procedure TMainForm.SaveBtnClick(Sender:
  TObject);
begin
  with MainDm do
  begin
    if EmployeeCds.ChangeCount > 0 then
    begin
    if HistoryCds.Active then
      HistoryCds.CheckBrowseMode;
    if EmployeeCds.Active then
      EmployeeCds.CheckBrowseMode;
    EmployeeCds.ApplyUpdates(0);
    EmployeeCds.Refresh;
    end; //if
  end; //with
end;
```

9. department ComboBox를 채울 **메소드**를 추가합니다.

```
procedure TMainForm.LoadDeptCombo;
begin
  with MainDm do
  begin
    DeptQry.Open;
    while not DeptQry.Eof do
    begin
      DeptCombo.Items.Add(
        DeptQryDept_No.AsString + ' ' +
        DeptQryDepartment.AsString);
      DeptQry.Next;
    end; //while
    DeptQry.Close;
  end; //with
end;
```

10. 폼의 **OnCreate** 이벤트 핸들러를 생성하여 다음과 같은 코드를 추가합니다.

```
procedure TMainForm.FormCreate(Sender:
  TObject);
begin
  MainDm.EmployeeCds.Open;
  LoadDeptCombo;
end;
```

BDE와 dbExpress의 비교

아래의 표는 BDE와 dbExpress를 몇가지 핵심적인 영역에서 비교한 것입니다. 이를 통해 전체적인 차이점을 알 수 있습니다. 두번째 표는 각 BDE 컴포넌트에 해당하는 dbExpress 컴포넌트를 보여줍니다.

기능	BDE	DbExpress
레코드 버퍼링	BDE가 로컬 메모리에서 보관 할 레코드의 수를 결정합니다.	개발자가 로컬에 보관할 레코드의 수를 결정합니다.
네트워크 트래픽 제어	BDE가 서버로부터 가져올 레코드의 수를 결정합니다.	사용자가 서버에서 가져올 레코드의 수와 시기를 결정합니다.
트랜잭션 제어	수동 및 자동 트랜잭션 제어가 가능합니다. 트랜잭션은 사용자가 데이터를 편집하는 동안 활성화됩니다.	수동 및 자동 트랜잭션 제어가 가능합니다. 트랜잭션은 갱신이 적용되는 짧은 순간에만 활성화됩니다.
배포	배포 크기가 큽니다. (약 18MB) 설치와 구성이 복잡하며, 레지스트리 변경이 필요합니다.	총 0.5 MB 미만의 DLL 2개만 필요하며, 사용자의 EXE에 컴파일되어 포함될 수 있습니다. dbxconnections.ini와 dbxdrivers.ini 파일을 배포하지 않는 한 레지스트리를 변경할 필요가 없습니다.
크로스 플랫폼	윈도우	윈도우, 리눅스
써드파티 드라이버	개발이 어렵습니다. 실제로 존재하는 것이 거의 없습니다.	개발이 쉬우며 많이 존재합니다. 써드파티 드라이버에 관한 정보는 Borland Developer Network 웹 사이트 bdn.borland.com를 참조하십시오.

아래는 각 BDE 컴포넌트에 해당하는 dbExpress 컴포넌트들을 보여주는 표입니다. dbExpress에는 BDE BatchMove 컴포넌트에 해당하는 것이 없습니다.

BDE	DbExpress
TDatabase	TSQLConnection
TQuery	TSQLQuery
TStoredProc	TSQLStoredProc
TTable	TSQLTable
No equivalent	TSQLDataSet
TBatchMove	No equivalent
SQL Monitor Utility	TSQLMonitor
Session	N/A
UpdateSQL	N/A
NestedDataSet	N/A
BDEClientDataSet	SimpleDataSet

세션의 개념은 dbExpress에 존재하지 않기 때문에 BDE 세션 컴포넌트에 대응하는 것이 필요없습니다. BDE UpdateSQL 컴포넌트는 BDE의 Query 컴포넌트의 캐시 업데이트 기능과 함께만 사용됩니다. dbExpress의 경우에는 ClientDataSet 컴포넌트의 기능이 이 기능을 대신합니다. 네스트된 데이터셋을 처리하는 기능은 dbExpress의 DataSetProvider와 ClientDataSet 컴포넌트에 내장되어 있습니다.

