멀티-tier 데이터베이스 어플리케이션의 제작

이번 장에서는 대규모 데이터베이스 어플리케이션을 작성할 수 있는 멀티-tier 데이터베이스 어플리케이션에 대해서 알아본다. 기본적으로 멀티-tier 데이터베이스 어플리케이션은 클라이언트에서 동작하며 데이터를 조작하기 위한 인터페이스를 가지고 있는 클라이언트 프로그램과 실제 DBMS 를 조작하기 위한 리모트 데이터베이스 서버, 그리고 네트워크 상에 연결되어 클라이언트 프로그램의 데이터에 대한 처리를 수행하게 되는 어플리케이션 서버의 3 가지 구조로 나누어볼 수 있다.
이렇게 설명하는 모델이 3-tier 모델이다. 델파이 4 에서는 이러한 모델을 지원하기 위한 여러가지 컴포넌트를 제공한다. 델파이 4 에서 멀티-tier 모델을 구현하기 위해서는 사용하는 네트워크 프로토콜에 따라 크게 몇 가지로 세분할 수 있다. TCP/IP 의 기본적인 소켓을 사용해서 이를 구현할 수도 있고, MS 에서 제공하는 DCOM 을 사용할 수 있다. 또한, DCOM 의 트랜잭션과 리소스 문제를 해결하기 위해 제공되는 MTS 를 사용하거나, ORB 를 이용한 CORBA 를 사용할 수도 있다. 그 밖에도 Inprise 에서 제공되는 OLEnterprise 나 Entera 와 같은 미들 웨어를 사용할 수 도 있으며, AS400 용 제품을 이용하여 DB2 를 이용할 수도 있다.
이러한 여러가지 접속 방법은 기본적으로 MIDAS(Multi-tier Distributed Application Services Suite)라는 기술을 이용하게 된다.

멀티-tier 데이터베이스 어플리케이션의 필요성

무엇 때문에 개발자는 기존의 1-tier 나 2-tier 환경에 어떠한 점에 만족하지 못하고, 새로운 멀티-tier 환경의 어플리케이션을 만들게 되었는가 ? 그 이유를 알아보자.
가장 큰 장점으로는 일을 구분하여 쉽게 개발과 유지보수가 된다는 점을 들 수 있다.
클라이언트 어플리케이션 작성 시에는 사용자의 사용자 인터페이스와 데이터의 제시 방법에 대한 내용을 구축하는데 중심을 맞출 수 있으며, 일의 단위를 구분하기 편리하다. 그리고, 어플리케이션 서버의 작성 시에는 많은 클라이언트의 요구에 따라 비즈니스 규칙이 적용된 데이터를 전송하거나 클라이언트에서 변경된 자료를 역시 비즈니스 규칙이 적용된 데이터를 데이터베이스 서버에 전송하는 내용만 작성하면 된다.
이렇게 일을 구분하여 작업할 수 있으므로, 팀 단위의 프로젝트가 쉬운 것은 두말할 나위 없으며, 모든 클라이언트 프로그램이 적용될 비즈니스 규칙이 구현된 어플리케이션 서버의 변경작업만으로 클라이언트 프로그램을 재배포하거나 재작성할 필요가 없이 해결이 가능하다. 마찬가지로, 데이터베이스 서버의 변경 작업이 발생할 경우에는 해당 계층의 변경만으로 작업을 최소화할 수 있다.
그 이외의 장점으로는 다음과 같은 것들이 있다.

- Thin 클라이언트

사용자의 PC 에 설치된 클라이언트 프로그램은 1-tier 나 2-tier 어플리케이션에 비해 그 크기가 훨씬 작은 가벼운 프로그램으로 대치가 가능하다. 1-tier 나 2-tier 데이터베이스 어플리케이션에서는 해당되는 DBMS 에 접속하기 위한 무거운 해당 데이터베이스 드라이버와 유틸리티를 가지고 있어야 한다. 하지만 멀티-tier 모델에서는 DB 클라이언트.DLL 이라는 하나의 파일만 있으면 해결이 된다. 아마도 델파이로 개발된 프로그램을 만들어본 개발자라면 이 고통을 이해할 수 있을 것이다. 필자도 델파이 1 시절에 개발한 소프트웨어를 보면 간단한 2M 상당의 프로그램을 보급하기 위해 프로그램 디스켓 1 장, BDE 디스켓 2 장, 그 당시 사용하던 크리스탈 리포트 런타임 1 장, 이런 식으로 총 4 장의 디스켓이을 사용하였다. 그렇지만, 여기에서 BDE 에 해당되는 내용을 줄일 수 있다. 그리고, 클라이언트에서 작업하는 여러가지 비즈니스 규칙에 관련된 코드를 모두 어플리케이션 서버로 이관시킬 수 있기 때문에 클라이언트 어플리케이션 자체의 크기도 많이 줄일 수 있다.

- 분산 환경

멀티-tier 어플리케이션에서는 어플리케이션 서버로 불리는 중간 계층이 서로 다른 서버나 클라이언트에서 구동될 수 있기 때문에, 해당 작업의 처리를 위해서 여러 대로 분리되어 있는 PC 를 사용할 수 있기 때문에 작업의 효율성을 높일 수 있다.

- 보안

1-tier 나 2-tier 모델에서는 데이터베이스가 노출되거나 데이터베이스의 접속이 쉽게 노출되어 데이터의 보안에 취약했다. 이러한 보안 부분을 어플리케이션 서버가 담당하도록 할 수 있기 때문에, 사용자는 DBMS 의 위치를 알 수 없게 할 수 있다.

- 비용의 절감

많은 개발자들은 DBMS 의 역할과 해당 DBMS 의 사용자 수마다 해당되는 고비용에 대한 비용을 많이 줄일 수 있다. 능력 있는 프로그래머라면, 파라독스나 DBASE 로 구성된 1-tier 데이터베이스 어플리케이션를 응용하여 어플리케이션 서버에서 사용하거나 델파이에 기본적으로 내장되어 있는 로컬 인터베이스를 사용하여 시스템을 구성할 수 있다.

