

DataSnap in Action!

Bob Swart

January 2011

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

DATA SNAP IN ACTION!

Last year, a RAD in Action technical paper was published about DataSnap as supported in Delphi 2010 Enterprise and higher (see the link at the end of this paper). That particular paper provides a good introduction into the basics of DataSnap and multi-tier development. However, there have been a number of significant improvements and enhancements. Rather than rewriting the original white paper, it felt more useful to describe how to use DataSnap in a practical way, illustrating existing but especially the new and enhanced functionality along the way.

The new DataSnap XE functionality that will be covered in this new white paper, include the following features:

- DataSnap Server Wizards
- HTTPS support for ISAPI DLLs
- Encryption filters (when you don't use HTTPS)
- Enhanced TAuthenticationManager component
- Role-based Authentication capabilities for server methods
- Role-based Authentication for TDataSetProvider
- TDSSessionManager
- Deployment using HTTPS/SSL (on IIS)

I will also cover features and functionality that were present already in the previous version of DataSnap, but which were not covered in the white paper (although they are covered in my more detailed DataSnap Development courseware manual – see the link at the end of this paper). These topics include:

- How to “hide” Server Module methods from being exposed
- Autoincrement key fields
- Master-detail relationships using DBX and DataSnap
- Passing parameter values from client to server
- Assigning parameter values at the server side

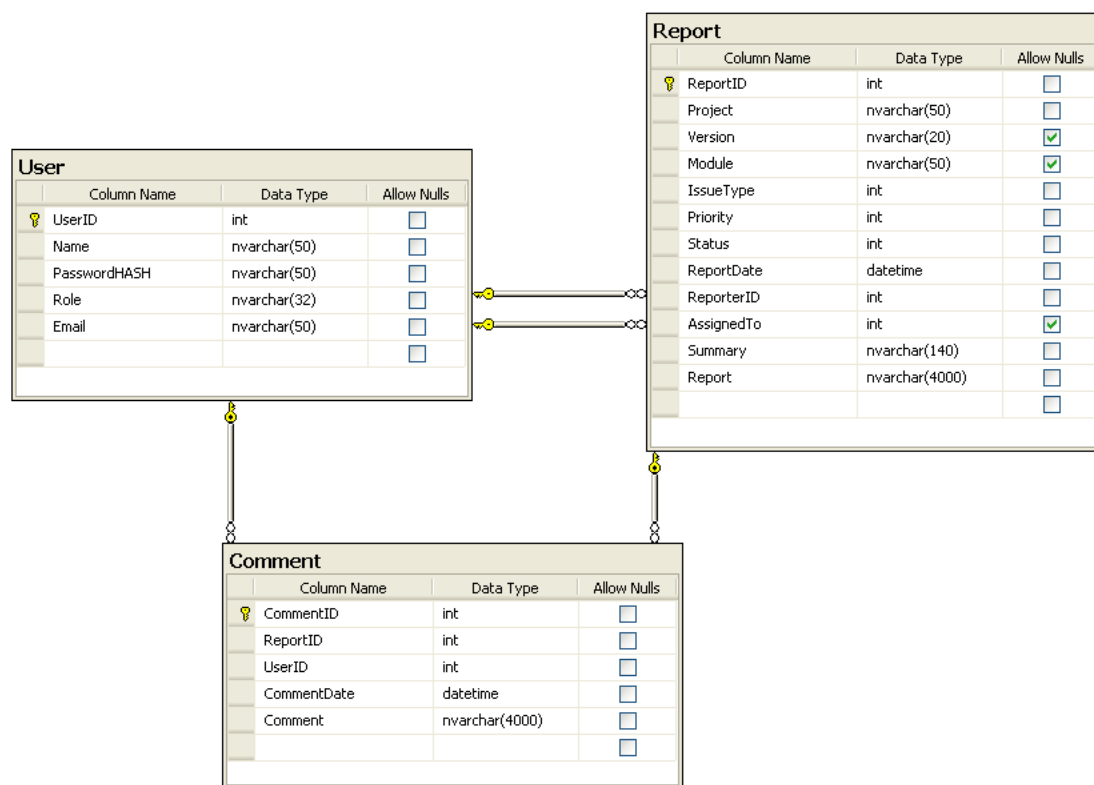
In this new paper, I will describe a small but real-world DataSnap application which was developed in a matter of hours and put in production the same day.

The application is an issue monitoring and tracking system, called D.I.R.T. which stands for Development Issue Report Tool. The system is already being used by developers (in different locations) to enter issue reports for a project of one of my customers, and the

developers often joke that they need to “clean some dirt” if an issue is reported and requires a fix.

DATA MODEL

Before we start to build the application, we need to design a data model to describe the issues and developers that will report and work on these issues. The process of designing this data model usually involves a brainstorm session with (end)users, and a white board, producing in this case the following result:



USER

The User table contains the list of developers as well as testers and other users that need to access the application. Required information are a username and (hashed) password, as well as the e-mail address. Other fields, like address or phone number, can be added later when necessary. In our example, a user can have a role, like “developer”, “tester” or “manager”. The system will use these role names plus a special “admin” role. This technique can be extended by using commas to separate multiple roles for a user when needed.

```
CREATE TABLE [dbo].[User](
  [UserID] [int] IDENTITY(1,1) NOT NULL,
  [Name] [nvarchar](50) NOT NULL,
  [PasswordHASH] [nvarchar](50) NOT NULL,
  [Role] [nvarchar](32) NOT NULL,
  [Email] [nvarchar](50) NOT NULL
)
```

We should not store the real password of the user, but only the HASH value of the password. This also means that only the hashed version of the password is sent over the (secure) connection from the client to the server part of the application. We'll see how to generate the hash shortly, and how to ensure that the connection is secure.

REPORT

The Report table is the most detailed dataset. Here, we should be able to enter a new report (and brief summary of 140 characters – ready to be tweeted as well), describing the Project, Version (optional), Module (optional), Type of issue, as well as Priority. A special field is used to contain the Status of the issue (like reported, assigned, opened, solved, tested, deployed, closed). Also important are the date of the report, the user who reported it, and the user who is assigned to the issue, although the latter is optional, since it may not be known right from the start who should solve the issue.

```
CREATE TABLE [dbo].[Report](
  [ReportID] [int] IDENTITY(1,1) NOT NULL,
  [Project] [nvarchar](50) NOT NULL,
  [Version] [nvarchar](20) NULL,
  [Module] [nvarchar](50) NULL,
  [IssueType] [int] NOT NULL,
  [Priority] [int] NOT NULL,
  [Status] [int] NOT NULL,
  [ReportDate] [datetime] NOT NULL,
  [ReporterID] [int] NOT NULL,
  [AssignedTo] [int] NULL,
  [Summary] [nvarchar](140) NOT NULL,
  [Report] [nvarchar](4000) NOT NULL
)
```

Note that the ReportedID and AssignedTo fields are actually foreign keys to the User table (as depicted in the diagram of the data module). Also note that the IssueType, Priority and Status fields are integers. The actual meaning of these values, which can for example range from 1 to 10, will have to be defined elsewhere. We could have used additional look-up tables to store the meaning as strings, but we've decided to let the GUI worry about the representation, and just use the integer values in our system.

We only have to agree on one rule: the IssueType and Priority range from 1 to 10, and 1 is lower than 10, so a Type of 10 and Priority of 10 is more urgent than a Type 1 issue with Priority 1.

For the Status, we've defined an initial list of potential status values that an issue can have:

Status	Meaning
1	Reported
2	Assigned
3	Opened
6	Solved
7	Tested
8	Deployed
10	Closed

Note that we've left some deliberate gaps, to extend the system in the future with states like "in progress", "need more info", and "verified" (after deployment), which we do not need right now, but may want to add in the future.

Although we can add a look-up table with the translation of the status values and the string representation, we've again decided to let the GUI handle that.

COMMENT

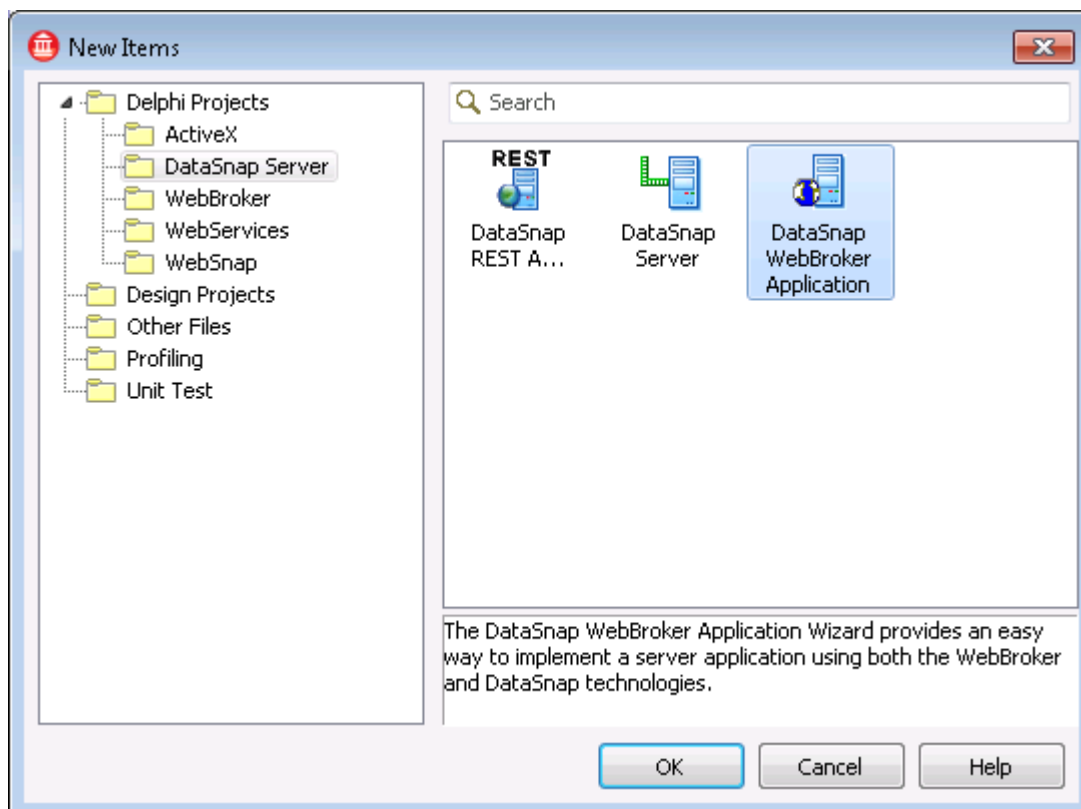
The Comment table is the one that will actually contain the workflow or logbook of what's being reported and done about the issue. This means that for each issue report, we can have a user placing a comment (at a certain date, to see the actual progress).

```
CREATE TABLE [dbo].[Comment](
  [CommentID] [int] IDENTITY(1,1) NOT NULL,
  [ReportID] [int] NOT NULL,
  [UserID] [int] NOT NULL,
  [CommentDate] [datetime] NOT NULL,
  [Comment] [nvarchar](4000) NOT NULL,
)
```

These three tables, linked together by the different ReportID, ReporterID, AssignedTo and UserID fields, are all we need to build the DataSnap Server application for the D.I.R.T. project. Obviously, we could have added more tables, but the goal was to get the optimum functionality as simple as possible (but not simpler).