BDE SQL Links 애플리케이션을 dbExpress로 마이그레이션하기

BDE와 SQL Links 드라이버를 사용하는 애플리케이션을 dbExpress로 전환하려면 다음의 절차가 필요합니다. 대부분의 클라이언트/서버 애플리케이션은 BDE의 Table 컴포넌트를 사용하지 않습니다. 로컬 데이터베이스 애플리케이션은 BDE의 Table 컴포넌트를 대단히 많이 사용하므로, 뒤에 나오는 "로컬 데이터베이스 애플리케이션을 dbExpress로 마이그레이션하기"에서 다룹니다.

- **BDE Database** 컴포넌트를 **SQLConnection 컴포넌트**로 바꿉니다.
- **BDE Query**와 **StoredProc** 컴포넌트를 **SQLQuery** 및 **SQLStoredProc** 컴포넌트로 바꿉니다.
- 양방향 스크롤링과 편집이 필요한 각 SQLQuery와 SQLStoredProc에 대하여 **DataSetProvider**와 **ClientDataSet**를 추가합니다.

BDE Database를 SQLConnection으로 바꾸기

각 BDE 데이터베이스 컴포넌트를 SQLConnection 컴포넌트로 바꾸어야 합니다.

연결 매개변수 설정하기

애플리케이션을 변경하지 않고 데이터베이스 연결을 변경하기 위하여 BDE 알리아스를 사용하는 경우에는 dbExpress 연결 이름을 사용하거나 연결 파라미터를 INI 파일에 저장해야 합니다. 가장 유연성 있고 강력한 방법은 연결 파라미터를 가진 애플리케이션 자신만의 INI 파일을 이용하는 것입니다. 이 방법을 선택하면 INI 파일을 읽기 위한 코드를 애플리케이션에 추가하고 시작할 때에 SQLConnection 컴포넌트의 Params 속성을 설정해야 합니다. INI 파일에 있는 연결 파라미터를 편집하기 위한 코드를 애플리케이션에 추가할 수도 있습니다.

트랜잭션의 처리

애플리케이션을 변환하기 전에 해야 할 중요한 결정 가운데 하나는 트랜잭션 제어를 어떻게 제어할 것인가 입니다. BDE 애플리케이션에서 트랜잭션 제어를 BDE가 자동으로 처리하도록 한 경우에는 변경 사항을 데이터에 저장하고자 하는 곳마다 ApplyUpdates의 호출을 추가하기만 하면 됩니다. 이렇게 하면 dbExpress는 BDE처럼 트랜잭션을 처리해줄 것입니다. 갱신 후에도 같은 레코드 셋으로 작업을 계속할 경우에는 ClientDataSet Refresh 메소드를 호출해야 합니다. 호출하면 소스 쿼리를 재실행하고, 새로운 결과 셋을 ClientDataSet에 읽어들입니다. 그러면 그 사이에 다른 사람이 실행한 변경 사항을 볼 수 있습니다.

BDE 애플리케이션이 StartTransaction, Commit 및 Rollback을 명시적으로 호출하는 경우에는 선택이 더 복잡해집니다. 한 가지 선택 방법은 코드와 관련된 트랜잭션을 모두 제거하고 Commit에 대한 각 호출을 해당 ClientDataSet의 ApplyUpdates 메소드에 대한 호출로 대체하는 것입니다. 그러면 dbExpress가 트랜잭션을 대신 처리해 주고 코드를 간단하게 합니다. 이 방법을 실행할 수 없는 단 한 가지 경우는 두 개 이상의 ClientDataSet의 갱신을 하나의 트랜잭션에 결합해야 하는 경우입니다. 명시적인 Rollback 호출은 ClientDataSet CancelUpdates 메소드에 대한 호출로 대체해야 합니다. CancelUpdates은 ClientDataset의 변경 로그에 남아 있는 변경 사항을 취소시킵니다.

다른 선택 방법은 명시적 트랜잭션 제어 문장을 유지하는 것입니다. 의외로, 이것은 다음의 세 가지 이유로 인해서 가장 어려운 방법입니다.

- 각 Commit 전에 ApplyUpdates에 대한 호출을 추가해야 합니다.
- dbExpress의 provide/resolve 구조가 제공하는 짧은 트랜잭션 시간의 장점을 이용하려면 모든 StartTransaction에 대한 호출의 위치를 변경해야 합니다.
- TTransactionDesc 파라미터를 전달하기 위해서 모든 StartTransaction, Commit 및 Rollback에 대한 호출을 변경해야 합니다.