미들웨어

BDE 나 ODBC 도 일종의 미들웨어라고 볼 수 있다. 그 밖에도 TP-monitor 계열의 엔테라나 턱시도 등의 제품도 미들웨어라 불리운다.
미들웨어를 사용하는 이유는 여러가지가 있으나, 대표적인 것은 대규모 데이터베이스 시스템을 구축하는 경우, 그 규모가 MB 나 GB 단위가 아닌 TB 단위의 대용량 데이터베이스인 경우에 단순한 DBMS 로 이를 감당하기 어렵다.
예를 들어, 여러가지 다양한 DBMS 가 사용되는 경우를 들 수 있는데 대표적으로 IMF 를 맞이하여 은행들 간에 합병을 할 때, 두 은행에서 사용되는 DBMS 가 이 기종일 가능성이 높다. 이때 이 기종 DBMS 에 일어나는 트랜잭션을 모두 처리하거나, 여러 개의 DBMS 를 사용하고 저장 프로시저를 이용한 비즈니스 규칙의 관리 등의 데이터 흐름을 단일화 하는 것이 미들웨어의 역할이다.
BDE 나 ODBC 를 살펴보자, 개발자는 하나의 API 를 호출할 수 만 있게 프로그램을 제작한다면 해당 API 에서 지원하는 DBMS 에는 간단한 접속만 변경함으로써 비즈니스 규칙부분의 코딩을 변경할 필요가 없어지게 된다. 그 밖에도, 한 번에 수십, 수백 개의 트랜잭션이 발생하는 은행과 같은 대규모 사이트의 경우 같은 비즈니스 규칙을 처리하는 여러 개의 어플리케이션 서버가 있을 수 있으며, 시스템의 안정성을 위하여 어플리케이션 서버를 확장할 수도 있다. 이런 경우에 작업을 분배하여 움직이는 스케쥴링이 필요한데, 이렇게 재배치를 하는 작업을 로드 밸런싱(load balancing)이라고 하며, 이런 역할도 미들웨어가 담당한다.

MIDAS 의 기술적인 구조

MIDAS 를 사용하는 경우에 클라이언트는 thin 클라이언트를 지향한다. 클라이언트 PC 에는 BDE 가 필요 없고, DB 클라이언트.DLL 파일만 있으면 된다. MIDAS 는 DCOM, CORBA, TCP/IP 소켓, OLEnterprise 를 지원한다.
MIDAS 컴포넌트 세트는 다음의 4 가지 요소로 구성되어 있다.

컴포넌트	내 용
Remote data modules	COM 자동화 서버나 CORBA 서버를 만들기 위한 특별한 데이터 모듈이다. 이 데이터 모듈에는 TProvider, TSession, TDatabase, TQuery, TTable, TStoredProc, TBatchMove 그리고 TTimer, TimageList 등의 컴포넌트가 올라갈 수 있다. 이것은 듀얼 인터페이스 자동화 서버로 IDataBroker 의 인터페이스를 따르고 있다. 또한, 디자인 작업 시에도 동일한 접속환경을 보여준다.
Provider component	서버에 위치하는 컴포넌트로 TQuery, TTable 의 내용을 클라이언트에서 데이터 패킷 단위로 가져갈 수 있도록 인터페이스를 구성하는 컴포넌트
Client dataset component	DBCLIENT.DLL 을 사용하여 서버의 Provider 와 통신하여 데이터 패킷을 읽어 들일 수 있는 컴포넌트

Connection component	네트워크 상의 서버 위치를 찾고 서버와 연계하며 IProvider 인터페이스를 활성화하여 통신을 할 수 있도록 해주는 통신 프로토콜

3-tier 어플리케이션

델파이로 작성하는 기본적인 데이터베이스 어플리케이션은 1-tier 이거나 2-tier 어플리케이션이다. 하지만 대규모의 데이터베이스 어플리케이션을 작성하다 보면 2-tier 에서 저장 프로시저나 트리거보다 더 많은 일을 하는 비즈니스 규칙을 적용할 어플리케이션을 만들 필요가 있게 된다. 델파이를 이용하면 이런 비즈니스 규칙을 적용한 어플리케이션 서버를 중간에 여러 계층으로 둘 수 있는 멀티-tier 환경을 쉽게 구성할 수 있다.
보통 멀티-tier C/S 어플리케이션은 논리적인 단계로 구분되는 비즈니스 규칙을 통하여 각각의 컴퓨터 시스템상의 데이터 공유 및 분산 처리와 다른 객체와의 통신을 처리하는 것으로 나누어 볼 수 있다. 이러한 환경을 멀티-tier 환경이라고 부르는데 보통은 클라이언트, 어플리케이션 서버, DBMS 로 구분되는 3-tier 환경이 가장 일반적이다.
이런 형태의 3-tier 환경은 다음과 같은 기본적인 구성요소로 이루어져 있다.

1. 클라이언트 어플리케이션

일반적인 사용자들이 자료의 조회 및 수정, 삭제 등의 데이터에 대한 조작을 취할 수 있는 프로그램이다.

2. 어플리케이션 서버

비즈니스 규칙을 적용하는 부분으로 각각의 작업단위 별로 구분되는 작업으로 일반적인 DBMS 의 저장 프로시저 트리거보다 더 많은 작업과 복잡한 작업을 하는 경우가 많다. 보통은 이러한 어플리케이션 서버는 시스템의 부하가 커짐에 따라 여러 개가 존재할 수 있으며 이들의 분배 작업을 위한 미들웨어의 필요성도 커지게 되었다.

3. 원격 데이터베이스 서버

일반적으로 알고 있는 데이터베이스 서버로서 실제 데이터를 저장하고, 단순하고 반복적인 작업을 수행하는 저장 프로시저나 트리거로 정의된 기본 비즈니스 규칙이 적용되어 있다. DBMS 를 선정하는 기준에는 데이터의 규모도 중요하지만 이런 DBMS 가 작업하는 성능도 중요한 고려 사항이 된다.

이 중에서도 어플리케이션 서버가 가장 중요한 역할을 하게 된다. 어플리케이션 서버는 클

라이언트와 DBMS 간의 자료 흐름을 관장하며 중간 작업을 해준다. 실제 3-tier 서버 시스템의 구축이 잘 이루어진 곳에서는 이러한 비즈니스 규칙을 얼마나 잘 어플리케이션 서버에 적용시키고 분배하느냐에 따라 시스템의 성능이 좌우된다. 이러한 어플리케이션 서버를 데이터 브로커(Data Broker)라고 한다.
이때 경우에 따라서는 여러 층의 어플리케이션 서버가 필요할 수 있으며, 이렇게 되면 3-tier 를 넘어선 멀티-tier 어플리케이션이 되는 것이다. 최근의 추세로 본다면 LAN 상에서 동작하는 구조가 아닌 인터넷이나 원격지에서 저속의 모뎀으로 접속하는 시스템을 구성하는 경우도 많기 때문에 이런 경우에는 다양한 층(layer)으로 구성된 멀티-tier 어플리케이션이 필요할 수 있다. 이럴 때에는 트랜잭션을 처리할 수 있는 중간 계층을 하나 더 만들 수 있고, 하나의 DBMS 가 아닌 이 기종의 DBMS 와 접속하는 부분도 만들어 볼 수 있을 것이다.