DATA SNAP SERVER

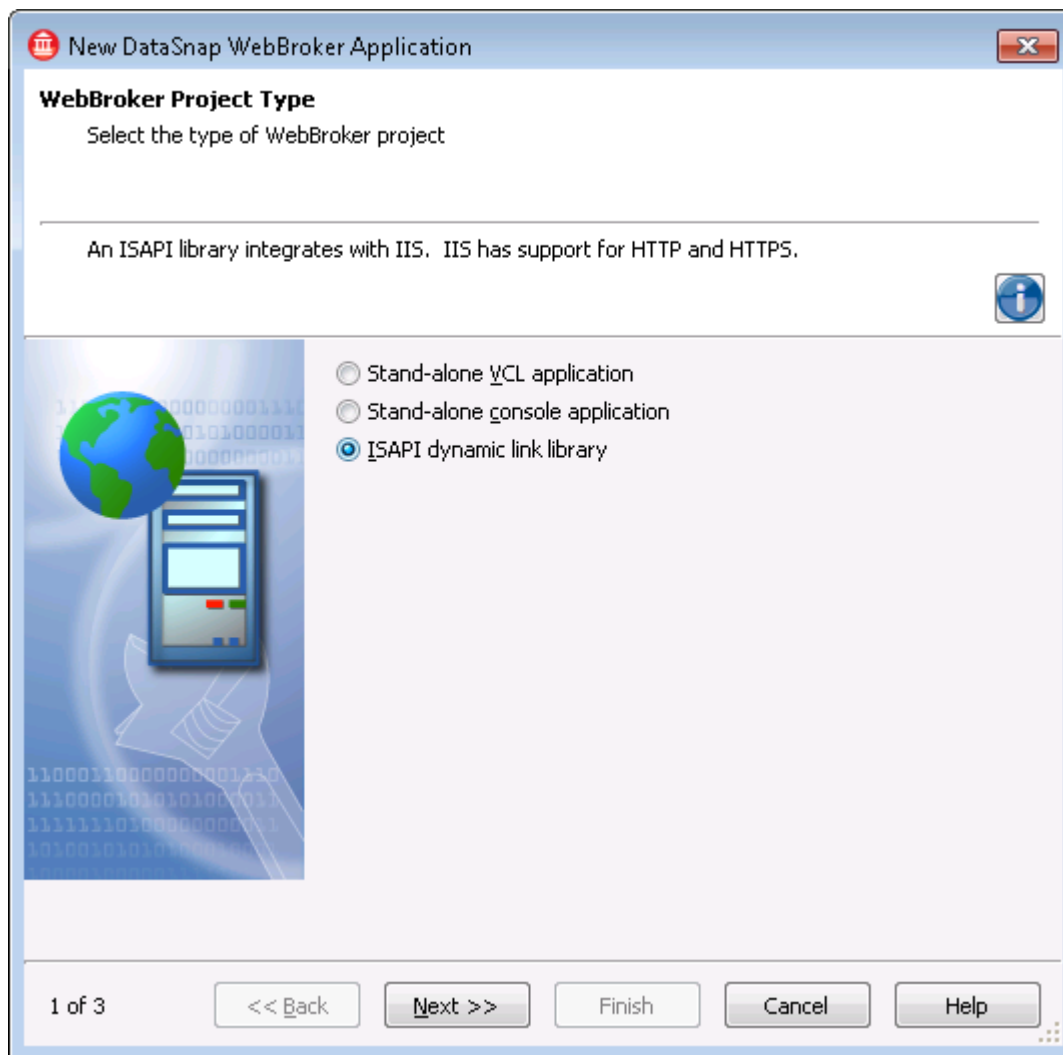
Delphi XE Enterprise (or Architect) provides no less than three new Wizards in the Object Repository to help us create DataSnap Servers.



For the D.I.R.T. application, we need a DataSnap Server type that can be accessed from all over the world in a secure and safe way (some of the issue reports may contain internal details that we do not want to expose to everybody). To allow access from different mobile devices and wireless connections, the use of only the HTTPS (HTTP Secure) protocol was chosen, using an SSL/TLS certificate to provide an encrypted communication and secure identification of the server. HTTPS is more secure than HTTP, which is why we picked HTTPS. Also, while TCP/IP has proven to be faster than HTTP and HTTPS, speed is not a concern for this application. If you need more raw speed, then a choice for TCP/IP as communication protocol may be better.

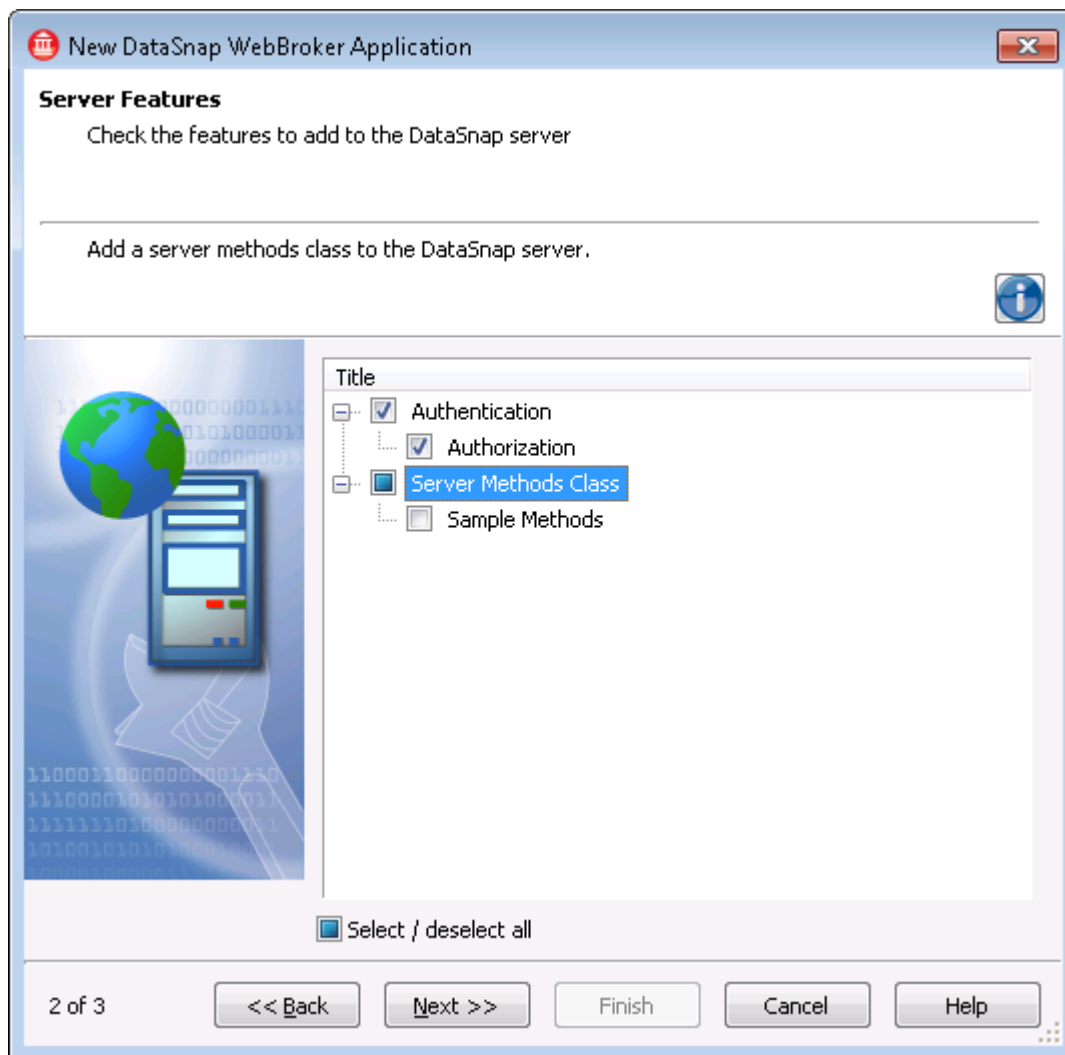
The choice for HTTPS means that there are two DataSnap Server project targets that the wizards can produce: an ISAPI DLL produced by the DataSnap WebBroker or DataSnap REST Application wizards. The difference between the two is the generation of additional files (JavaScript, templates, style sheets and images) for the REST application. These additional files are not needed at this time, so the choice was made to use the DataSnap

WebBroker Application wizard, producing an ISAPI dynamic link library. This ISAPI DLL will be deployed on Microsoft Internet Information Services (IIS), as discussed later in this paper.

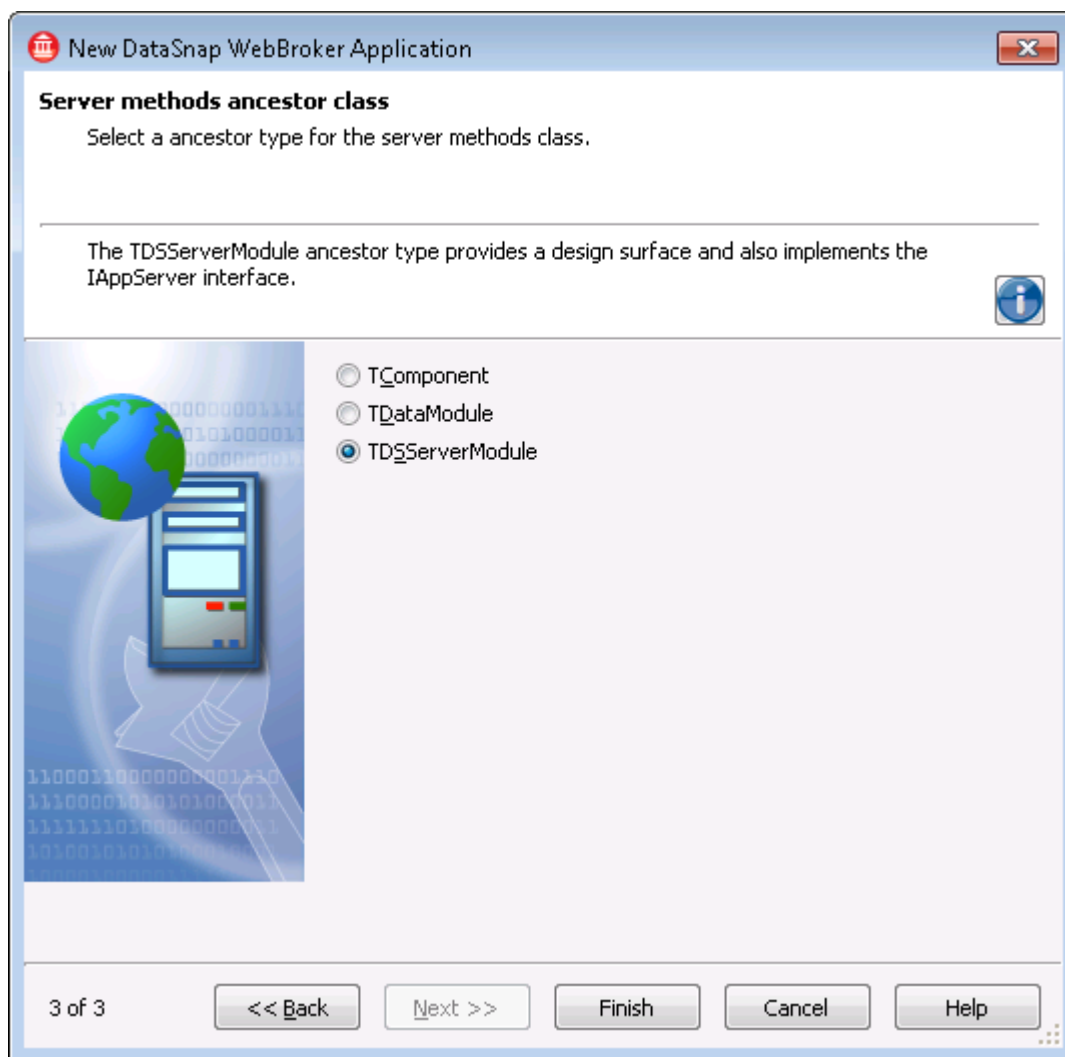


In the second page of the DataSnap WebBroker Application wizard, we can specify if we want to use Authentication and Authorization, and if we want a Server Methods Class with optionally some Sample Methods. Authentication means checking if a user is who he or she claims he is (using a username and password combination), while authorization controls which functionality is explicitly allowed or forbidden based on a user role.

In our case, we want both Authentication and Authorization, as well as a Server Methods Class, but we don't need any Sample Methods, since we can easily add our own custom server methods:



During the last step of the DataSnap WebBroker Application wizard, we can specify the ancestor class for the Server Methods Class. Since we want to export both server methods and datasets (using `TDataSetProviders`), we need to choose `TDSServerModule`. Using a `TComponent` ancestor class, we could define server methods to expose, and using a `TDataModule` we could add non-visual components, but the `TDSServerModule` is the ancestor class that also implements and exposes the `IAppServerFastClass` interface methods using a `TDSPProviderDataModuleAdapter` class behind the scenes.



If we now click on Finish, a new project is generated with two source files: a ServerMethods unit and a unit with a web module inside. Save the project as DirtServer (or any other name you want to give your D.I.R.T. project), the server methods unit as DirtServerMethods.pas and the web module as DirtWebMod.pas.

Note that by renaming the Server Methods unit, the project won't compile anymore. We need to adjust the uses clause in the web module unit, changing ServerMethodsUnit1 to DirtServerMethods.

We also need to adjust the assignment to PersistentClass in the OnGetClass event handler in the same web module:

```
procedure TWebModule1.DSServerClass1GetClass(  
    DSServerClass: TDSServerClass;
```

```

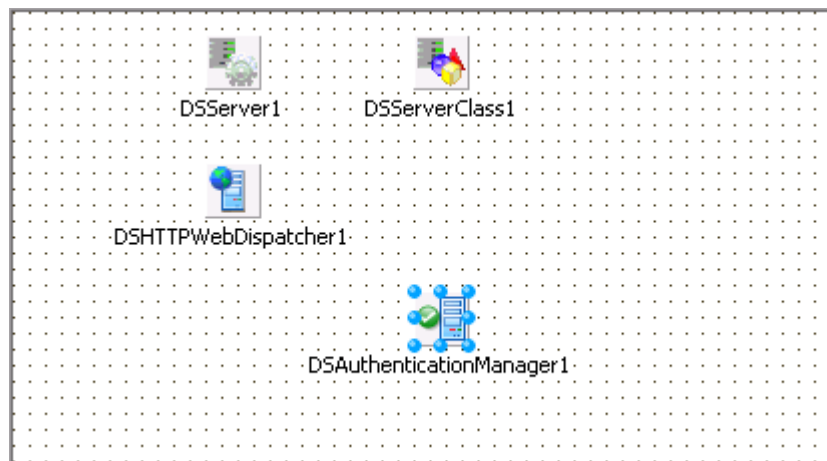
    var PersistentClass: TPersistentClass);
begin
    PersistentClass := DirtServerMethods.TServerMethods1;
end;

```

With these two amendments we should be able to compile the DirtServer DataSnap Server project, producing an ISAPI DLL. This means it's time to start adding the real functionality, starting with the authentication checks.