보시다시피 명시적 트랜잭션을 유지하는 것보다 제거하는 것이 더 간단합니다. 이 세 단계를 자세히 검토하겠습니다. 트랜잭션을 시작하고 Commit에 대한 각 호출을 하기 전에 트랜잭션에 참여하는 각 ClientDataSet의 ApplyUpdates 메소드를 호출해서 ClientDataSet 변경 로그에 있는 변경 사항들이 데이터베이스에 적용되도록 해야 합니다. ApplyUpdates를 호출하기 전에는 데이터베이스가 변경되지 않으므로 변경사항을 적용 할 것이 없습니다.

BDE 클라이언트/서버 애플리케이션의 경우 트랜잭션 제어 프로세스는 다음과 같습니다.

1. **트랜잭션**을 시작합니다.
2. 사용자가 데이터를 **편집**하도록 합니다.
3. **트랜잭션**을 적용하거나 취소합니다.

이 모델에서 트랜잭션은 사용자가 데이터를 편집하는 동안 계속 활성화 상태에 있습니다. dbExpress provide/resolve 구조의 주요 장점 가운데 하나가 짧은 트랜잭션 시간 입니다. 이 장점을 이용하려면 트랜잭션 제어 프로세스가 다음 절차를 따라야 합니다. 그래야 사용자가 변경 사항을 만드는 동안 트랜잭션이 활성화되지 않습니다.

1. 사용자가 ClientDataset의 데이터를 편집하도록 합니다.
2. 트랜잭션을 시작합니다.
3. ApplyUpdates를 호출합니다.
4. 트랜잭션을 적용하거나 취소합니다.

프로그램에서 트랜잭션 제어 프로세스를 dbExpress 모델로 변경하려면 **StartTransaction**에 대한 각 호출을 각각에 해당하는 **Commit**에 대한 호출의 바로 앞으로 **이동**해야 합니다. 그런 후, **ApplyUpdates**에 대한 호출을 StartTransaction에 대한 호출과 Commit에 대한 호출 사이에 추가합니다.

BDE Database 컴포넌트와 dbExpress SQLConnection 컴포넌트는 모두 StartTransaction, Commit 및 Rollback 메소드가 있습니다. BDE와는 달리, dbExpress는 데이터베이스가 지원하는 한도 내에서 동시에 여러 트랜잭션이 활성화되는 것을 허용합니다. 다수의 트랜잭션을 지원하려면 SQLConnection StartTransaction, Commit 및 Rollback 메소드가 TTransactionDesc 타입의 파라미터을 가져야 합니다. TTransactionDesc는 다음과 같이 선언됩니다.

```
TTransactionDesc = packed record
  TransactionID : LongWord;
  GlobalID : LongWord;
  IsolationLevel :
    TTransIsolationLevel;
  CustomIsolation : LongWord;
end;
```

각각의 동시 트랜잭션에 대하여 TTransactionDesc 형식의 변수를 선언하고 TransactionId를 현재 활성화되어 있는 트랜잭션 가운데 고유한 번호로 설정해야 합니다. GlobalId 필드는 오직 Oracle 데이터베이스 트랜잭션과 함께만 사용됩니다. IsolationLevel은 `xilDIRTYREAD`이나 `xilREADCOMMITTED`, 또는 `xilREPEATABLEREAD`로 설정되어야 합니다. CustomIsolation 필드는 현재 사용되지 않습니다.

BDE은 동시에 활성화된 여러 트랜잭션을 지원하지 않습니다. 그러므로 트랜잭션 제어 문장을 포함하는 각 유닛의 uses 절에 있는 유닛의 인터페이스 섹션에서 TTransactionDesc 형식의 하나의 변수만 선언하면 됩니다. 애플리케이션의 시작 코드에서 **TransactionId** 필드를 **one**으로 설정하고, **IsolationLevel**를 **xilREADCOMMITTED** 또는 **xilREPEATABLEREAD**으로 설정합니다. 마지막으로, **StartTransaction, Commit** 및 **Rollback**에 대한 모든 **호출**을 변경하여 TTransactionDesc 변수를 파라미터로 전달하도록 합니다.

모든 DataSet 컴포넌트 바꾸기

데이터셋 컴포넌트를 모두 바꾸는 것은 변환 과정에서 가장 큰 작업입니다. 각 BDE Query 컴포넌트를 SQLQuery 컴포넌트로 바꾸어야 하고, 각 BDE StoredProc 컴포넌트를 SQLStoredProc 컴포넌트로 바꾸어야 합니다. BDE 컴포넌트에는 dbExpress 컴포넌트에는 없는 속성이나 이벤트가 있기 때문에 이 과정은 복잡합니다. 코드에 존재하지 않는 속성에 대한 참조가 있으면 모두 제거해야 합니다.