기본적인 구조

사용자 에게는 2-tier 나 3-tier 나 아무런 차이가 없다. 오히려 3-tier 어플리케이션이 수행속도가 더 떨어질 지도 모른다. 그 이유는 소프트웨어에서 하나의 계층이 더 생긴다는 것은 그만큼의 데이터 전송 속도와 수행 속도를 떨어뜨리는 원인이 될 수 있기 때문이다. 예를 들어, 실제 클라이언트의 개수가 20 여대 미만인 경우에 MIDAS 를 쓴다면 이는 '닭 잡는데 소 잡는 칼을 쓴 꼴'이 될 것이다.
일반적인 2-tier 에서 사용하던 컴포넌트 들을 그대로 3-tier 에서도 사용해도 상관없다. 다만 TDataSet 에 연결되었던 내용 대신에 TClientDataSet 이라는 계층으로 바뀌는 차이가 가장 큰 차이이다. 그리고, 클라이언트에 위치하는 TRemoteServer 컴포넌트는 어플리케이션 서버의 Provider 와 통신을 하게 된다. Provider 클래스는 TProvider 로 선언되어 있다. 이 RemoteServer 컴포넌트를 사용하는 것은 클라이언트의 TClientDataSet 컴포넌트인데, 독특한 구조를 가지는 경우에는 개발자가 이런 클라이언트 데이터 세트와 Provider 를 재설계할 수도 있다.
TProvider 컴포넌트는 클라이언트의 데이터 요구를 받아서, 데이터 서버로부터 데이터를 얻어온 뒤에 전송할 내용을 모아서 TClientDataSet 컴포넌트에게 전송하며, 반대로 클라이언트에서 자료의 변경사항을 받게 되면 데이터베이스에 이를 적용시키는 일도 처리한다. 이때 에러가 발생하여 적용에 실패하면 그 내용을 다시 클라이언트에 전송하여 에러에 대응하도록 한다.
클라이언트와 어플리케이션 서버가 연결되면 서버의 Provider 과 클라이언트의 클라이언트 데이터 세트는 정해진 프로토콜에 따라 데이터를 교환한다. 이때 Provider 는 이미 알고 있는 데이터 접근 컴포넌트를 사용하여 자료를 DBMS 나 로컬 데이터베이스에서 읽어온다. 이러한 단계를 하나씩 정리해보자.

- 클라이언트가 어플리케이션 서버에 접속할 때 어플리케이션 서버가 동작 중이 아니라면

해당 서버를 구동시킨다 (이 작업은 각각의 프로토콜을 설정하는 부분에서 해당 서버를 등록하는 작업을 거친다). 그리고, 클라이언트는 해당 어플리케이션 서버로부터 Provider 의 인터페이스를 받아온다.

- 클라이언트는 필요한 자료를 어플리케이션 서버에 요청하고 이때에 자료는 모든 자료를 받거나 세션을 통하여 데이터의 일정 부분만 받아 올 수 있다. 일반적으로 전달 받는 자료의 크기를 조절하여 네트워크 트래픽이나 시스템의 수행속도를 높이는 것이 좋다.
- 어플리케이션 서버는 클라이언트의 요청을 받아 DBMS 에서 데이터를 데이터 접근 컴포넌트를 통하여 데이터를 요청한다. 데이터를 읽어내어 해당되는 비즈니스 규칙을 적용한 데이터를 패킷으로 포장하여 클라이언트에 전송한다.
- 클라이언트는 어플리케이션 서버에서 전송된 데이터를 사용자에게 보여줄 수 있도록 화면에 표시한다. 사용자는 화면에 나타난 데이터를 참고하여 새로운 데이터의 첨삭작업을 가하고 변경된 내용을 클라이언트의 log 에 저장한다. 클라이언트는 변경된 log 자료들을 변경시점에 맞추어서 서버에 전송하기도 하고 사용자의 요구에 따라 어플리케이션 서버에 전송하기도 한다.
- 어플리케이션 서버는 클라이언트에서 전송된 데이터를 데이터베이스 서버를 통하여 데이터의 변경작업을 취한다. 이 시점에서 해당 데이터(레코드)를 다른 사용자가 변경 작업을 하고 있지 않다면 이 자료는 어플리케이션 서버에 의해 데이터베이스 서버에 저장된다. 이 상황을 'resolving'이라고 부른다.
- 어플리케이션 서버가 해당 'resolving' 과정 중에 문제가 발생하여 정상적으로 처리하지 못하면, 이 데이터는 데이터베이스 서버를 통해 저장할 수 없다. 해당 데이터는 클라이언트에 오류 신호와 데이터를 돌려 보내, 다시 작업을 취하도록 한다. 클라이언트는 'resolving' 과정 중에 문제가 생긴 데이터를 다시 받아 재처리하게 되는데, 이러한 과정을 재조정(Reconcile)이라고 한다.
- 이러한 재조정 과정은 여러가지가 있지만, 보통은 반송되어온 데이터를 다시 어플리케이션 서버에 전송하여 재처리를 취하는 방법이 있을 것이고, 아예 이 데이터를 취소하여 버리고 다시 데이터를 만들도록 하는 방법이 있을 수 있다.

MIDAS 어플리케이션 만들기

그러면 실제로 간단한 예제 어플리케이션을 이용하여 사용방법을 익히도록 하자.

- MIDAS 클라이언트 어플리케이션의 구조

사용자들이 사용하는 클라이언트 어플리케이션은 일반적인 2-tier 프로그램과 거의 동일하다. 심지어는 TClientDataSet 을 사용함으로써 플랫-파일을 사용하는 1-tier 어플리케이션이 되기도 한다. 다만 차이가 있다면 IProvider 와 인터페이스하는 부분만 틀릴 뿐이다.

MIDAS 를 사용하여 접속하는 방법에는 다음과 같은 4 가지 방식이 있다.

컴포넌트	프로토콜
TDCOMComponent	DCOM
TSocketConnection	윈도우 소켓 (TCP/IP)
TOLEnterpriseConnection	OLEnterprise (RPCs)
TCorbaConnection	CORBA (IIOP)

● 어플리케이션 서버의 구조

어플리케이션 서버로 사용될 수 있는 원격 데이터 모듈에는 다음의 3 가지가 있다. 이들이 모두 IDataBroker 인터페이스를 사용한다.

1. TRemoteDataModule

듀얼 인터페이스 자동화 서버로 구성된 모듈로 DCOM, 소켓, OLEnterprise 를 사용하여 어플리케이션 서버와 연결한다. 보통 MTS 를 만들지 못할 경우에 이 데이터 모듈을 사용한다.

2. TMTSDataModule

이 모듈은 액티브 라이브러리(.DLL)를 사용하는 MTS 데이터 모듈이다. 이 모듈도 DCOM, 소켓, OLEnterprise 를 사용하여 어플리케이션 서버와 연결된다.

3. TCorbaDataModule

CORBA 서버를 사용할 때 이용되는 데이터 모듈이다.