LOGIN AND AUTHENTICATION

The Web Module that we modified earlier is also the place where we can control the authentication and authorization for the DataSnap Server. The new TDSAuthenticationManager component is our key for this task.



The TDSAuthenticationManager component has an OnUserAuthenticate event handler where we can verify the User, Password (against the User table in the DIRT database) as well as the used protocol (to make sure it's HTTPS and not HTTP). The initial implementation can start as follows, to ensure we're using the secure HTTPS as communication protocol:

```

procedure TWebModule1.DSAAuthenticationManager1UserAuthenticate(
    Sender: TObject; const Protocol, Context, User, Password: string;
    var valid: Boolean; UserRoles: TStrings);
begin
    valid := LowerCase(Protocol) = 'https';
    // now also validate User and Hashed password...
end;

```

The next step involves the verification of the User and Password (which should actually contain the hashed value of the password), making sure a corresponding record for this combination exists in the User table.

For this, we can use a local TSQLConnection component with the sole purpose to perform a direct execute on the User table. Alternately, we can store the user names and hashed password values in a configuration file or passwd.ini file for example at the server machine, and read that file when the user tries to login.

Assuming we want to use a direct connection to the database, place a TSQLConnection component on the web module, set its Driver property to MSSQL, and specify the Hostname, Database, UserName and Password in order to make a valid connection to the database. Make sure to set the LoginPrompt property to False, and toggle the Connected property to see if a successful connection can be made. Apart from the TSQLConnection to talk to the database, we also need a TSQLDataSet to perform the query to look for the User and Password in the User table.

LOCAL DB CONNECTION

If you do not want to place the TSQLConnection and TSQLDataSet components on the web module, you can also create them dynamically. Especially for situations where we only need the TSQLConnection once (to verify the login), this feels like a “clean” solution. Especially if you consider that the SQLConnection parameter settings can be read from a .ini or config file (which is left as exercise for the reader, by the way).

In both cases, the SQL query that we need to perform does a SELECT of the UserID and Role from the User table, passing the User and (hashed) Password in the WHERE clause. Note that we do not have to hash the password here, since it will already have been sent in hashed format (something the client will need to do).

If the SELECT command returns a record, then we can store the UserID value in the current session, using the TDSSessionManager.GetThreadSession to get the session for the current thread, calling PutData to put the value of the UserID in a field with the name “UserID”. The associating Role retrieved by the SELECT command can be used to add to the UserRoles collection.

```
procedure TWebModule1.DSAAuthenticationManager1UserAuthenticate(  
    Sender: TObject; const Protocol, Context, User, Password: string;  
    var valid: Boolean; UserRoles: TStrings);  
var  
    SQLConnection: TSQLConnection;  
    SQLQuery: TSQLQuery;  
    Role: String;  
  
begin  
    valid := LowerCase(Protocol) = 'https';  
  
    if valid then  
        begin // validate User and Hashed password  
            SQLConnection := TSQLConnection.Create(nil);  
            try
```

```

SQLConnection.LoginPrompt := False;
SQLConnection.DriverName := 'MSSQL';
SQLConnection.Params.Clear;

// The following parameters can be read from a config file
SQLConnection.Params.Add('HostName=.');
SQLConnection.Params.Add('Database=DIRT');
SQLConnection.Params.Add('User_Name=sa');
SQLConnection.Params.Add('Password=*****');

SQLQuery := TSQLQuery.Create(nil);
SQLQuery.SQLConnection := SQLConnection;
SQLQuery.CommandText :=
  'SELECT UserID, Role FROM [User] WHERE ' +
  ' (Name = :UserName) AND (PasswordHASH = :Password)';

SQLQuery.ParamByName('UserName').AsString := User;
SQLQuery.ParamByName('Password').AsString := Password;
try
  SQLQuery.Open;
  if not SQLQuery.Eof then
    begin
      valid := True;
      TDSSESSIONMANAGER.GetThreadSession.PutData('UserID',
        SQLQuery.Fields[0].AsString);
      Role := SQLQuery.Fields[1].AsString;
      if Role <> '' then
        UserRoles.Add(Role)
      end
    else
      valid := False
    finally
      SQLQuery.Close;
      SQLQuery.Free
    end
  finally
    SQLConnection.Connected := False;
    SQLConnection.Free
  end
end
end;

```

Note that this implementation only allows one role per user. You can extend this either by adding more table to the data model (defining Roles as well as the UserRoles connections) or by separating the roles using commas in the Role field. I leave both extensions as exercises for the reader (they will be covered in my DataSnap XE Development Essentials courseware manual).

AUTHORIZATION

Once authenticated, we must handle the authorization as well. DataSnap offers the ability to connect user roles with server classes and server methods. The OnUserAuthorize event of the DSAAuthenticationManager component can be used to implement the way we want to deal with authorization. Basically, there are two ways to handle authorization: the optimistic or the pessimistic approach. Optimistic means that any operation is allowed, unless explicitly forbidden. While pessimistic means that no operation is allowed, unless explicitly allowed. The best technique to use depends on the type of application that you want to build. In this case, security is important (you do not want unauthorized people to be able to edit or even view the details of issue reports or comments), so we decided to implement the pessimistic approach.

In short, this means that in the OnUserAuthorize event handler, we should change the default implementation (which assigns True to valid), and start with False instead. We should then examine the collection of Authorized Roles to see if our User Role can be found in the collection of Authorized Roles. This is implemented as follows:

```
if Assigned(AuthorizeEventObject.AuthorizedRoles) then
  for UserRole in AuthorizeEventObject.UserRoles do
    if AuthorizeEventObject.AuthorizedRoles.IndexOf(UserRole) >= 0
      then valid := True;
```

However, sometimes there are operations for which no explicit Authorized Roles have been specified, like the generation of the DataSnap proxy classes for the client side. This functionality would be unavailable if we implement the OnUserAuthorize with the previous code snippet. For that reason, we should use an alternative approach: if no Authorized Roles have been explicitly specified, then we can assume the operation is allowed, except when the user role is found in the list of explicit Denied Roles (in practice, when the Authorized Roles are empty, the Denied Roles will most often also be empty).

This leads to the final operational implementation of the OnUserAuthorize event handler which is as follows:

```
procedure TWebModule1.DSAAuthenticationManager1UserAuthorize(Sender: TObject;
  AuthorizeEventObject: TDSAAuthorizeEventObject; var valid: Boolean);
var
  UserRole: String;
begin
  if Assigned(AuthorizeEventObject.AuthorizedRoles) then
    valid := False // assume NOT OK, unless explicitly allowed !!
  else valid := True; // no Authorized Roles?

  if not valid then
    if Assigned(AuthorizeEventObject.AuthorizedRoles) then
      for UserRole in AuthorizeEventObject.UserRoles do
```

```
    if AuthorizeEventObject.AuthorizedRoles.IndexOf(UserRole) >= 0
    then valid := True;

    if valid then
        if Assigned(AuthorizeEventObject.DeniedRoles) then
            for UserRole in AuthorizeEventObject.UserRoles do
                if AuthorizeEventObject.DeniedRoles.IndexOf(UserRole) >= 0
                then valid := False;
            end;
        end;
```

Note that the above implementation also makes an explicit check to see if a user role was included in both the Authorized Roles and the Denied Roles (in which case the user ends up being denied for the requested server method).

The actual assignment of authorizations by role will need to be done at the server methods level. We'll make sure that admin has the right add new users, a manager can view all issues but only in a read-only way, a tester can report new issues, and both the tester and developer can add comments to issues (as they work to resolve the issues).

SERVER METHODS FOR THE CLIENT

Moving on to the DirtServerMethods unit, we can extend the TServerMethods1 class, derived from TDSServerModule, to define and implement our server methods. The first few methods that I'd like to add here are mainly to support the client application by allowing the right set of options and operations to be enabled (or disabled). Based on the user role, a user can for example only view all issues, or edit only certain issues (belonging to the user in one way or another).

For this purpose, I've implemented two simple server methods: GetCurrentUserID and GetCurrentUserRoles.

```
public
{ Public declarations }
function GetCurrentUserID: Integer;
function GetCurrentUserRoles: String;
```

The GetCurrentUserID will be useful when we want to add a new issue report (or when we want to comment on an existing issue), in which case our UserID is needed. The GetCurrentUserRoles is even more useful, since that function will return the role (or roles) that the current user belongs to, which we can use to open up functionality at the client side.

The implementation of GetCurrentUserRoles consists of returning the UserRoles.Text value from the current session, which is obtained by calling TDSSessionManager.GetThreadSession, as follows:

```
function TServerMethods1.GetCurrentUserRoles: String;  
begin  
    Result := TDSSESSIONMANAGER.GetThreadSession.UserRoles.Text;  
end;
```

Note that the UserRoles.Text will return zero, one or more roles in a single string, optionally separated by CRLF pairs if we have more than one role.

The implementation of the GetCurrentUserID server method is only slightly more complex. Using the same session, we can call the GetData method to return the value of the 'UserID' field (the one that we assigned when a user had a successful login). If the UserID field does not exist or has an invalid value, we return -1 instead.

```
function TServerMethods1.GetCurrentUserID: Integer;  
begin  
    Result := StrToIntDef(  
        TDSSESSIONMANAGER.GetThreadSession.GetData('UserID'), -1);  
end;
```

We won't be using the UserID value for now, but it brings us to an interesting topic: retrieving the list of users and/or adding new users.

SERVER METHOD TO GET USER NAMES

When adding new issues and especially when assigning an issue to another user, it would be helpful if the client application could show a list of user names (and associated UserID values). For this purpose, we can add a third server method to the TServerMethods1 class, called GetUserNames, returning a dataset with the user names and UserID values.

This server method should only be called by users with the Tester or Developer role, since these are the only roles that can report new issues or add comments to an existing reported issue. As a result, the declaration can be decorated with the TRoleAuth custom attribute, explicitly defining the list of authorized roles as follows:

```
[TRoleAuth('Tester,Developer')]  
function GetUserNames: TDataSet;
```

We could optionally include a second argument to the TRoleAuth custom attribute, specifying roles for which this server method is explicitly forbidden. However, since we've implemented the pessimistic authorization approach, this is not needed, and only users with the roles that are explicitly allowed will be able to execute this server method.

For the implementation of this server method, we not only need the TSQLConnection component, but also a TSQLDataSet component called sqlUser. The TSQLDataSet has its SQLConnection property connected to the TSQLConnection component, and its

CommandType property set to Dbx.SQL. The CommandText itself consists of the SQL SELECT command to select the UserID and Name fields from the User table as follows:

```
SELECT UserID, Name FROM [User]
```

With this SQL SELECT command specified, we can implement the GetUserNames server method, which consists of opening the sqlUser table and moving to the First record (in case it was already opened, but not positioned at the beginning).

```
function TServerMethods1.GetUserNames: TDataSet;  
begin  
  try  
    sqlUser.Open;  
    sqlUser.First;  
    Result := sqlUser  
  except  
    on E: Exception do  
      begin  
        Result := nil;  
        sqlUser.Close;  
        SQLConnection1.Close  
      end  
    end;  
end;
```

Note that we do not have to open the TSQLConnection component, since this will be opened implicitly as soon as we open the TSQLDataSet. We do need to make sure that the sqlUser is not closed before the function ends, since the "open" TSQLDataSet must be passed as function result of this server method.

Getting a list of existing users is nice, but initially there will be no users. It's time to explore the server method to add a new user with a specific password and role to the database.

SERVER METHOD TO ADD NEW USER

Only a user with the Admin role is explicitly allowed to add new users. We can implement this by adding a 4th server method called AddUser to the TServerMethods1 class, using the TRoleAuth custom attribute again to explicitly specifying that only the Admin role is allowed to call this server method:

```
[TRoleAuth('Admin')]  
procedure AddUser(const User, Password, Role, Email: String);
```

The parameters for the AddUser procedure are the new user name, the password (unencrypted, it will be hashed by the AddUser server method itself, although you are free to change this and pass the already hashed version of the password along), the role and the e-mail address.