양방향 액세스 또는 레코드 편집 기능이 필요한 각 SQLQuery 및 SQLStoredProc 컴포넌트에 대하여 DataSetProvider와 ClientDataSet를 추가해야 합니다. SQLQuery와 SQLStoredProc에 없는 이벤트에 대한 이벤트 핸들러를 ClientDataSet의 해당 이벤트에 연결합니다. 기존 코드에 대한 변경을 최소화하려면 ClientDataSet 컴포넌트의 이름을 바꾸려는 Bolrand Database Engine Query 또는 BDE StoredProc 컴포넌트와 같이 정하는 것을 고려할 수 있습니다.

SQLQuery 및 **SQLStoredProc 컴포넌트**로 되돌아가서 Fields Editor를 사용하여 필드 객체를 생성합니다. 필드 객체를 생성한 후에는 기본키에 있는 각 필드에 대하여 **pfInKey** 프로바이더 플래그를 **true**로 설정합니다. 갱신하지 않아야 할 계산 필드가 있는 경우에는 **pfInUpdate** 및 **pfInWhere** 프로바이더 플래그를 **false**로 설정합니다.

DataSetProvider와 ClientDataSet를 추가할 필요가 없는 경우도 있을 수 있습니다. 예를 들어, 콤보박스나 목록 박스의 Items 속성의 필드 값 가운데 하나를 추가하기 위해서 결과 세트를 첫 행부터 마지막까지 한번 읽은 쿼리가 있다고 가정합시다. 이런 경우, 레코드를 한 방향으로만 스캔하고 읽기 전용 액세스만을 필요로 하기 때문에 SQLQuery 또는 SQLStoredProc 컴포넌트를 직접 사용할 수 있습니다.

BDE 애플리케이션에서 캐시 업데이트를 사용하는 경우에는 **BDE Query 컴포넌트**를 **SQLQuery 컴포넌트**로 변환할 때 **UpdateSQL 컴포넌트**를 제거합니다. ClientDataSet와 DataSetProvider는 같은 기능과 장점을 가지고 있으므로 dbExpress에서는 캐시 업데이트가 필요 없습니다.

마지막으로, 앞에서 설명한 바와 같이 **ClientDataSet의 ApplyUpdates** 메소드에 대한 호출을 추가하여 트랜잭션 제어에 대해 ClientDataSet 로컬 버퍼의 데이터에 대한 변경 사항이 데이터베이스에 기록되도록 할 것을 잊지 말아야 합니다.

데이터 타입 매핑

dbExpress는 BDE에서 작업할 때에 접하지 못했던 두 가지 새로운 데이터 타입을 사용합니다. 배정도 부동 소수점 값에 적합하지 않은 모든 숫자 값은 TFMTBCDField 객체에 들어가서 애플리케이션으로 반환됩니다. TFMTBCDField 객체는 값을 32자리 최대 정밀도의 true BCD 값인 TBCD 타입으로 저장합니다. TFMTBCDField의 값으로 수식을 계산하려면 **AsVariant** 속성을 사용하여 **값**을 v**ariant 변수**로 저장합니다.

DbExpress는 날짜-시간 값을 반환하기 위해 TSQLTimeStampField 필드 객체 타입를 사용합니다. TSQLTimeStampField는 날짜-시간 정보를 TSQLTimeStamp 타입의 변수에 저장합니다. TSQLTimeStamp는 연, 월, 일, 시, 분, 초 및 그 이하에 대해 각각의 필드를 가진 Delphi 레코드(C++의 구조체)입니다. 그 이하란 1천분의 1초의 단위를 말합니다. TSQLTimeStamp는 정밀도의 손실 없이 날짜-시간 값이 저장되도록 합니다. TSQLTimeStampField에는 날짜-시간 값을 다른 데이터 타입으로 바꾸기 위한 As… 속성이 있습니다.

BatchMove 컴포넌트 바꾸기

dbExpress에는 BDE의 BatchMove 컴포넌트에 해당하는 것이 없습니다. 애플리케이션에서 BatchMove 컴포넌트를 사용하는 경우에는 이를 코드로 바꾸어야 합니다.

로컬 데이터베이스 애플리케이션을 dbExpress로 마이그레이션하기

Paradox나 dBase와 같은 로컬 데이터베이스를 사용하는 애플리케이션을 dbExpress로 마이그레이션하는 경우에는 몇 가지 추가로 처리해야 할 문제가 있습니다.