델파이 4 에서 어플리케이션 서버를 제작하기 위해서는 델파이 3 부터 제공되던 DCOM 환경을 사용할 수도 있고, DCOM 의 환경을 분산환경의 이점을 살려 이용할 수 있도록 MTS 를 사용하는 방법, CORBA 를 적용하거나 TCP/IP 의 소켓을 사용하는 등의 여러가지 방법을 사용할 수 있다. 이중에서 CORBA 를 이용하면 윈도우 NT 를 사용해야 한다는 한계를 벗어나, 리눅스나 상용 유닉스에서 만들어진 어플리케이션 서버를 사용할 수 있다. 실제로 윈도우 NT 와 유닉스로 만들어진 프로그램의 성능을 비교해 보면, CORBA 를 사용하는 방법이 얼마나 효율적인지 알 수 있을 것이다.
그 이후에 필요한 클라이언트 어플리케이션을 작성한다. 물론, 클라이언트 어플리케이션을 먼저 작성해도 무방하지만, 작업의 편리함을 위해서는 어플리케이션 서버를 먼저 작성한 다

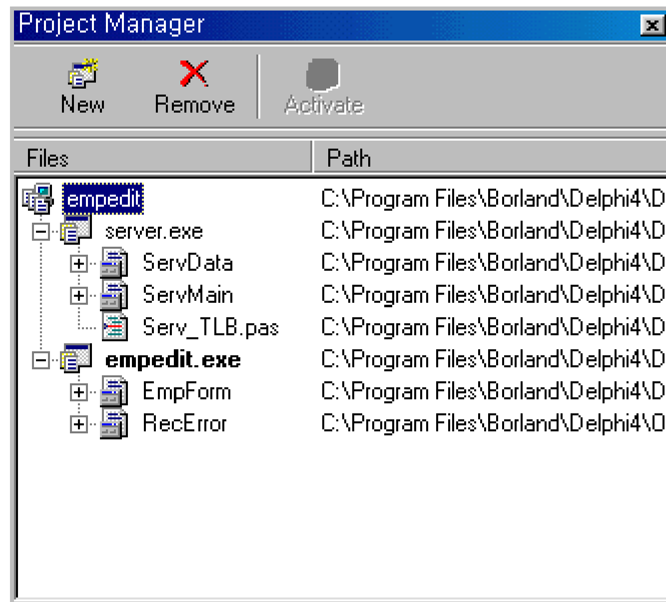
음에 클라이언트를 구성하는 것이 당연히 쉽다. 그래서 보통은 어플리케이션 서버를 클라이언트보다 먼저 작성하게 된다.

- 어플리케이션 서버의 제작

먼저 어플리케이션 서버의 구성에 대해서 살펴보자. 앞에서도 설명하였듯이 여러가지 방식의 어플리케이션 서버를 작성할 수 있다. 그러나, 개발자들은 이를 굳이 구별할 필요가 없다. 왜냐하면, 어떠한 개발방식을 사용하여도 개발하는 구조가 동일하기 때문이다. 다만, 연결되는 프로토콜만 설정하고 해당되는 컴포넌트만 교체하면 된다.
또한, 이러한 어플리케이션 서버의 작성 방법이 약간의 차이점을 제외하고는 일반적인 1-tier 나 2-tier 어플리케이션을 개발하는 방법과 거의 유사하다. 다만, 앞에서 설명한 Provider 인터페이스를 가져야 한다는 점이 다르다.
이러한 Provider 인터페이스를 지원하기 위해서는 Midas(델파이 3 에서는 Data Access) 컴포넌트 팔레트의 TProvider 컴포넌트를 사용하면 간단하게 해결된다. 이 Provider 인터페이스에 필요한 데이터 세트(TTable, TQuery)를 사용하여 데이터베이스에 접속하면 된다.
File|New 메뉴를 선택하고, 객체 저장소에서 Multitier 탭을 선택하면 사용가능한 여러 종류의 원격 데이터 모듈을 생성할 수 있는 위저드 들이 나타날 것이다. 일단 간단한 원격 데이터 모듈을 만들어 보도록 하자. 참고로 원격 데이터 모듈을 만들려면 일단 어플리케이션이 만들어져 있어야 한다. 이에 비해 CORBA 데이터 모듈이나 MTS 데이터 모듈은 단독으로 만들어 질 수 있다.
원격 데이터 모듈을 만들 수 있는 Remote Data Module 아이템을 더블 클릭하면, 다음과 같은 위저드가 실행될 것이다. 이렇게 해서 작성된 원격 데이터 모듈은 OLE 자동화를 통하여 구동되므로, 일반적인 데이터 모듈과는 큰 차이가 있다.

이제 해당 데이터 모듈에 원하는 TDataSet 용 컴포넌트인 TTable, TQuery, TStoredProc 등의 컴포넌트를 사용하여 단독 파일이나 DBMS 에 접속하고, 해당 TProvider 를 사용하면 된다. 또는, TProvider 컴포넌트를 사용하지 않고 해당 데이터 세트의 Provider 프로퍼티

를 사용하여도 무방하다. 단, 이 프로퍼티는 런타임에서만 사용할 수 있다.
TProvider 컴포넌트를 사용한다면 DataSet 프로퍼티에 원하는 데이터 세트를 연결하고, 원하는 이벤트에 코드를 추가한다. TProvider 컴포넌트에는 사용가능한 몇 가지의 이벤트가 있는데 이 이벤트에 대해서는 뒤에서 자세히 설명한다.
해당 어플리케이션 서버 프로젝트를 저장하고 컴파일한 다음 OLE 자동화 서버 정보를 등록해야 한다. 이 방법은 해당 레지스트리에 수동으로 저장하는 방법도 있고, 이 경우와 같이 out-of-process 서버로 작성되는 경우에는 OLE 자동화 서버를 실행하여 등록할 수도 있다. 여기에 대한 더 자세한 내용은 5 부의 내용을 참고하기 바란다. 이렇게 자동화 서버가 등록된 뒤에는 클라이언트에서 해당 서버를 호출할 때 자동적으로 서버가 구동된다.
이제 자세한 Provider 인터페이스를 작성하는 방법을 알아보자. 예제로는 Demos 디렉토리의 MIDAS 서브 디렉토리에 있는 Empedit 프로젝트를 이용한다. .bpg 파일을 로드하면 다음과 같은 화면이 나타날 것이다.



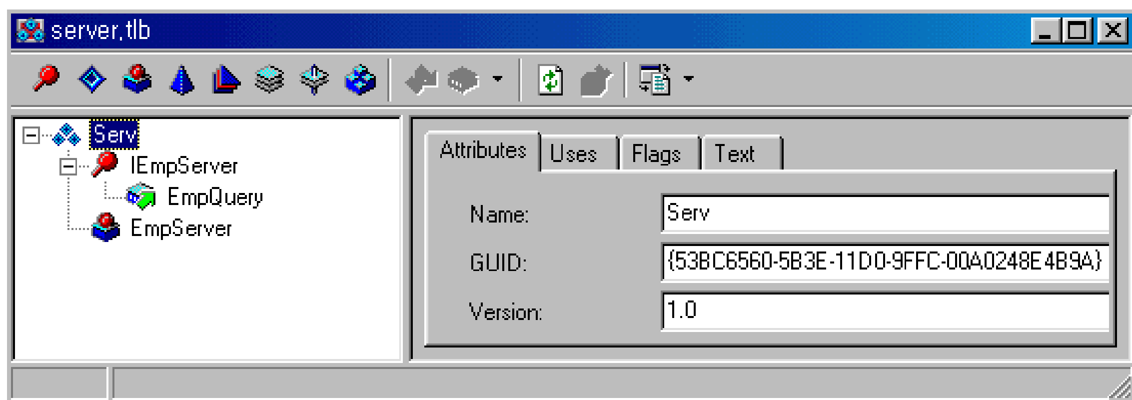
이중에 Serv_TLB.pas 유닛이 원격 데이터 모듈을 지원하기 위해 자동으로 생성된 파일로 OLE 자동화를 지원하기 위한 각종 내용이 생성되어 있다. 만들어진 클래스 선언부분을 살펴 보자.

```
IEmpServer = interface(IDataBroker)              //데이터 브로커가 선언되어 있다.
  ['{53BC6561-5B3E-11D0-9FFC-00A0248E4B9A}']
  function Get_EmpQuery: IProvider; safecall;              //SQL 을 Export 시키면 자동으로 생성
  property EmpQuery: IProvider read Get_EmpQuery;        //SQL 을 Export 시키면 자동으로 생성
end;
```

```
  IEmpServerDisp = dispinterface
    ['{53BC6561-5B3E-11D0-9FFC-00A0248E4B9A}']
    function GetProviderNames: OleVariant; dispid 22929905;
                                                      //SQL 을 Export 시키면 자동으로 생성
    property EmpQuery: IProvider readonly dispid 1;   //SQL 을 Export 시키면 자동으로 생성
end;

  CoEmpServer = class
    class function Create: IEmpServer;
    class function CreateRemote(const MachineName: string): IEmpServer;
end;
```