The AddUser method will require a TSQLConnection component, connected to the DIRT database. Once that component is placed on the Server Methods unit, we can implement the AddUser server method as follows:

```
procedure TServerMethods1.AddUser(const User, Password, Role, Email: String);  
var  
    SQLQuery: TSQLQuery;  
begin  
    SQLQuery := TSQLQuery.Create(nil);  
    SQLQuery.SQLConnection := SQLConnection1;  
    SQLQuery.CommandText := 'INSERT INTO [User] ' +  
        ' (Name, PasswordHASH, Role, Email) ' +  
        ' VALUES (:Name, :PasswordHASH, :Role, :Email)';  
    SQLQuery.ParamByName('Name').AsString := User;  
    SQLQuery.ParamByName('PasswordHASH').AsString := HashMD5(Password);  
    SQLQuery.ParamByName('Role').AsString := Role;  
    SQLQuery.ParamByName('Email').AsString := Email;  
    SQLConnection1.Open;  
    try  
        SQLQuery.ExecSQL;  
    finally  
        SQLConnection1.Close;  
    end;  
end;
```

The HashMD5 support function is a private function defined in the TServerMethods1 class, which can be implemented as follows, using the Indy IdHashMessageDigest unit with the TIdHashMessageDigest5 type:

```
function TServerMethods1.HashMD5(const Str: String): String;  
var  
    MD5: TIdHashMessageDigest5;  
begin  
    MD5 := TIdHashMessageDigest5.Create;  
    try  
        Result := LowerCase(MD5.HashStringAsHex(Str, TEncoding.UTF8))  
    finally  
        MD5.Free  
    end  
end;
```

Note that you may want to make the function HashMD5 private (or protected) to ensure that it's not exported as server method if you do not want to share the MD5 hashing as a server method.

Finally, note that you will also need to enter at least one user in the database with the Admin role in order to be able to add other users (and the first user either needs to be added by hand, or without the explicit custom attribute that defines that only users with

the Admin role can add users, otherwise you're faced with a chicken-and-the-egg problem and will not be able to login to add new users in the first place).

SERVER METHOD TO GET ALL ISSUES

After all this administration, it's finally time to do some real functional work. In this case, presenting the list of all reported issues. With a filter on the status, so we can retrieve issues between a MinStatus and MaxStatus value, with the result value being a dataset so we can connect it to a grid or other data-aware controls at the client.

The 5th server method is a function called GetIssues and should only be available to users with the Manager role, defined as follows:

```
[TRoleAuth('Manager')]  
// Return all issues (read-only) between MinStatus..MaxStatus  
function GetIssues(MinStatus,MaxStatus: Integer): TDataSet;
```

For the implementation of this server method, we not only need the TSQLConnection component, but also a TSQLDataSet component called sqlReports.

The TSQLDataSet has its SQLConnection property connected to the TSQLConnection component, and its CommandType property set to Dbx.SQL. The CommandText itself consists of the SQL SELECT command to select all fields from the Report, and specifying in the WHERE clause the minimum and maximum values for the Status field, as follows:

```
SELECT "ReportID", "Project", "Version", "Module", "IssueType",  
        "Priority", "Status", "ReportDate", "ReporterID", "AssignedTo",  
        "Summary", "Report"  
FROM "Report"  
WHERE Status >= :MinStatus AND Status <= :MaxStatus  
ORDER BY Status
```

A potential downside of using this simple SQL command is that we need to translate some of the field values at the client side. For IssueType, Priority and Status, that is not a big problem. But for the ReporterID and AssignedTo values, we would have to have a list of usernames to substitute for these UserID values. This list is available by calling the GetUserNames, but only for a user with the Tester or Developer role right now.

Since the SQL query should result in a read-only list anyway, we can extend it by joining with the User table and replacing the ReporterID and AssignedTo fields with the actual names of these users, as follows:

```
SELECT "ReportID", "Project", "Version", "Module", "IssueType",  
        "Priority", "Status", "ReportDate",  
        UReporterID.Name AS "Reporter", UAssignedTO.Name AS Assigned,  
        "Summary", "Report"
```

```

FROM "Report"
LEFT OUTER JOIN [User] UReporterID ON UReporterID.UserID = ReporterID
LEFT OUTER JOIN [User] UAssignedTO ON UAssignedTO.UserID = AssignedTO
WHERE Status >= :MinStatus AND Status <= :MaxStatus
ORDER BY Status

```

Note that I've used LEFT OUTER JOIN commands because the AssignedTo field of the User table is not required (i.e. it can be a NULL value, in which case a normal LEFT JOIN would not yield a result).

With this TSQLDataSet, we can implement the server method GetIssues as follows, passing the MinStatus and MaxStatus integer argument values to the MinStatus and MaxStatus query parameters:

```

function TServerMethods1.GetIssues(MinStatus,MaxStatus: Integer): TDataSet;
begin
  try
    SQLConnection1.Open;
    sqlReports.Close;
    sqlReports.ParamByName('MinStatus').Value := MinStatus;
    sqlReports.ParamByName('MaxStatus').Value := MaxStatus;
    sqlReports.Open;
    Result := sqlReports
  except
    on E: Exception do
      begin
        Result := nil;
        sqlReports.Close;
        SQLConnection1.Close
      end
    end;
  end;
end;

```

Note that we must not close the sqlReports, but allow the TSQLDataSet to remain open while the function is executed. At the client side, we'll be able to work with the resulting dataset, as we'll see shortly.

SERVER METHOD REPORT NEW ISSUE

While the user with the Manager role can get a read-only overview of all reported issues, the user with Tester role can actually submit new issues. For this, we need a 6th server method, ReportNewIssue, with the following declaration:

```

[TRoleAuth('Tester')]
function ReportNewIssue(Project, Version, Module: String;
  IssueType, Priority: Integer; // Status = 1
  Summary, Report: String;
  AssignedTo: Integer = -1): Boolean;

```

Note that we pass all fields of the Report table, except for the ReportID (which is an identity or autoincrement field), the ReporterID, and the Status fields. The ReporterID can be determined by the server method based on the logged in UserID, and the status field will automatically get the value 1 for "reported". The AssignedTo field is optional, with a default parameter value of -1. This value means that the reported issues is not yet assigned to a user. Based on the value of AssignedTo, the SQL INSERT command will be slightly different (either with or without the AssignedTo field and parameter).

The implementation of the ReportNewIssue server method is as follows:

```
function TServerMethods1.ReportNewIssue(Project, Version, Module: String;
    IssueType, Priority: Integer; // Status = 1
    Summary, Report: String;
    AssignedTo: Integer = -1): Boolean;
var
    InsertSQL: TSQLQuery;
begin
    Result := False;
    InsertSQL := TSQLQuery.Create(nil);
    try
        InsertSQL.SQLConnection := SQLConnection1;
        if AssignedTo >= 0 then
            InsertSQL.CommandText := 'INSERT INTO [Report] ([Project], ' +
                ' [Version],[Module],[IssueType],[Priority],[Status], ' +
                ' [ReportDate],[ReporterID],[AssignedTo],[Summary],[Report]) ' +
                ' VALUES (:Project,:Version,:Module,: IssueType,:Priority, ' +
                ' 1, @Date, :ReporterID,:AssignedTo,:Summary,:Report) '
        else // No AssignedTo
            InsertSQL.CommandText := 'INSERT INTO [Report] ([Project], ' +
                ' [Version],[Module],[IssueType],[Priority],[Status], ' +
                ' [ReportDate],[ReporterID],[Summary],[Report]) ' +
                ' VALUES (:Project,:Version,:Module,:IssueType,:Priority, ' +
                ' 1, @Date, :ReporterID,:Summary,:Report) ';

        try
            InsertSQL.ParamByName('Project').AsString := Project;
            InsertSQL.ParamByName('Version').AsString := Version;
            InsertSQL.ParamByName('Module').AsString := Module;
            InsertSQL.ParamByName('IssueType').AsInteger := issueType;
            InsertSQL.ParamByName('Priority').AsInteger := Priority;
            InsertSQL.ParamByName('ReporterID').AsInteger :=
                StrToIntDef(TDSSessionManager.GetThreadSession.GetData('UserID'),0);
            if AssignedTo >= 0 then
                InsertSQL.ParamByName('AssignedTo').AsInteger := AssignedTo;
            InsertSQL.ParamByName('Summary').AsString := Summary;
            InsertSQL.ParamByName('Report').AsString := Report;
            Result := InsertSQL.ExecSQL = 1
        except
            on E: Exception do
                CodeSite.SendException(E)
```

```
        end
    finally
        InsertSQL.Free;
        SQLConnection1.Close
    end;
end;
```

The function will return True in case the new record has been inserted correctly. Note that the error handling only consists of sending the exception to CodeSite at this time. Feel free to re-raise the exception or add your own error handling.

At this point, we've added no less than 6 server methods to retrieve information (GetCurrentUserID, GetCurrentUserRoles, GetUserNames, GetIssues) or add new users (AddUser) or reports (ReportNewIssue). Two methods, GetUserNames and GetIssues, return datasets, but as read-only dataset that cannot be modified.

In order to allow users with the Developer role to retrieve reports and add comments, we need to add functionality to the server module that goes beyond that which the server methods can offer.

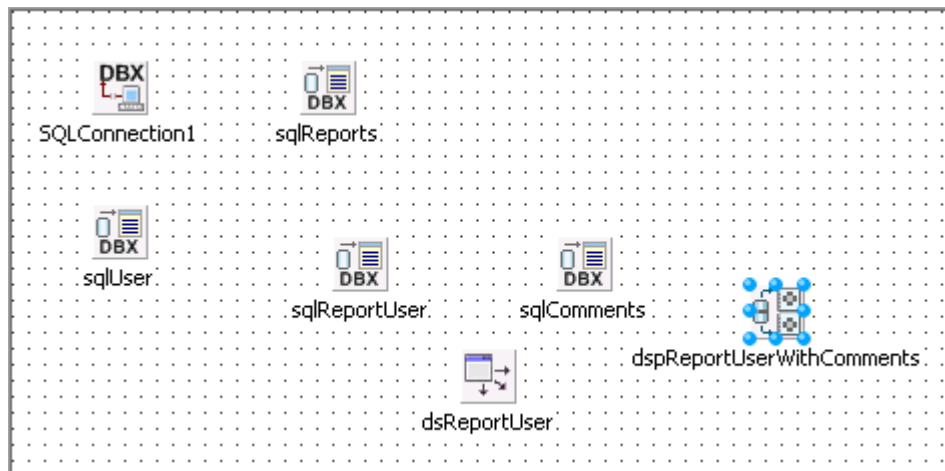
EXPORTING DATA: OPEN ISSUES

In order to not only view but also edit the reported issues for a developer, we need to export a TDataSetProvider that can be used at the client side to make changes to the data (insert, update and/or delete), and apply the updates back to the server.

A developer working on an issue report should see not just the initial issue report, but also all comments that have been posted for that particular issue. As a result, the TDataSetProvider should export not only records from the Reports table, but also from the Comments table, in a master-detail relationship.

To implement this, we need to add two new TSQLDataSet components, called sqlReportUser and sqlComments, and a TDataSource component called dsReportUser that points its DataSet property to the sqlReportUser, and is used in turn as DataSource for the sqlComments component.

Finally, we need a TDataSetProvider component, called dspReportUserWithComments connecting its DataSet property to the master sqlReportUser dataset.

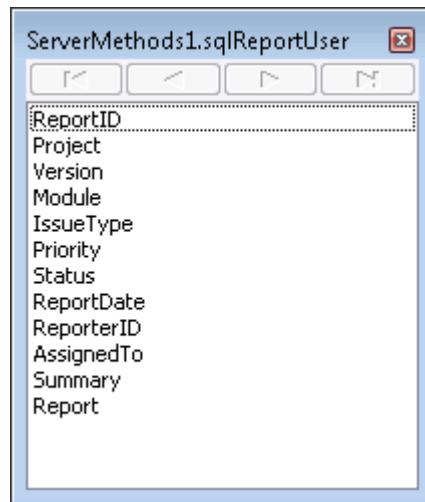


Both TSQLDataSet components must point their SQLConnection property to the TSQLConnection component of course. The SQL property for the sqlReportUser is defined as follows, retrieving the records from the Report table within the specified minimum and maximum status values, where either the ReporterID or the AssignedTo fields are equal to the current user:

```
SELECT "ReportID", "Project", "Version", "Module", "IssueType",  
        "Priority", "Status", "ReportDate", "ReporterID", "AssignedTo",  
        "Summary", "Report"  
FROM "Report"  
WHERE (Status >= :MinStatus) AND (Status <= :MaxStatus)  
        AND ((AssignedTo = :AssignedTo) OR (ReporterID = :ReporterID))
```

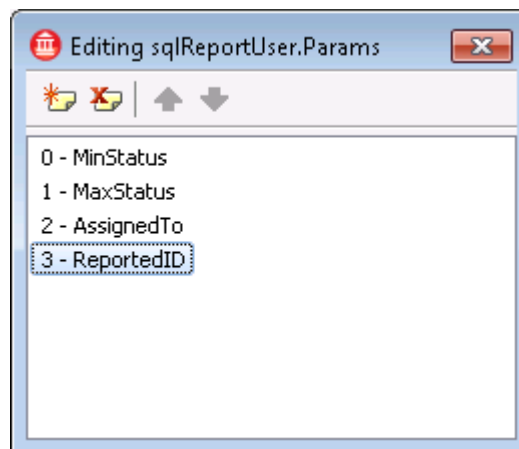
Note that this will return both the issues reported by the current user, and the issues assigned to the current user. Which can both be considered issues “belonging” to the current user.