데이터 변환

dbExpress는 Paradox나 dBase 테이블을 지원하지 않으므로 dbExpress가 지원하는 SQL 데이터베이스 서버로 변경하고 데이터를 새로운 데이터베이스로 옮겨야 합니다. 이것을 위해서는 다음과 같은 세 가지 방법이 있습니다.

- Delphi와 C++Builder에 포함되어 있는 BDE Data Pump 유틸리티를 사용합니다.
- 써드파티 변환 유틸리티를 사용합니다. 일부 데이터베이스에는 데이터 임포트 또는 변환 유틸리티가 포함되어 있습니다. 또한, 인터넷에서 무료 또는 저렴한 가격에 사용할 수 있는 변환 유틸리티를 찾을 수도 있습니다.
- 데이터 변환 프로그램을 직접 작성합니다.

변환한 SQL 데이터베이스가 로컬 데이터베이스에 있는 모든 데이터 종류를 지원하지 않을 수 있습니다. 예를 들어, Paradox에는 Boolean 데이터 타입이 있지만, 모든 SQL 데이터베이스 서버가 그런 것은 아닙니다. 변환 유틸리티를 이용하려는 경우에는 새로운 데이터베이스가 직접 지원하지 않는 데이터 타입을 확인하고 변환 툴이 이러한 타입을 어떻게 처리하는지 결정해야 합니다.

대부분의 SQL 데이터베이스 서버는 CHAR와 VARCHAR 데이터 타입을 모두 지원하지만, 로컬 데이터베이스는 단일 문자열 타입을 가지고 있습니다. 여러분의 변환 툴이 문자열 필드를 여러분의 데이터베이스 서버에 있는 올바른 데이터 타입으로 변환하는지 확인해야 합니다.

데이터 타입을 변환하려는 이유는 여러 가지가 있을 수 있습니다. 예를 들어, Paradox 테이블은 모든 실수를 배정도 부동 소수점 형식으로 변환하는 반면, 대부분의 SQL

데이터베이스 서버는 NUMERIC이나 DECIMAL과 같은 고정 소수점 형식을 지원합니다. 부동 소수점과는 달리, 고정 소수점 형식은 모든 소수점 이하의 수를 정확하게 표현할 수 있고, 따라서 화폐 단위나 기타 소수점 이하의 수가 중요한 데이터를 저장하는 경우 훨씬 좋은 선택입니다. 여기에서도 사용하려는 변환 툴이 이러한 종류의 변환을 처리할 수 있는지 확인해야 합니다.

서버 보안

정확한 사용자 이름과 비밀번호를 입력하지 않으면 SQL 데이터베이스 서버에서 데이터베이스를 열 수 없습니다. 애플리케이션의 모든 사용자가 데이터베이스 서버에서 같은 사용자 이름과 비밀번호를 사용하거나, 각 사용자가 각각의 계정을 가지도록 할 수도 있습니다. 보안을 어떻게 처리하든 애플리케이션에 코드를 추가해야 합니다.

대부분의 로컬 데이터베이스에서는 승인되지 않은 접근을 방지하기 위해서 데이터베이스를 암호화할 수 있으나, 데이터베이스 서버에서는 그렇지 않은 경우가 많습니다. 암호화 하지 않는 대신, 승인되지 않은 접근으로부터 데이터베이스 파일을 보호하기 위해서 운영 시스템의 보안 기능에 의존합니다. 일반적으로 물리적인 데이터베이스 파일에 대한 접근이 필요한 사용자는 데이터베이스 관리자와 데이터베이스 서버 서비스를 사용하는 계정뿐입니다. 만일 애플리케이션이 판매원이 들고 다니는 노트북 컴퓨터와 같은 독립 시스템에 설치된 경우에는 필요로 하는 보안 방식을 데이터베이스가 지원하는지 여부를 확인해야 합니다.

셋 구조

전형적인 로컬 데이터베이스 애플리케이션은 BDE Table 컴포넌트를 사용하여 전체 테이블을 열고 살펴볼 수 있도록 합니다. 이 방식은 SQL 데이터베이스 서버에서는 전체 테이블을 데이터베이스 서버로부터 네트워크를 거쳐 각 워크스테이션으로 가져와야 하므로 성능이 저하될 수 있습니다. 테이블 컴포넌트도 메타데이터를 가져오면서 많은 네트워크와 서버 대역을 사용하므로 서버에서 레코드를 셀렉트하고 갱신하도록 SQL 문장을 구성할 수 있습니다.