이제 데이터 세트를 클라이언트에서 접속하여 해당 프로토콜에서 원하는 데이터 세트에 접속할 수 있도록 provider 와 연결하여야 한다. 이때에 연결하는 방법은 간단하다. 데이터 모듈에 데이터 세트를 올려놓은 다음 해당 컴포넌트를 선택하고 마우스의 오른쪽 버튼을 클릭하면, 팝업 메뉴가 나타나는데 여기에서 'Export 데이터 세트 이름 from data module'이나 'Export 데이터 세트 이름 from data module' 메뉴를 선택하면 해당되는 데이터 세트 컴포넌트가 연결된다.
이때 변경되는 내용은 *.tlb 파일에 적용되는데, 다음 그림을 보면 쿼리 컴포넌트가 IEmpServer 인터페이스의 EmpQuery 프로퍼티로 등록된 것을 알 수 있다.



마찬가지로 *.tlb 파일을 생성시킨 ServerData.pas 유닛에도 자동적으로 다음과 같은 코드가 추가된다.

```
function TEmpServer.Get_EmpQuery: IProvider;
begin
```

```
  Result := EmpQuery.Provider;
end;
```

이밖에도 다음과 같은 코드도 추가된다.

```
initialization
 { This creates the class factory for us.  This code is generated automatically }
  TComponentFactory.Create(ComServer, TEmpServer, Class_EmpServer,
    ciMultiInstance);
end.
```

이 코드는 초기화 코드로 해당 폼의 초기화 작업을 할 때 수행된다. 소스 코드를 살펴보면 TComponentFactory 를 이용하여 OLE 자동화 객체를 생성해 주는 코드이다.
이와 같은 작업들이 단순한 export 작업 하나로 이루어진다. 이제 원하는 TDataSet 컴포넌트를 Provider 인터페이스에 포함시키는 작업이 끝났다.
그러면, 데이터 세트의 Provider 프로퍼티를 이용하지 말고 TProvider 컴포넌트를 이용하여 보자. 이 경우에는 해당 데이터 세트의 export 를 하는 것이 아니라 Provider 컴포넌트를 선택한 다음 마우스 오른쪽 버튼을 눌러 나오는 pop-up 메뉴에서 'Export Provider1 from data module'을 선택하면 해당 데이터 모듈의 Provider 역할을 개발자가 선택한 Provider 컴포넌트가 담당하게 된다.
참고로, 이런 방법으로 Provider 컴포넌트를 export 했을 때 *.tlb 파일에 추가되는 내용은 다음과 같다. 생성한 데이터 모듈의 이름은 Test 이다.

```
  ITest = interface(IDataBroker)
    ['{8648718F-5651-11D2-B28C-004F49002780}']
    function Get_Provider1: IProvider; safecall;
    property Provider1: IProvider read Get_Provider1;
  end;

  ITestDisp = dispinterface
    ['{8648718F-5651-11D2-B28C-004F49002780}']
    property Provider1: IProvider readonly dispid 1;
    function GetProviderNames: OleVariant; dispid 22929905;
  end;
```

해당 *.pas 유닛에 추가되는 내용은 다음과 같다.

```
function TTest.Get_Provider1: IProvider;
begin
  Result := Provider1.Provider;
end;
```

그리고, 해당 *.tlb 파일을 F12 를 눌러서 에디터를 살펴보면, 해당 프로퍼티로 Provider1 이 선언된 것을 확인할 수 있다. 이 *.tlb 파일은 OLE 자동화 객체의 선언 내용을 간편하게 설정할 수 있게 해준다. 이곳에서 원하는 프로퍼티나 메소드 등을 선언하고 수정할 수 있다. 자세한 내용은 OLE 자동화에 대해 다룬 장을 참고하자.

마찬가지로 컴파일 하고, 한번만 실행하면 해당되는 OLE 자동화 객체가 등록된다. 주의할 점은 한번 등록된 클래스 명으로는 다시 등록할 수 없다는 점이다. 그러므로, 테스트나 수행된 내용이 필요 없으면 UnRegister 하는 습관을 기르도록 한다.

이제 개발자는 OLE 자동화 객체를 사용하여 하나의 어플리케이션을 객체로 사용할 수 있게 되었다. 그리고, 원하는 데이터 세트 컴포넌트를 export 하여 OLE 자동화 객체의 메소드로 사용할 수 있으며 이 부분을 통하여 개발자는 원하는 데이터를 읽어 들이거나 조작할 수 있게 된다. 델파이에서는 이러한 일련의 과정을 통하여 델파이 특유의 VCL 컴포넌트들을 OLE 자동화 객체로 변환할 수 있게 해준다. 이러한 *.tlb 파일을 이용하면 MTS 나 CORBA 데이터 모듈에도 그대로 적용할 수 있다. 한번 만들어 놓은 소스로 다양한 프로토콜을 지원하는 어플리케이션 서버를 만들 수 있는 것이다.

- 클라이언트 어플리케이션의 작성

이제 클라이언트 어플리케이션을 작성해 보자. 앞에서 설명한데로 먼저 간단한 테스트용 어플리케이션 서버를 하나 작성하자. ServerForm 이라는 기본적인 폼과 ServerData 라는 원격 데이터 모듈을 선언한 다음 TTable 컴포넌트를 하나 올려놓고 DBDEMOS 앨리어스의 animals.dbf 테이블을 TableName 프로퍼티의 값으로 설정한 뒤에 export 시킨다. 그리고, 이 프로젝트를 Server.dpr 이라는 이름으로 저장한 뒤에 이를 컴파일 하고, 실행하여 해당 클래스를 등록하도록 한다.