This SQL command returns 12 fields. We can double-click on the sqlReportUser component to start the Fields Editor. Next, we can right-click in the Fields Editor and select “Add all fields”. This will produce a list of the 12 fields that the query will produce:



Some of these fields require special treatment. The ReportID field for example, is an autoincrement field. This means that we should modify the ProviderFlags property for this field. The pflnUpdate subproperty must be set to False (since we cannot assign an initial value to this primary key, and should never update it anyway), but the pflnKey should be set to True. The pflnWhere should already be set to True.

Note that the SQL command has four parameters: MinStatus, MaxStatus, AssignedTo and ReporterID. However, only two of them will be provided by the client, and two will be automatically filled by the server itself (and as a consequence need to be removed from the list).

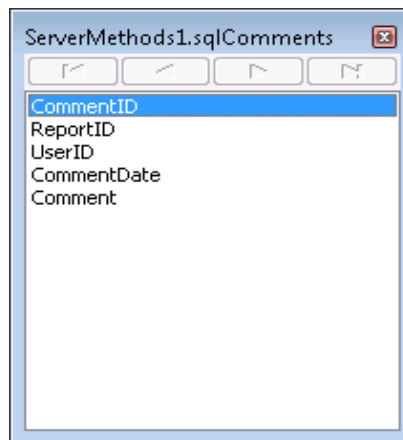


The MinStatus and MaxStatus input parameters are of type ftInteger and should be kept. The AssignedTo and ReportedID parameters should be removed from the Params collection, so they will not show up at the client side (and can be filled in by the server side itself, based on the UserID value in the session of the current user).

Moving on to the detail: the sqlComments dataset should point its DataSource property to dsReportUser, pointing to the master sqlReportUser. The actual SQL command for the comments details is as follows, using a parameter called ReportID that connects the comments to the reports:

```
SELECT CommentID, ReportID, UserID, CommentDate, Comment
FROM Comment
WHERE (ReportID = :ReportID)
```

This time, we must again modify the ProviderFlags for the primary key CommentID. So, double-click on the sqlComments component to start the Fields Editor, right-click in the Fields Editor and select "Add all fields".



Select the CommentID field, and in the Object Inspector, expand the ProviderFlags property of the CommentID field. Set the pflnUpdate subproperty to False, and the pflnKey subproperty to True (while pflnWhere should already be True as well). This should ensure that the primary key CommentID is never sent to the server as assignment, but is used in the WHERE clause to locate a record.

Since the sqlComments dataset is connected to the master sqlReportUser dataset, the TDataSetProvider will export them both as so-called nested datasets, with the report being the master and the comments (if any) as nested detail dataset. We'll see that in the client application in more detail.

There is one more property of the dspReportUserWithComments TDataSetProvider that you may want to change: the value of the UpdateMode. By default, this property is set to upWhereAll. Which means that for UPDATE and DELETE commands, all fields will be part of the WHERE clause (with the exception of Blob fields), to locate the correct record. This is especially required if a table has no primary key. However, when a primary key is

present, which is our case, then we can change the `upWhereAll` to the more efficient `upWhereChanged`. Using that setting, the `WHERE` clause will only contain the primary key as well as the fields that have been changed (and not the fields that remained unchanged). This allows us to actually merge records – in our case modified issue reports – as long as two people do not modify the same field at the same time.

Finally, note that there is also an `upWhereKeyOnly` option, but that's dangerous, since it only passes the primary key in the `WHERE` clause of `UPDATE` and `DELETE` commands, ignoring any changes that other users may have done.

Since both datasets are read-only unidirectional datasets, we need to perform one final task to allow the master-detail relationship to work. While stepping through the result set of the master, each time we move one record further next in the master record set, we must "reset" the detail. This does not happen automatically, because the detail is also a unidirectional dataset. In the `AfterScroll` event of the `sqlReportUser` we can close and re-open the `sqlComments` dataset, and also pass the new value of the `ReportID` parameter.

This is implemented as follows:

```
procedure TServerMethods1.AS_sqlReportUserAfterScroll(DataSet: TDataSet);  
begin  
    sqlComments.Close;  
    sqlComments.ParamByName('ReportID').AsInteger :=  
        sqlReportUser.FieldByName('ReportID').AsInteger;  
    sqlComments.Open;  
end;
```

This will, in effect, ensure that the detail query is re-executed for every master record as soon as we scroll to that master record, filling the entire nested dataset at the server side before it's exported by the `TDataSetProvider` to the client.

Note that the event was named `AS_sqlReportUserAfterScroll` and not the default name `sqlReportUserAfterScroll`. This is done on purpose. The reason is as follows: any public method of a server method class, will be exposed. And this includes public event handlers. However, there is one exception to this rule: methods that start with the `AS_` prefix will be hidden and not exposed from the server methods class. We can use this knowledge to hide event handlers, so they won't resurface in the server methods proxy unit at the client side (since they are quite useless to call from the client anyway).

While the client side will supply the value for the `MinStatus` and `MaxStatus` parameters of the master query, we have to enter the values for the `AssignedTo` and `ReporterID` parameters by ourselves at the server side. This can be done by implementing the `BeforeGetRecords` event handler of the `TDataSetProvider`. Here, we can fill the parameters of the `sqlReportUser TSQLDataSet` before it is opened. Both the `ReporterID` and

AssignedTo parameter values can be assigned to the value of the UserID for the current user that we stored in the current thread for the user right after the login. This is implemented as follows:

```
procedure TServerMethods1.AS_dspReportUserWithCommentsBeforeGetRecords(
    Sender: TObject; var OwnerData: OleVariant);
var
    UserID: Integer;
begin
    UserID := StrToIntDef(TDSSessionManager.GetThreadSession.GetData('UserID'), 0);
    if sqlReportUser.Params.FindParam('ReporterID') = nil then
    begin
        with sqlReportUser.Params.AddParameter do
        begin
            DataType := ftInteger;
            ParamType := ptInput;
            Name := 'ReporterID';
            AsInteger := UserID
        end
    end
    else
        sqlReportUser.Params.FindParam('ReporterID').AsInteger := UserID;

    if sqlReportUser.Params.FindParam('AssignedTo') = nil then
    begin
        with sqlReportUser.Params.AddParameter do
        begin
            DataType := ftInteger;
            ParamType := ptInput;
            Name := 'AssignedTo';
            AsInteger := UserID
        end
    end
    else
        sqlReportUser.Params.FindParam('AssignedTo').AsInteger := UserID;
    end;
```

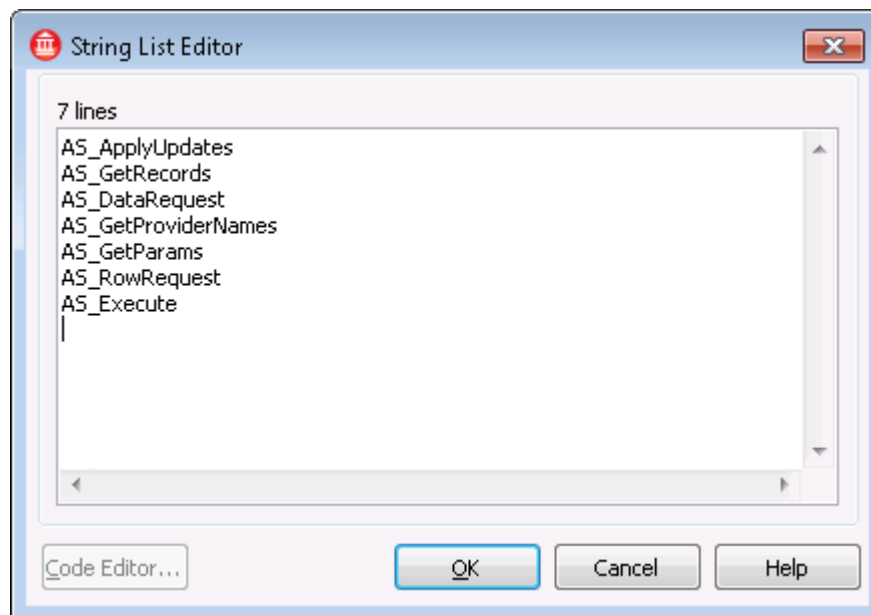
This will ensure that all parameters of the sqlReportUser are filled before the query is opened, and, as a consequence, any user who requests the Report and Comments records from the dspReportUserWithComments will only get the reports where the current user was either the reporter (i.e. ReporterID) or the one assigned to it (i.e. AssignedTo), or both. But you can only see “your reports”, and not reports that belong to someone else. For that, you need to have the Manager role and call the GetIssues server method that will return an overview of all reported issues (but without the comments).

TDataSetProvider ROLE BASED AUTHORIZATION

There’s one final thing missing for the DataSnap Server: the role based authorization for the TDataSetProvider. Unlike the server methods, we cannot just add a custom attribute to

the component in the server module. Instead, we have to switch to the DirtWebMod and use the TDSAuthenticationManager component.

This component has a property called Roles, where we can define a collection of Roles with ApplyTo, AuthorizedRoles and DeniedRoles properties. The ApplyTo property is the place where we can specify a method name, a class name, or a class name followed by a dot followed by a method name. In our case, I want to specify that only users with the Developer role should be allowed to call the 7 pre-defined IAppServerFastCall methods. So in the Apply property we need to add these 7 methods as follows:



Note that we could prefix them with TServerMethods1. in order to indicate that we only want to secure the 7 methods from that specific server methods class.

Once the Apply property has been specified, we can enter Developer as role for the AuthorizedRoles property, specifying that only a user with the Developer role is allowed to use the exported TDataSetProvider.

You may feel that it's a bit too strict to allow only the developer to edit existing issue reports and add comments. Feel free to add the Tester and even the Manager roles to this list as well (although it depends on the manager if you really want to give this user the ability to modify reports and/or add comments).

SERVER DEPLOYMENT

For the DataSnap D.I.R.T. Server to be safely and securely deployed, we either need to deploy the ISAPI DLL in Microsoft's Internet Information Services using the HTTPS

protocol, or we should deploy it as stand-alone DataSnap server with the RSA and PC1 filters to encrypt the transport channels. In the latter case, it's recommended to use TCP/IP as transport protocol and not HTTP, since TCP/IP is a lot faster. So the real-world deployment options that we've used are:

- ISAPI DLL using HTTPS on IIS
- Stand-alone using TCP/IP with PC1 and RSA filters

IIS has the additional benefit of allowing us to configure application pools that support automatic load balancing, recycling, and limitation of the CPU and memory usage of the server. For this reason, the D.I.R.T. DataSnap Server has been deployed as an ISAPI DLL on IIS on a web server that has an SSL certificate installed for the domain. We will use www.bobswart.nl for the purpose of this paper. A virtual directory DataSnapXE has been created, and a special application pool configured for the applications in this virtual directory. Finally, the DirtServer.dll has been placed inside this virtual directory, resulting in the following URL: <https://www.bobswart.nl/DataSnapXE/DirtServer.dll>

Calling this URL will only present us with the "DataSnap XE Server" text, but at least that is confirmation enough that the server is deployed. Of course, we should also ensure that the DirtServer.dll can connect to the database. For that purpose, the required database driver should also be deployed to the server machine. For SQL Server 2008, this means the dbxmss.dll which should be placed in the search path (like the Windows\System32 directory, to ensure that the ISAPI DLL can load it).

Note that apart from the DirtServer.dll and the dbxmss.dll we do not have to deploy any other files to the web server. The MIDAS.dll is not needed either, since we can add the MidasLib unit to the uses clause of the project, which will link in the MIDAS.dll inside the server project itself.

If you deploy the DataSnap standalone server, using TCP/IP and the RSA and PC1 filters, then you must also deploy two Indy specific SSL DLLs: libeay32.dll and ssleay32.dll – or make sure they already exist at the server machine. These DLLs are needed for the RSA filter (which encrypts the password used by the PC1 filter). Without these two DLLs, any client who wants to connect to the server will get a "Connection Closed Gracefully" message, because the server was unable to load the two DLLs to start the RSA filter to encrypt the PC1 keys, etc.

By the way, the same two DLLs will be required for any DataSnap client, whether connected to the TCP/IP server using the RSA and PC1 filters, or whether connected to the ISAPI filter using HTTPS. But let's first build the client, and worry about deployment later.