클라이어트/서버 애플리케이션이 올바로 디자인되어 있다면 ClientDataSet에서 메모리에 레코드를 보유하는 것은 애플리케이션이 서버로부터 한번에 레코드를 조금씩 가져오기 때문에 문제 되지 않습니다. 그러나, 사용자가 큰 테이블을 볼 수 있는 로컬 데이터베이스 애플리케이션을 변환하는 것은 전체 테이블이 ClientDataSet에 의해서 메모리에 저장되기 때문에 클라이언트에 상당한 메모리를 소모하게 합니다.

BDE Table 컴포넌트를 사용하는 애플리케이션을 변환하는 경우, 다음과 같은 세 가지 방법이 있습니다.

1. 각 BDE 컴포넌트를 SQLTable, DataSetProvider 및 ClientDataSet로 바꿉니다.
2. 각 BDE 컴포넌트를 SQLQuery, DataSetProvider 및 ClientDataSet로 바꾸고, 테이블에서 모든 열과 행을 셀렉트합니다.
3. 각 BDE 컴포넌트를 SQLQuery, DataSetProvider 및 ClientDataSet로 바꾸고, 애플리케이션을 클라이언트/서버 셋 지향으로 변환합니다.

첫째 방법은 SQLTable 컴포넌트의 속성들이 BDE Table 컴포넌트의 속성들과 가장 비슷하므로 가장 쉽습니다. 이 경우 테이블이 너무 크지 않다면 성능은 좋습니다. 이 방법의 단점은 나중에 데이터셋의 일부 또는 전부를 변환하여 더 작은 셋의 레코드를 선택하도록 하는 경우 SQLTable 컴포넌트를 SQLQuery 컴포넌트로 바꾸어야 하는 것입니다.

둘째 방법은 더 나은 선택입니다. 행과 열을 모두 선택해야 하기는 하지만 애플리케이션은 쿼리에 기반을 둘 수 있습니다. 나중에 더욱 셋 지향적인 디자인으로 바꾸려면 데이터 액세스 컴포넌트에서 SQL 문장을 바꾸면 됩니다.

셋째 방법은 코드를 많이 변경해야 합니다. 클라이언트/서버 환경을 위해서 디자인된 애플리케이션은 일반적으로 데이터를 표시하기 전에 사용자가 선택 범위를 입력해야 합니다. 선택 범위는 사용자가 작업할 약간의 레코드를 가져오도록 쿼리의 WHERE 절에서 사용됩니다. 일단 사용자가 한 셋의 레코드에서 작업을 끝낸 후에는

다른 셋의 선택 범위를 입력하여 다음 세트의 레코드를 가져옵니다. 애플리케이션에서 전체 테이블을 살펴볼 수 있도록 되어있다면 사용자가 선택 범위를 입력할 수 있도록 코드와 폼을 추가해야 하고 사용자가 제공한 값을 이용하기 위해 WHERE 절을 변경해야 합니다.

요약

애플리케이션을 BDE에서 dbExpress로 마이그레이션하면 다음과 같은 많은 이점이 있습니다.

- 더 짧은 트랜잭션
- 네트워트 트래픽과 시스템 리소스 사용에 대한 제어권 확대
- 용량 및 시스템 리소스 사용 감소
- 성능 향상
- 더 용이한 배포
- 더 작아진 배포 패키지
- 크로스 플랫폼 지원

dbExpress는 BDE 데이터셋 컴포넌트의 속성, 메소드 및 이벤트와 유사하게 디자인된 데이터셋 컴포넌트들을 포함하고 있으므로 변환하는데 드는 노력을 최소화 해줍니다. 따라서, 기존의 코드의 변경을 최소화하면서도 더 뛰어난 기술로 옮겨갈 수 있습니다.

저자 소개

Bill Todd는 Phoenix 인근에 있는 데이터베이스 컨설팅 및 개발 회사인 The Database Group, Inc.의 사장입니다. 그는 4권의 데이터베이스 프로그래밍 서적과 90개 이상의 기사 공동 저자이며, Team Borland의 회원으로서 볼랜드 인터넷 뉴스그룹에 기술 지원을 제공하고 있습니다. 그는 미국 및 유럽의 Borland Developer Conferences에서 24개 이상의 글을 발표하였습니다. 빌은 전국적으로 잘 알려진 강사로서, 미국과 해외에 걸쳐 Delphi와 InterBase 프로그래밍을 강의하였습니다. 빌의 연락처는 bill@dbginc.com입니다.

Borland®

100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com | 831-431-1000