이제부터 만들 클라이언트 어플리케이션은 개발자들이 많이 접했던 1-tier 나 2-tier 방식의 어플리케이션과 크게 다른 점이 없다. 다만 TRemoteServer 와 TClientDataSet 컴포넌트가 새롭게 쓰이게 된다. TRemoteServer 컴포넌트는 클라이언트와 어플리케이션 서버를 해당 프로토콜을 통하여 연결하여 준다. 그리고 TClientDataSet 컴포넌트는 일반적인 TTable, TQuery 등의 DataSet 컴포넌트 들을 대신하여 준다. 간단하게 TDatabase 컴포넌트와 TDataSet 컴포넌트를 두 개의 컴포넌트로 대체한다고 생각하면 비유가 조금 틀리지만 비슷한 내용이 될것이다.

이제 클라이언트를 만들어 보자.
New Application 메뉴를 선택하여 새로운 프로젝트와 데이터 모듈을 추가한다.
만들어진 DataModule 에 TRemoteServer 를 올려놓고 해당 프로퍼티중에 ServerName 을 클릭하면 등록된 서버가 나타날 것이다. 이때 만들어진 서버가 DCOM 을 사용한다면 간단하게 ComputerName 프로퍼티를 먼저 설정하면, 해당 컴퓨터에 설치된 서버들이 나타날 것이다. 이때 원하는 서버를 선택하면 된다. Server.ServerData 는 조금 전에 만들어진 Server.exe 의 원격 데이터 모듈인 ServerData 의 이름을 지칭한다. 선택되어지면 ServerGUID 는 자동으로 설정된다.
참고로 예제로 제공되는 클라이언트 어플리케이션의 경우에 컴퓨터 이름이 독자 들의 컴퓨터 이름과 다를 것이므로 소스를 분석할 때 이를 고려해서 서버가 설치된 컴퓨터 이름을 잘 지정하도록 한다.
그 다음에는 TClientDataSet 컴포넌트를 올려 놓고 RemoteServer 프로퍼티를 RemoteServer1 으로 지정한다. 그리고, ProviderName 을 설정하면 자동적으로 연결된 Server.exe 가 수행되면서 원격 데이터 모듈에 선언된 Table1 이 나타난다. 이 Table1 을 프로퍼티로 설정하면 TClientDataSet 컴포넌트와 Table1 이 연결된 것이다.
이제 이렇게 만들어진 TClientDataSet 을 사용하여 일반 어플리케이션을 제작하는 방법과 동일한 과정으로 제작에 임하면 된다.
여기에서 TRemoterServer 컴포넌트는 서버로 제작된 OLE 자동화 객체의 Provider 에 연결할 수 있도록 해주는 중간 역할을 한다. 엄밀히 말해 해당 Provider 의 DataSet 컴포넌트와 TClientDataSet 컴포넌트를 연결한다.
참고로, TRemoteServer 컴포넌트는 델파이 3 와의 호환성을 위해 남겨진 컴포넌트로 델파이 4 에서는 사용하고자 하는 프르토콜을 직접 지정하여 컴포넌트를 사용해야 한다. 지금과 같이 DCOM 을 사용한다면 TDCOMConnection 컴포넌트를 사용하여 똑 같은 방법으로 데이터 모듈을 연결하여 사용하면 된다. 앞에서도 여러 차례 언급한 TCORBAConnection, TSocketConnection 등도 비슷한 역할을 한다.
MIDASConnection 을 사용하면 앞에서 설명한 3 가지 프로토콜을 혼용할 수 있다. 다시 말해 DCOM, OLEnterprise 와 소켓 접속을 ConnectType 의 변환에 따라서 간단하게 조절할 수 있다. 그렇다면, OLEnterprise 와 SimpleObjectBroker 는 무엇을 하는 것일까? SimpleObjectBroker 에는 LoadBalanced 라는 프로퍼티가 있다. SimpleObjectBroker 에서는 서버에 등록한 서버의 어플리케이션 서버를 사용하게 해준다. 여러 어플리케이션 서버에 대한 로드 밸런싱을 수행하는 것이 그 역할이며, 여러 개의 어플리케이션 서버를 두고 클라이언트의 요구를 분산시켜 구동시는 것이다. 이렇게 하기 위해서는 접속 방법으로 TOLEnterpriseConnection 을 사용하는 것이 좋다. 또는 접속 방법을 마음대로 바꿀 수 있는 TMIDASConnection 컴포넌트를 이용할 수도 있겠다.
DCOM 이나 CORBA 등을 지원할 수 없는 환경에서는 TCP/IP 를 사용하여 소켓 접속을 통해서 접속이 가능하다. 이럴 때에는 TSocketConnection 컴포넌트를 이용한다.

● 멀티-tier 어플리케이션 제작에서 주의할 점

보통 어플리케이션 서버는 로컬 시스템이 아닌 원격 시스템일 경우가 많다. 이 경우에는 해당 시스템의 이름을 지정해 주는데 디자인 타임에서 설정해도 좋고 런타임에 설정하여도 무방하다. 하지만 DCOM 을 이용할 경우에는 시스템 레지스트리에 등록되어 있는 경우에 시스템의 이름을 지정하지 않아도 동작한다. 그 이유는 델파이에서 기본적으로 지원하는 어플리케이션 서버는 OLE 자동화 객체이기 때문이다.
ServerName 프로퍼티에는 어플리케이션 서버의 '이름.exe'에서 이름으로 지정된 것을 사용한다. 이 이름은 OLE 자동화 객체로 사용되는 이름과 동일하다.
보통 해당 접속 컴포넌트의 ServerName 과 ComputerName 을 설정하고 TClientDataSet 컴포넌트의 RemoterServer 와 ProviderName 프로퍼티를 설정하면 Active 프로퍼티가 자동으로 True 가 되지만, 기본적으로 ServerName 과 ComputerName 만 설정하면 데이터를 이용할 수 있다. 이때에 해당 어플리케이션 서버는 이미 동작을 시작하게 되는데, 이는 IProvider 인터페이스를 통하여 GetProvider 메소드가 수행되고 원하는 ProviderName 이 전달되기 때문이다.
Provider 는 OLE 자동화 객체와 데이터 세트 컴포넌트를 연결하기 위한 고리로서 OLE 자동화 객체의 메소드로 쓰일 수 있게 해주는 방법이다. 더구나 Provider 는 기본적으로 1 대 다의 클라이언트가 연결될 수 있는 구조를 가지고 있다는 점이 강점이다. 클라이언트에서 1 개의 어플리케이션 서버가 아닌 몇 개의 어플리케이션 서버의 연결이 필요할 경우에는 해당 되는 수 만큼의 접속 컴포넌트를 올려놓고 이를 사용하면 된다.
이제 필요한 접속을 설정하고 해당 어플리케이션 서버의 Provider 를 연결하여 사용하는 방법을 알아 보았다. 이러한 통신은 어플리케이션 서버와의 접속이 끊기기 전까지 이루는데, 접속 전에 할 일이 있으면 접속 컴포넌트의 BeforeConnect 나 AfterConnect 등의 이벤트를 활용하면 된다.
이제 원하는 작업을 클라이언트에서 수행할 수 있는 프로그램을 만들면 된다. 참고로, 중간에 접속을 중단하는 방법은 Connected 프로퍼티를 False 로 변경하기만 하면 된다. 그 밖에도 ServerName 프로퍼티를 변경하거나 클라이언트 프로그램을 종료하면 자동적으로 어플리케이션 서버는 동작을 멈추고 종료한다.