DATASNAP CLIENT

With the D.I.R.T. DataSnap Server deployed and available for testing, we can start to write the DataSnap Client. This will be a smart client; a standalone executable that we can put on a USB stick or place anywhere and can run from any Windows machine (provided we have an internet connection to the <https://www.bobswart.nl> server to access the DirtServer project).

Create a new VCL Forms application, and add a Data Module to centralize the data access components to the DataSnap Server. As first step, we should place a TSQLConnection component on the Data Module. Set the Driver property to Datasnap, and then expand the Driver node to view the DataSnap specific properties in the Object Inspector. We should first set the CommunicationProtocol to https, the Port to 443 and the HostName to www.bobswart.nl (feel free to connect to your own server, of course). The URLPath property should be set to /DataSnapXE/DirtServer.dll which specifies the location of the DirtServer on the web server. Don't forget to set the LoginPrompt property to False, or you will be presented with a login dialog when you try to connect to the server. And that's not useful, since we need to pass the hashed password (instead of the real password) from the client to the server.

There is one catch here... without a valid value for the DSAuthUser and DSAuthPassword properties, we are not allowed to connect to the server, and, as a consequence, the generation of the DataSnap Client Classes will fail. For the DirtServer on my machine, that will be a problem (but you can use the pre-generated client classes unit DBXClientClasses.pas from the project archive if you want). For your own DataSnap server, you could temporarily disable the OnUserAuthenticate event handler of course, assigning True to valid at least for as long as you need to generate your client classes, and then enable the authentication checks again.

Once the client classes have been generated, you'll see the TServerMethods1Client class which consists of the following public server methods (among others):

```
type
  TServerMethods1Client = class(TDSAdminClient)
  private
    ....
  public
    constructor Create(ADBXConnection: TDBXConnection); overload;
    constructor Create(ADBXConnection: TDBXConnection;
      AInstanceOwner: Boolean); overload;
    destructor Destroy; override;

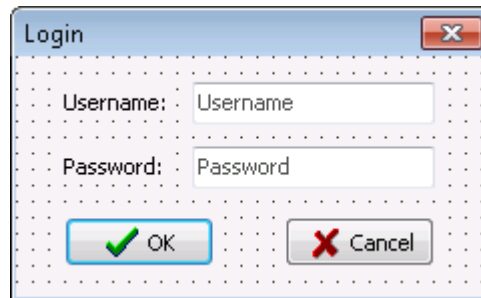
    function GetCurrentUserID: Integer;
    function GetCurrentUserRoles: string;
    function GetUserNames: TDataSet;
```

```
procedure AddUser(User: string; Password: string;  
  Role: string; Email: string);  
function ReportNewIssue(Project: string; Version: string;  
  Module: string; IssueType: Integer; Priority: Integer;  
  Summary: string; Report: string; AssignedTo: Integer): Boolean;  
function GetIssues(MinStatus: Integer; MaxStatus: Integer): TDataSet;  
end;
```

But before we can create an instance of the TServerMethods1Client and call the server methods, we should first ensure we can make a connection.

LOGIN

We can add a new form to the project and design it to be our custom login form, for example as follows:



Make sure to set the PasswordChar property of the TEdit (called edPassword) used to enter the password.

This dialog can be displayed as a modal dialog, and the resulting username and password can be assigned to the TSQLConnection's Params, adding or replacing the existing values. We should hash the password the same way we did in the server application, leading to the following code for a Login event:

```
procedure TFormClient.Login1Click(Sender: TObject);  
var  
  MD5: TIdHashMessageDigest5;  
  Server: TServerMethods1Client;  
  UserRoles: String;  
  
begin  
  DataModule1.SQLConnection1.Connected := False;  
  UserID := -1;  
  
  with TFormLogin.Create(Self) do  
  try  
    if ShowModal = mrOK then  
    begin  
      SQLConnection1.Params.Values['DSAuthenticationUser'] := Username;  
    end  
  finally  
  end  
end;
```

```

MD5 := TIdHashMessageDigest5.Create;
try
  SQLConnection1.Params.Values['DSAuthenticationPassword'] :=
    LowerCase(MD5.HashStringAsHex(Password, TEncoding.UTF8));

  SQLConnection1.Connected := True; // try to login...
  Server := TServerMethods1Client.Create(SQLConnection1.DBXConnection);
  try
    UserID := Server.GetCurrentUserID;
    UserRoles := Server.GetCurrentUserRoles;

    // Adjust GUI capabilities based on UserRoles...

  finally
    Server.Free
  end
finally
  MD5.Free
end
end
finally
  Free // TFormLogin
end
end;

```

Note that I haven't shown my actual code to disable and/or hide portions of the GUI based on the user roles. This is left as exercise for the reader. You can, of course, download the example DirtServer and DirtCleaner client projects and examine the source code in more detail.

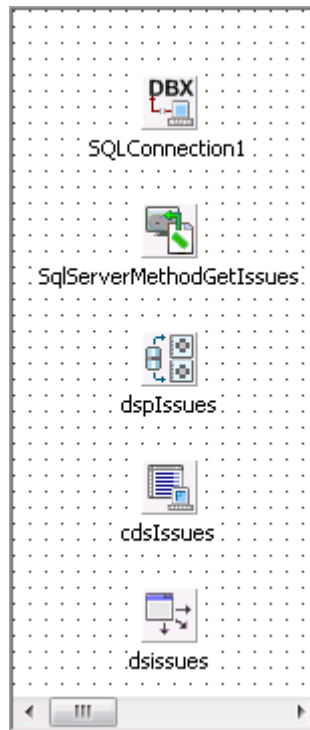
The Login form will close the existing connection to the DataSnap Server and assign new values to the DSAAuthenticationUser and DSAAuthenticationPassword parameters of the TSQLConnection component. Whether or not the login was successful can be seen if the connection is re-established, followed by the call to the GetCurrentUserID and GetCurrentUserRoles server methods.

DATA MODULE AND SERVER METHODS

Although we already generated the DataSnap client classes using the TSQLConnection component on the data module, in practice, we will seldom need to call the server methods directly. The Login code showed two calls already: to GetCurrentUserID and to GetCurrentUserRoles, but the other server methods are typically used to retrieve a dataset, read-only or modifiable. There's one exception, the server method ReportNewIssue, but that one will be covered later.

For now, return to the data module, and place four more components on the data module: a TSqlServerMethod component, called SqlServerMethodGetIssues, a TDataSetProvider

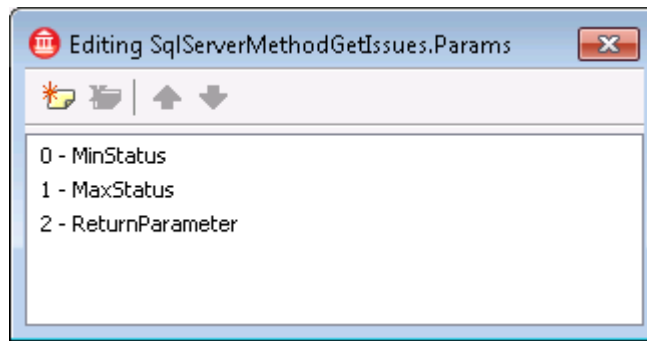
component called `dspIssues`, a `TClientDataSet` component called `cdsIssues`, and finally a `TDataSource` component named `dsIssues`.



The `SqlServerMethodGetIssues` component needs to connect its `SqlConnection` property to the `SqlConnection1` component, so it can talk to the DataSnap Server. We can then use the drop-down list for the `ServerMethodName` to select the server method we want to execute. Note that the `TSqlConnection` component must be connected to the (running) DataSnap Server in order to display the list of Server Methods.

For our example here, we need to call the `TServerMethods1.GetIssues` server method. This method can only be called by a user in the "Manager" role, and will return a read-only dataset with all issues (within a certain range of Status values).

As soon as we've selected this server method in the drop-down list, we can also look at the `Params` property of the `SqlServerMethodGetIssues`. This will show three parameters: `MinStatus`, `MaxStatus` and the `ReturnParameter` (a dataset):



You can define a default value for the MinStatus and MaxStatus parameters, or assign their values in code (which is what the next code snippet will show).

The dsplIssues TDataSetProvider needs to connect his DataSet property to the SqlServerMethodGetIssues.

Next, we need to point the ProviderName property of the cdsIssues TClientDataSet to dsplIssues. Since the resulting dataset cannot be modified, it is a good idea to set the ReadOnly property of the cdsIssues to True, so any data-aware controls connecting to this dataset will not allow the user to make any changes (which is frustrated if you find out later that the changes cannot be saved or send back to the server).

Finally, the dsIssues TDataSource needs its DataSet property assigned to the cdsIssues TClientDataSet, so we can connect data-aware controls to the dsIssues.

Now, on the client main form, we can place a TDBGrid and name it TDBGridReports, to be filled with the reports from this or other server method. In order to put the result of the GetIssues server method, passing explicit values for the MinStatus and MaxStatus filters, we can write the following code:

```
procedure TFormClient.ViewAllIssues1Click(Sender: TObject);  
// Manager - all reports (read-only)  
begin  
    DBGridReports.DataSource := nil;  
    DataModule1.SQLConnection1.Connected := True;  
    try  
        DataModule1.cdsIssues.Close;  
        DataModule1.SqlServerMethodGetIssues.Params.  
            ParamByName('MinStatus').AsInteger := MinStatus;  
        DataModule1.SqlServerMethodGetIssues.Params.  
            ParamByName('MaxStatus').AsInteger := MaxStatus;  
  
        DataModule1.cdsIssues.Open;  
        DBGridReports.DataSource := DataModule1.dsIssues;  
    finally  
        DataModule1.SQLConnection1.Connected := False  
    end
```

```
end;
```

Note that we open the connection to the DataSnap Server using the TSQLConnection component on the data module, we fill the SqlServerMethodGetIssues parameters, call the server method, assign the datasource to the DBGridReport's datasource so we can see the result, and finally we close the connection to the DataSnap Server again. Although you can keep the connection open, disconnecting right after your request will help make the client a bit more robust (you will always start with a fresh connection) although also a bit slower (you always need to open that connection), but the server a bit more scalable (only active connections need to be maintained).

The resulting dataset will contain fields like IssueType, Priority and Status, presenting themselves as integer values. We can map these integer values into more human-friendly string values. For this, we need to double-click on the cdsIssues TClientDataSet on the data module, to start the Fields Editor. Right-click inside the Fields Editor and "Add all fields".

For the IssueType, Priority and Status fields, we can implement the OnGetText event handler to change the simple integer values into more human readable string values, as follows:

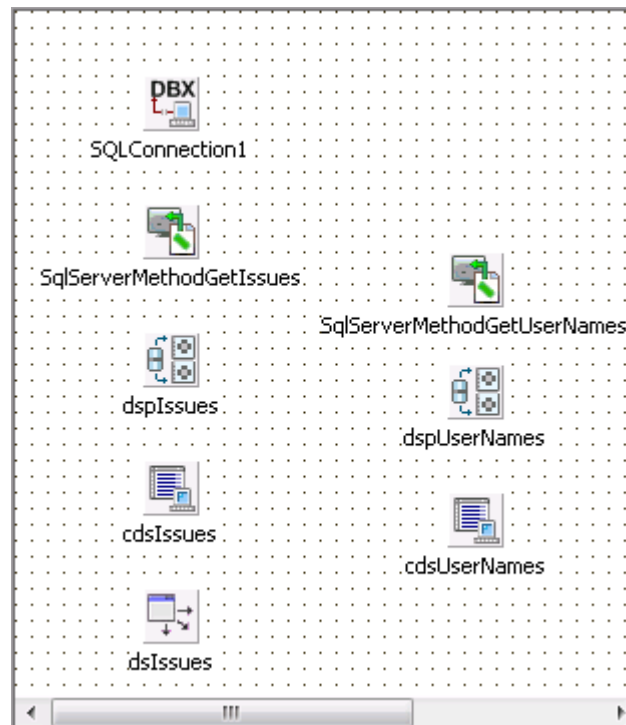
```
procedure TDataModule1.IssueTypeGetText(Sender: TField;  
  var Text: string; DisplayText: Boolean);  
begin  
  case Sender.AsInteger of  
    1..4: Text := 'Minor ' + Sender.AsString;  
    5..8: Text := 'Major ' + Sender.AsString;  
    9..10: Text := 'Critical ' + Sender.AsString;  
  end;  
end;  
  
procedure TDataModule1.PriorityGetText(Sender: TField;  
  var Text: string; DisplayText: Boolean);  
begin  
  case Sender.AsInteger of  
    1..3: Text := 'Low ' + Sender.AsString;  
    4..7: Text := 'Medium ' + Sender.AsString;  
    8..10: Text := 'High ' + Sender.AsString;  
  end;  
end;  
  
procedure TDataModule1.StatusGetText(Sender: TField; var Text: string;  
  DisplayText: Boolean);  
begin  
  case Sender.AsInteger of  
    1: Text := 'Reported';  
    2: Text := 'Assigned';  
    3: Text := 'Opened';
```

```
6: Text := 'Solved';  
7: Text := 'Tested';  
8: Text := 'Deployed';  
10: Text := 'Closed';  
end;  
end;
```

Feel free to add your own translations to these fields, but for our purpose it was enough. Note that the original integer value is still presented in the IssueType and Priority field values as well, as a little reminder what the actual value was.