TClientDataSet 컴포넌트가 다른 데이터 세트 컴포넌트와 다른 점

멀티-tier 모델을 이용하여 클라이언트 어플리케이션을 작성할 경우에 TClientDataSet 컴포넌트를 사용한다는 점을 이미 언급한 바 있다. 그렇다면, TClientDataSet 컴포넌트는 일반적인 데이터 세트와 어떤 점이 같고 어떤 점이 다른가 ? 실제 클라이언트 데이터 세트는 델파이 3 에도 존재하였던 일종의 MemoryDataSet 과 거의 유사하다. 이것은 원격 컴퓨터

에 떨어져 있는 어플리케이션 서버의 데이터 세트의 내용을 클라이언트에 있는 것처럼 일반적인 데이터 인식 컴포넌트를 속임으로써 동작한다.
VCL 의 계승도를 봐도 TClientDataSet 컴포넌트가 TDataSet 에서 파생된 클래스임을 알 수 있다. 그러므로, TClientDataSet 컴포넌트는 일반 테이블에서 지원하는 대다수의 기능을 모두 처리하며, 이를 모두 메모리에서 수행하므로 더욱 빠르게 동작한다.
TClientDataSet 컴포넌트의 다른 용도는 플랫-파일 단위의 1-tier 어플리케이션을 작성할 수 있도록 해준다. 이렇게 하려면, TClientDataSet 컴포넌트를 마우스로 선택한 다음 오른쪽 버튼을 클릭한 뒤에 나타나는 팝업 메뉴에서 Assign Local Data 메뉴를 선택하면 해당 테이블이나 쿼리의 테이블 하나를 몽땅 메모리에 적재하고, 이 내용을 플랫 파일로 저장할 수 있다. 그리고, 해당 플랫 파일을 로드하여 BDE 가 없이 DBCLIENT.DLL 파일만으로도 데이터 어플리케이션을 작성할 수 있게 되었다. 물론, 메모리에서 이루어지는 작업이기 때문에 메모리가 적은 시스템에서는 곤란한 경우를 만날 수도 있을 것이다.
이러한 모델을 서류가방 모델(Briefcase model)이라 부르는데, 이 기능을 사용하여 필요할 때에 서버로부터 자료를 받아와서 클라이언트에 저장한 다음 클라이언트에서 작업을 수행하고 나중에 필요한 시기에 데이터를 전송하는 방식을 취할 수 있다. POS 와 같은 시스템에 쓰기에 적당한 방법으로, 특히 네트워크의 성능이 불량하거나 속도가 느린 경우에 유용하게 사용할 수 있는 방법이다.
일반적으로 TClientDataSet 컴포넌트의 사용 방법은 일반 데이터 세트의 CachedUpdates 메소드를 사용하는 것과 유사하다. 특히, 어플리케이션 서버로부터 데이터를 메모리로 가져오고 변경된 자료가 있을 경우에 해당 자료를 어플리케이션 서버로 보내어 변경된 내용을 데이터베이스에 적용하는 과정은 해당 데이터베이스와 동기화를 취한다는 점에서 비슷하다.
다만, OLE 자동화나 CORBA, 소켓 등을 사용한다는 점이 다를 뿐이다.

클라이언트 데이터 세트의 레코드 조작

클라이언트에서 어플리케이션 서버에 접속하면 필요한 자료를 전송 받아 작업하게 된다.
이때 사용자가 보는 데이터는 실제 어플리케이션 서버에 존재하는 데이터가 아닌 복제된 자료들이다. 그러므로, 사용자가 데이터 조작을 취하면 클라이언트 데이터 세트의 Delta 프로퍼티에 변경된 내용이 기록되며, 이후에 ApplyUpdates 메소드를 호출할 경우 Delta 프로퍼티에 저장된 데이터가 어플리케이션 서버로 전송되어 데이터 조작을 취하게 된다.
클라이언트에서 TClientDataSet 컴포넌트의 메소드인 ApplyUpdates 를 호출하면 연결된 Provider 에 Delta 에 저장된 데이터 조작 내용을 전송한다. 어플리케이션 서버의 Provider 는 데이터를 전송 받아 자신의 ApplyUpdates 메소드를 수행하고 데이터베이스에 기록한다. 이때에 데이터의 처리는 해당 레코드 단위로 이루어지는데 그 이유는 데이터베이스에 저장하고 조작하는 동안에 오류가 발생하면 해당 레코드 단위의 내용을 되돌려주기 위해서다.

Provider 가 변경된 내용을 데이터베이스에 기록한 다음에는 오류가 발생한 레코드를 클라이언트에 되돌려준다. 이때에 클라이언트는 반송된 자료를 재처리하기 위한 작업을 수행하는데, 변경된 내용을 레코드 단위로 처리하면서 문제가 발생된 내용을 Result 프로퍼티를 통하여 어플리케이션 서버로 전달한다.

이제 클라이언트 데이터 세트와 캐쉬 업데이트가 유사하다고 이야기한 부분이 이해가 되리라 생각한다. ApplyUpdates 메소드가 필요한 시점에서 데이터를 전송함으로써 데이터의 조작이 이루어지므로 사용자에게 원하는 시점에 변경된 내용을 수정할 수 있게 해주거나 자동으로 ApplyUpdates 가 일어나게 해주어야 한다.

그리고, ApplyUpdates 는 MaxErrors 라는 파라미터를 통해서 데이터 조작 시에 일어난 해당 에러의 갯수가 이보다 작을 경우 동작을 계속 수행하게 해주며, MaxErrors 이상의 에러가 발생하면 그 자리에서 수행을 멈춘다. 이 갯수는 에러가 발생하여 변경하지 못한 레코드의 갯수를 의미한다.

해당되는 레코드의 값은 OnReconcileError 이벤트를 통하여 알 수 있다. 이 이벤트는 에러가 발생할 때마다 한번씩 수행되므로 에러에 대한 조작이나 반응을 이 이벤트에 넣어놓으면 에러를 처리할 수 있다.

에러가 발생한 레코드의 처리

문제없이 자료가 전송되었다면 좋지만, 멀티-tier 환경에서 가장 믿을 수 없는 것이 네트워크 환경이다. 시시때때로 변화하며 부족한 대역폭을 같고, 혹시나 패킷을 잃어버리기라도 한다면 에러의 발생은 당근(!)이다.