USERNAMES SERVER METHOD

In a similar way, we can place a TSqlServerMethod component with the sole purpose to call the GetUserNames server method. We should also place a TDataSetProvider component called cdsUserNames and a TClientDataSet component called cdsUserNames to hold the list of user names and UserID values.



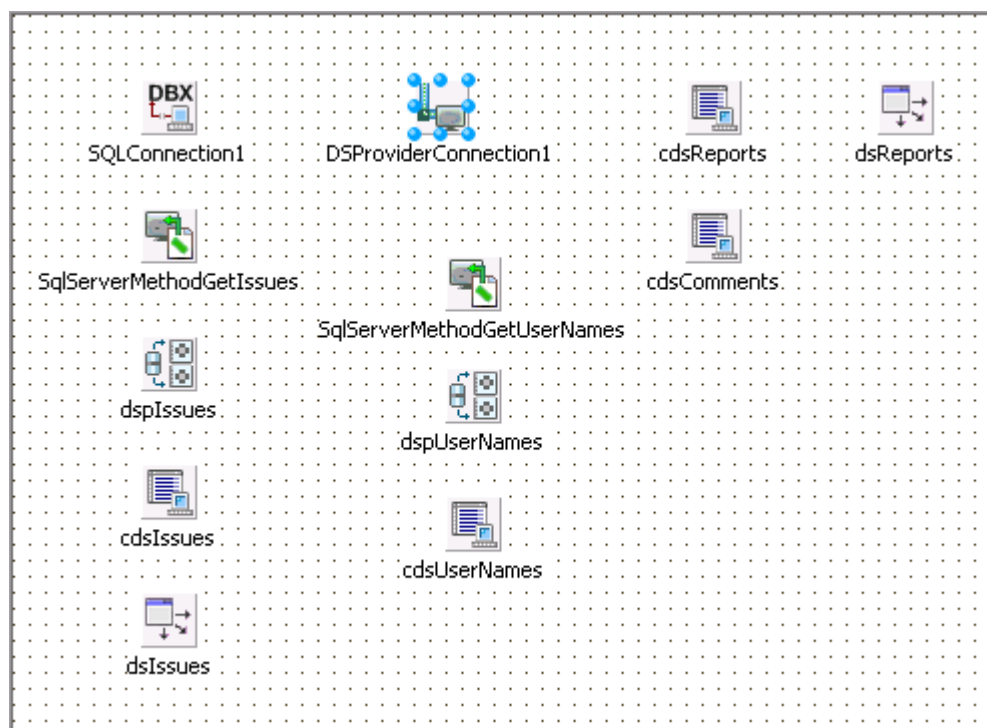
The TSqlServerMethodGetUserNames should be connected to the TSQLConnection component and pointing to the TServerMethods1.GetUserNames server method. The cdsUserNames is the connecting TDataSetProvider that is used by the cdsUserName TClientDataSet to get all these records.

Using the Fields Editor, we can verify that `cdsUserNames` has two fields: `Name` and `UserID`. This dataset will be used as “support” dataset to provide a list of names, or to change a `UserID` value (for example from the `AssignedTo` or `ReportedID` fields) from an ID to a readable name.

DSPROVIDERCONNECTION

Let’s now move to the reported issues for a user with the Developer or Tester role. In that case, we cannot call the `GetIssues` server method, but we should use the exported `TDataSetProvider` called `dspReportUserWithComments` from the server.

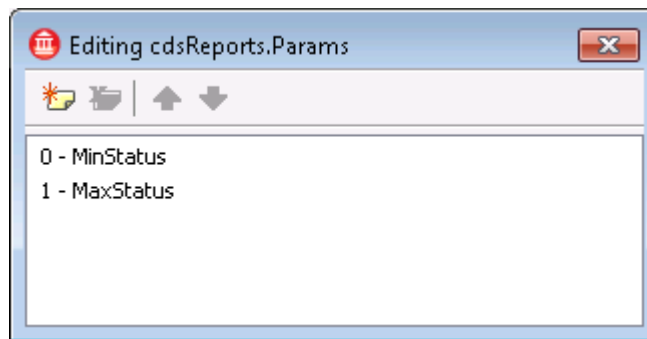
For this, we need to add a `TDSPProviderConnection` component on the data module, as well as two `TClientDataSets` – called `cdsReports` and `cdsComments`, and one `TDataSource` component called `dsReports` connected to the `cdsReports`.



The `DSPProviderConnection` component is the one connecting to a server methods class of the DataSnap Server. In our case, there is only one class: `TServerMethods1`. We must point the `SQLConnection` property of the `DSPProviderConnection` component to the `TSQLConnection`, and then specify the exact name of the server methods class in the `ServerClassName` property. This name is `TServerMethods1` (obviously, you may want to check in your case if the server methods class is called the same).

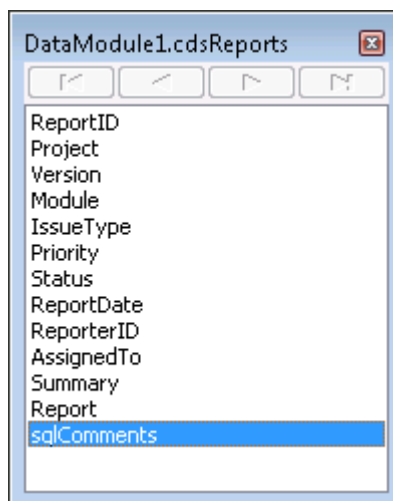
As next step, we should connect the cdsReports TClientDataSet component to the DSProviderConnection, by assigning the later to the RemoteServer property of cdsReports. When everything is setup correct, and the DataSnap Server is running and accepting connections, we can open up the drop-down list for the ProviderName property of the cdsReports and select the dspReportUserWithComments.

As a result, the complete master-detail dataset exported by the dspReportUserWithComments will now be contained in the cdsReports. We now need to configure the fields as well as the parameters. First of all, right-click on the cdsReports and select Fetch Params. This will fill the Params collection of the cdsReports, where we can expect the exported parameters MinStatus and MaxStatus (but not the AssignedTo and ReportedID parameters that we removed from the parameter collection at the server).



You can assign a default value to these parameters, but we'll assign an explicit value to them in code shortly.

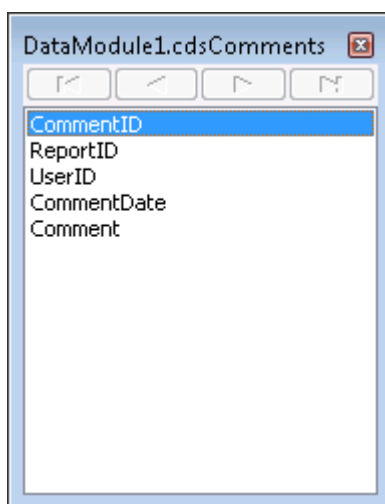
We also need to double-click on cdsReports to start the Fields Editor, and right-click inside the Fields Editor to "Add all fields". This will result in a list of 13 fields with one special field at the end: sqlComments.



This particular field contains the nested detail records, in this case "comment" records, that belong to the master Report record. We can use this field of type TDataSetField to feed the second TClientDataSet called cdsComments.

Select the cdsComments TClientDataSet, and using the Object Inspector, assign cdsReportssqlComments to the DataSetField property. The cdsComments dataset will now automatically contain the detail records for the current master record in the cdsReports dataset.

We should also double-click on the cdsComments TClientDataSet to start the Fields Editor and right-click in it to "Add all fields".

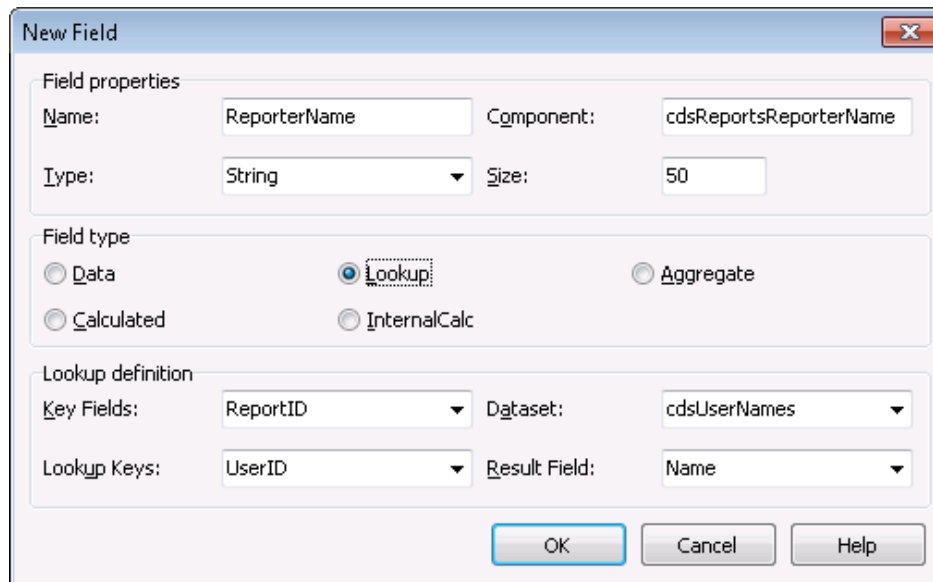


Both TClientDataSets need a little modification with regards to their first field; the primary key. Since we've specified the ReportID (in the Reports table) and the CommentID (in the Comments table) to be autoincrement fields, we should not write these values to the database. The DBMS itself will generate these values for us. As a consequence, for both the cdsReportsReportID field and the cdsCommentsCommentID field, we must edit the ProviderFlags and set the pfInUpdate subflag to False but the pfInKey subflag to True. This will ensure that the TClientDataSets know that the primary key is in fact the key, but that it should not be updated or written to the server.

ADDING LOOKUP FIELDS

Now, go back to the cdsReports and look at the list of fields again. There are a number of fields that need to be translated, like we did earlier for the cdsIssues that were returned for the Manager user. These fields are IssueType, Priority and Status, for which we can share the same OnGetText event handlers as shown before (it's the same data module).

Two other fields may also require translation: the ReportedID and AssignedTo fields, which only contain a UserID value, but not the name of the user. We can use the cdsUserNames table as lookup source to add two lookup fields to cdsReports: one for ReporterName and one for AssignedName. Right-click in the Fields Editor, select "New field...", and define a new lookup field, for example for ReporterName as follows:



We can do the same thing for an AssignedName lookup field for the AssignedTo field in the cdsReports. And don't forget the UserID field in the cdsComments, which could also benefit from a lookup field called UserName.

AUTOINCREMENT FIELDS

A few more steps are required to make the autoincrement fields work, especially in a master-detail relationship. Although we cannot write a value for the ReportID and CommentID primary keys back to the server, we still need to assign a value at the client side to allow us to create new issues or new comments. For these new records, we should use negative key values, starting to count down from -1. This can be implemented using the AfterInsert event of the two TClientDataSets as follows:

```
procedure TDataModule1.cdsReportsAfterInsert(DataSet: TDataSet);  
begin  
    cdsReportsReportID.AsInteger := -1;  
end;  
  
procedure TDataModule1.cdsReportsOrCommentsAfterPostOrDelete(DataSet: TDataSet);  
begin  
    cdsReports.ApplyUpdates(-1)  
end;
```

Note that the code here simply assigns -1 and does not attempt to count further down. Counting further down is not needed, since the client application will send all updates back to the server right after a post or delete. This means that we will never have more than one (currently being edited) record with a possible negative key value. Right after the post (after insert or edit), the new record will be send to the server, and all we need to do is refresh the contents of the cdsReports and cdsComments in order to retrieve the actual autoincrement values from the server.

For that purpose, we need to implement the AfterDelete and AfterPost event handlers of both TClientDataSets. The good news is that all four event handlers can be shared in one, since in all cases there's just one thing that we need to do:

```
procedure TDataModule1.cdsReportsOrCommentsAfterPostOrDelete(DataSet: TDataSet);
begin
    cdsReports.ApplyUpdates(0);
    cdsReports.Refresh
end;
```

This will ensure that we call the ApplyUpdates to send any changes to the server, and right after that call the Refresh method to refresh the data from the server.

Refresh will not just refresh the primary key values, but will actually refresh all data in the records that we have selected. This may sound like a bad idea, and it may certainly be unwise if you select a large number of records, but in this case, the exported DataSetProvider will only return the Reports (with Comments) that belong to the current user, within the specified MinStatus and MaxStatus ranges, so if that produces thousands and thousands of records, then perhaps this user is either a fanatic when it comes to testing, or a developer who has too much work on his or her plate...