앞에서 설명한대로 OnReconcileError 이벤트를 통하여 일차적으로 재조정 작업을 시도하며, 자동적으로 수행하는 이벤트이다. 문제는 이 이벤트를 작성할 때 주의할 점인데 이 이벤트 내에서는 레코드의 위치를 변경할 수 있는 다른 어떠한 이벤트도 수행하면 안된다는 점이다. 왜냐하면, 다음을 보자.

```
procedure TDataModule1.ClientDataSet1ReconcileError(
  DataSet: TClientDataSet; E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction);
```

이 코드를 보면, 해당 데이터 세트와 에러 내용, 수정/추가/삭제의 변경방법, 그리고 에러에 반응하는 raSkip, raAbort, raMerge, raCorrect, raCancel, raRefresh 와 같이 지정되는 Action 부분이 리턴되어 온다.

그러므로, 잘못하면 무한루프에 빠질 수 있다. 실제 이 이벤트의 역할은 에러가 발생한 레코드의 처리를 어떻게 할 것이냐 ? 라는 점이다. 실제 Action 에 들어갈 수 있는 값은 다음과 같다.

코 드	내 용
raSkip	내용을 건너뛰고 Delta 자료는 남겨두어 다음 ApplyUpdates 에 다시 한번 처리할 수 있도록 한다.
raAbort	재조정을 하지 않는다. 모든 처리를 중단하고 변경작업을 중단한다.
raMerge	변경내용을 서버의 레코드 내용과 합병한다.
raCorrect	변경을 다시 한번 시도한다.
raCancel	변경 사항의 Delta 자료를 삭제한다.

이때 참고할 내용은 UpdateKind 파라미터에 담겨있는데 시도된 데이터가 어떠한 상태인가 하는 점이다. ukModify, ukInsert, ukDelete 의 3 가지 상태는 수정, 추가, 삭제의 3 가지 형태를 말한다. 물론, E 파라미터에는 해당 에러메시지를 전송받는다.
개발자는 이러한 점을 참고로 하여 Action 파라미터에 원하는 값을 넣으면 된다.
그리고, 이 이벤트에서 해당 레코드의 OldValue, NewValue, CurValue 의 프로퍼티를 참고하면 해당 에러의 수정에 도움이 될 것이다.
에러메시지를 보는 방법을 잠깐 살펴보자. 이 에러 메시지는 일반적인 에러 메시지 처리법과 동일하다. E.message 는 메시지의 내용, EReconcileError(E).ErrorCode 하면 어떤 에러가 발생하였는지 알 수 있다.

```
with ReconcileError(E) do
begin
  if ErrorCode = DBIERR_KEYVIOL then
    Action := raSkip            //Key violation 이 발생하였다면 레코드는 그냥 건너 뛰어라
  else
    Action := raAbort           //나머지는 수행중지
end;
```

플랫-파일 다루기

TClientDataSet 컴포넌트를 이용하여 플랫 파일을 다루는 방법은 앞에서도 잠시 설명하였지만, 그 사용 용도가 무궁무진하다. 이 방법은 대입된 로컬 데이터를 Delta 프로퍼티에 몽땅(!) 저장하여 사용하는 방법으로 SaveToFile 과 LoadFromFile 메소드를 통하여 자료를 저장하고 읽어들일 수 있다.
이렇게 하기 위해서는 다음과 같은 프로퍼티와 메소드에 대해서 잘 알아야 한다.

프로퍼티/메소드	내 용

FetchOnDemand	필요할 때에 자동적으로 데이터를 가져올 수 있다. 이 값이 True 면 자동적으로 데이터를 읽어오고, False 인 경우에는 GetNextPacket 메소드를 사용하여 수동으로 읽어 와야 한다.
PacketRecords	한번에 몇 개의 레코드를 가져올 것인가를 결정한다. 보통 –1 를 값으로 설정하여 전체 데이터를 읽어온다.
GetNextPacket	설정된 PacketRecords 수 만큼의 다음 레코드를 읽으라는 메소드

보통은 PacketRecords 프로퍼티가 적당한 값을 가진다. DBGrid 나 StringGrid 를 사용할 경우에 적당한 데이터를 읽어 올 수 있도록 한번에 읽어올 자료의 크기를 지정하는 것이 좋다. PacketRecords 프로퍼티의 값이 0 보다 크다면 GetNextPacket 메소드를 통하여 해당 자료의 크기만큼 자료가 전송되어 오며, 0 일 경우에는 해당 데이터의 정보를 다시 읽어 온다. 그 밖의 일반적인 동작은 일반 데이터 세트와 동일하다.
그 밖에 TClientDataSet 컴포는트의 또다른 중요한 기능은 데이터를 처리하면서 Delta 데이터에 대한 Log 를 관리할 수 있게 해주는 것이다. 이 Log 는 LogChange 프로퍼티를 True 로 설정하는 것으로 이용할 수 있는데, 이 경우 수정/삽입/변경의 모든 내용의 히스토리를 저장한다. 그리고, 이렇게 만들어진 히스토리를 사용하여 변경 내용을 취소할 수 있다. 그 방법은 다음의 3 가지 방법이 있다.

1. 한단계 취소:

UndoLastChange 메소드를 사용하여 바로 전단계의 자료로 돌아가는 방법이다. 이때에 FollowChane 라는 프로퍼티가 True 이면 하면 undo 된 자료의 위치로 이동하고, False 이면 이동하지 않는다. 이 작업이 성공하면 True, 실패하면 False 가 반환된다.

2. 한꺼번에 취소:

히스토리는 하나의 레코드에 대한 여러 번에 걸친 변경 자료를 모두 가지고 있다. 이때에 RevertRecord 메소드를 사용하면 선택된 레코드의 모든 변경을 취소하고 원래 내용으로 돌려놓는다.

3. 전체 레코드의 변경 내용 취소:

변경된 내용을 전부 되돌려 놓고자 할 경우에는 CancelUpdates 메소드를 사용하면, 모든 변경사항이 원상복귀 된다.

- 플랫-파일 상태에서의 데이터 저장

플랫-파일 상태에서는 ApplyUpdates 메소드를 사용하여 데이터를 저장하지 않는다. 이 경우에는 변경된 자료가 히스토리인 로그 자료에 쌓여 있으므로, Data 프로퍼티는 원래의 자료를 고스란히 가지고 있다. 이때에 MergeChangeLog 메소드를 수행하면 로그 자료를 Delta 프로퍼티에 적용하여 저장할 수 있도록 해준다.

● 실제 클라이언트 데이터 세트의 저장

실제 클라이언트 데이터 세트의 데이터를 저장하려면 SaveToFile 메소드를 통하여 하나의 파일에 저장하고, 실제 해당 파일이 존재한다면 그 파일을 덮어서 다시 쓰게 된다.

정 리 (Summary)

이번 장에서는 멀티-tier 데이터베이스 어플리케이션을 제작하는 방법에 대해서 알아보았다. 멀티-tier 데이터베이스 어플리케이션은 대부분의 컴퓨터가 네트워크에 물리게 되는 환경이 되면 그 중요성이 점점 커질 것이다. 그러므로, 이번 장에서 다룬 내용에 대해서 잘 익혀 둘 필요가 있다.
다음 장부터 이어지는 제 4 부에서는 델파이의 가장 커다란 장점이라고 할 수 있는 컴포넌트 개발에 대한 내용을 다루게 된다.