Viewing the issues, for a user with the Developer or Tester role, can now be implemented as follows:

```
procedure TFormClient.ViewMyIssues1Click(Sender: TObject);
// Developer/Tester personal reports
begin
    DBGridReports.DataSource := nil;
    DataModule1.SQLConnection1.Connected := True;
    try
        DataModule1.cdsReports.Close;
        DataModule1.cdsReports.Params.ParamByName('MinStatus').AsInteger :=
            MinStatus;
        DataModule1.cdsReports.Params.ParamByName('MaxStatus').AsInteger :=
            MaxStatus;
        DataModule1.cdsReports.Open;
        DBGridReports.DataSource := DataModule1.dsReports;

        // nested dataset
```



```
    DBGridReports.Columns[DBGridReports.Columns.Count-1].Visible := False;  
  finally  
    DataModule1.SQLConnection1.Connected := False  
  end  
end;  
end;
```

The last column in the TDBGrid is the nested dataset. We can either set the Visible property of that field to false in the Fields Editor of the cdsReports, or set Visible to false for the last column in the TDBGrid. The code above implements the latter choice.

Note that apart from that nested field column, the code to display the reported issues is very similar to what was needed for the Manager's call to the GetIssues server method, exposed by the SqlServerMethodGetIssues. However, where the Manager's result is a read-only dataset (actually set to read-only using the property ReadOnly), the Developer and Tester's result is a dataset where we can make modifications.

For these modifications, however, we do not want users to edit data in a grid, so a new form was designed to allow editing and updating of individual issue reports.

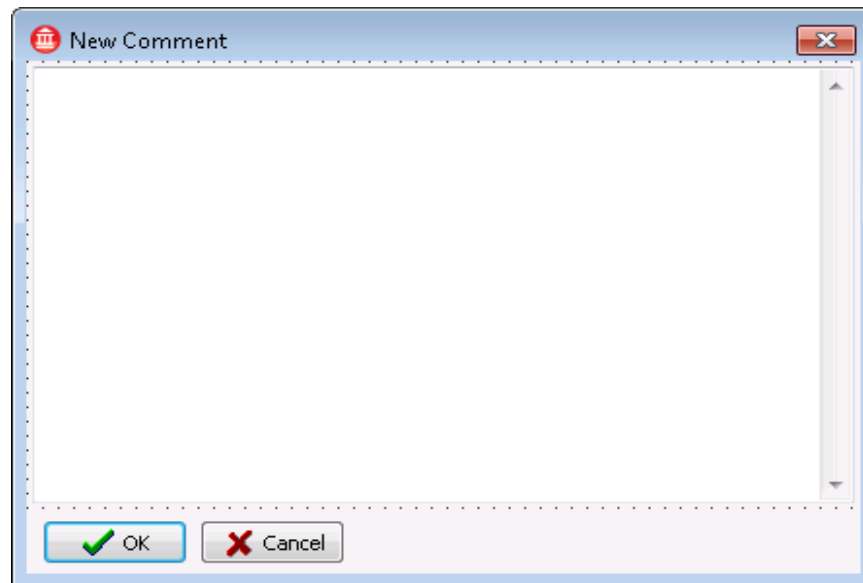
REPORTED ISSUE FORM

Building the Reported Issue Form now consists of nothing more than placing two TDataSources (called dsComments and dsReports) and a number of data-aware controls to display the contents of the Report master and Comments detail records. For our purpose, the form has the following layout, but feel free to add your own look and feel of course:

The comments are displayed in a TDBGrid, with the current comment details shown in the second TDBMemo. Adding a comment should not be done in the grid (although this is possible), but is nicer in a new form. For that, we've designed a special form to enter the new comments, and display it as follows:

```
procedure TFormIssue.btnNewCommentClick(Sender: TObject);
begin
    with TFormComments.Create(Self) do
        try
            ShowModal;
        finally
            Free;
        end
    end;
end;
```

The new form itself also has a very simple layout, using two buttons and a TMemo with the vertical scrollbar option enabled.



Note that only the actual comments are needed to insert a new comment record, since the client application knows the (current) report we want to comment on, it knows the current user, as well as the date, so the only field that needs “manual” input is the actual comment itself.

As a result, the implementation of the OK button is as follows:

```
procedure TFormComments.BitBtn1Click(Sender: TObject);  
begin  
    DataModule1.cdsComments.Insert;  
    DataModule1.cdsComments.CommentID.AsInteger := -1;  
    DataModule1.cdsComments.UserID.AsInteger := UserID;  
    DataModule1.cdsComments.CommentDate.AsDateTime := Now;  
    DataModule1.cdsComments.Comment.AsString := Memo1.Text;  
    DataModule1.cdsComments.Post;  
    Close  
end;
```

This will also close the dialog, returning the user back to the overview with refreshed contents of the reports and comments.

REPORT NEW ISSUE

There is one more action left to explore: reporting new issues. For this feature, we can call the ReportNewIssue server method manually, passing the Project, Version, Module, IssueType, Priority, Summary, Report and optionally AssignedTo arguments. This is not hard to do, using simple input controls and a direct call to the server method.

However, we can also use the existing cdsReports TClientDataSet, connected to the DSProviderConnection to simply insert a new record in the cdsReports table, with -1 assigned to the primary key (automatically), and using data-aware controls to design the GUI. Thereby ignoring the ReportNewIssue server method (which was only allowed for Testers anyway), and using the dataset provider functionality which was enabled for both Testers and Developers.

The data-aware Report New Issue form is defined as follows, with a single TDataSource connected to the cdsReports on the data module, and a set of data-aware controls.

During the FormCreate, we can do an Insert of the cdsReports dataset, assigning the UserID to ReporterID, the Status as reported (value 1), as well as the ReportDate (as Now). The ReporterID, Status and ReportDate fields are made read-only in this form.

The OK button will do a Post and close the form, while the Cancel button will do a Cancel and close the form. Implemented as follows:

```
procedure TFormNewIssue.FormCreate(Sender: TObject);
begin
    dsReports.DataSet.Open;
    dsReports.DataSet.Insert;
    dsReports.DataSet.FieldByName('ReporterID').AsInteger := UserID;
    dsReports.DataSet.FieldByName('Status').AsInteger := 1;
    dsReports.DataSet.FieldByName('ReportDate').AsDateTime := Date;
end;
```

```

procedure TFormNewIssue.btnOKClick(Sender: TObject);
begin
    dsReports.DataSet.Post;
    Close
end;

procedure TFormNewIssue.btnCancelClick(Sender: TObject);
begin
    dsReports.DataSet.Cancel;
    Close
end;

```

Note that the IssueType, Priority, and AssignedTo fields are assigned using simple TDBEdit controls. It would be a good idea to use a numerical spinedit control for the IssueTypes and Priorities (limiting the input to a range of 1..10), and a drop-down combobox showing the known users for the AssignedTo field. Since the AssignedTo field is not required, we cannot use a TDBLookupComboBox component (which would force us to always make a choice, even if we do not want to assign the issue report to a user, yet). So, instead I've used a regular TCombobox filled with the names of the users in the FormCreate event, with the following additional code:

```

DataModule1.cdsUserNames.Open; // in case it was closed
DataModule1.cdsUserNames.First;
cbAssignedTo.Items.Clear;
cbAssignedTo.Items.Add('Nobody');

while not DataModule1.cdsUserNames.Eof do
begin
    // current user name
    if DataModule1.cdsUserNames.UserID.AsInteger = UserID then
        edReporter.Text := DataModule1.cdsUserNames.Name.AsString;

    cbAssignedTo.Items.AddObject(
        DataModule1.cdsUserNames.Name.AsString,
        TUserID.Create(DataModule1.cdsUserNames.UserID.AsInteger));

    DataModule1.cdsUserNames.Next;
end;

```

The TUserID class is a special support class that I've created to hold the UserID so we can add this object as second argument to the AddObject method of the ComboBox's Items.

```

type
    TUserID = class
        UserID: Integer;
        constructor Create(const NewUserID: Integer);
    end;

constructor TUserID.Create(const NewUserID: Integer);
begin

```

```
inherited Create;  
  UserID := NewUserID  
end;
```

To avoid a memory leak, we should also cleanup the objects from the TComboBox again when the form is closed, which can be done as follows in the FormClose:

```
procedure TFormNewIssue.FormClose(Sender: TObject;  
  var Action: TCloseAction);  
var  
  i: Integer;  
begin  
  for i:=0 to cbAssignedTo.Items.Count-1 do  
    if Assigned(cbAssignedTo.Items.Objects[i]) then  
      cbAssignedTo.Items.Objects[i].Free  
  end;
```

Note that we're also assigning the current user's name to the Text property of a TEdit called edReporter, so we see the name of the person who submitted the report (which should be our own name).

This results in the following design of the Report New Issue form:

The screenshot shows a Windows-style dialog box titled "Report New Issue". It contains the following controls:

- Project:** A text box containing "DBEditProject".
- Version:** A text box containing "DBEditVersion".
- Module:** A text box containing "DBEditModule".
- IssueType:** A TSpinEdit control showing the value "1".
- Priority:** A TSpinEdit control showing the value "1".
- Status:** A text box containing "DBEditStatus".
- ReportDate:** A text box containing "DBEditReportDate".
- Reporter:** A text box containing "edReporter".
- Assigned To:** A dropdown menu.
- Summary:** A text box containing "DBEditSummary".
- Report:** A large memo area containing "DBMemoReport".

In the center of the form, there is a data source icon labeled "dsReports". At the bottom right, there are two buttons: "OK" (with a green checkmark) and "Cancel" (with a red X).

Note that the IssueType and Priority TSpinEdit controls have a default value of 1, and a MinValue of 1 and MaxValue of 10. Feel free to change the default Value to, for example, 5 if you want, to make it easier to report more urgent issues.

As final optional extension, we could also add TComboBox controls for the Project, Version and Module input fields, retrieving a list of existing values for the Project, Version and Module fields in the current dataset. That, however, is left as a final exercise for the reader.

CLIENT DEPLOYMENT

If we add the MidasLib unit to the uses clause of the DataSnap Client project, then we end up with an almost stand-alone executable. No database drivers are needed. However, since the ISAPI DLL uses HTTPS, and the TCP/IP stand-alone server uses the RSA and PC1 filters, we must also deploy the libeay32.dll and ssleay32.dll files with our DataSnap Client—or make sure they already exist at the client machine.

The DirtCleaner executable and two SSL support DLLs require an internet connection to connect to the DirtServer, but apart from that, nothing else is needed. A valid user name and password is all that's needed to connect to the server and participate in the role of Manager, Developer or Tester. The source code for the DirtServer and DirtCleaner projects is available for download. A more refined version will be made available and described in my DataSnap XE courseware manual (and can be used to report omissions or typo/bug reports in my courseware manuals for registered readers).

SUMMARY

In this DataSnap XE in Action paper, I've described how to design and implement a small but real-world secure DataSnap Server application, and deployed it on Microsoft's Internet Information Services as an ISAPI DLL. Security was implemented using authentication and authorization as well as HTTPS (or RSA/PC1 filters for a stand-alone server) for a secure transport channel. I also showed how to connect to the DataSnap Server from a DataSnap Client in order to call server methods and work with exported DataSetProviders.

Among the technical issues covered when working with datasets, I explained how to work with autoincrement fields, especially in combination with master-detail and nested datasets.

This application would not have been possible without the many new features and enhancements in found in DataSnap XE. With the release of Delphi XE, DataSnap was again substantially expanded and enhanced compared to DataSnap 2010 or 2009, and a lot improved since the COM-based original versions of DataSnap and MIDAS.

REFERENCES

RAD Studio in Action – DataSnap 2010 white paper

<http://www.embarcadero.com/rad-in-action/datasnap>

Delphi XE DataSnap Development courseware manual

<http://www.ebob42.com/courseware/>

ABOUT THE AUTHOR

Bob Swart (aka Dr.Bob) is an IT consultant, developer, reseller, author, trainer and webmaster for his company Bob Swart Training & Consultancy (eBob42) based in Helmond, The Netherlands. Bob has spoken at Delphi Conferences since 1993. Bob is co-author of several books and contributing author for numerous computer magazines. His courseware manuals are sold word-wide from Lulu.com or via his own website (including e-mail support and free updates). Bob leads the Delphi section of the Dutch Software Development Network, and is webmaster and member of the UK Developers Group.

Website: <http://www.drbob42.com>

Email: Bob@eBob42.com

LinkedIN: <http://www.linkedin.com/in/drbob42>



Embarcadero Technologies, Inc. is the leading provider of software tools that empower application developers and data management professionals to design, build, and run applications and databases more efficiently in heterogeneous IT environments. Over 90 of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero's award-winning products to optimize costs, streamline compliance, and accelerate development and innovation. Founded in 1993, Embarcadero is headquartered in San Francisco with offices located around the world. Embarcadero is online at www.embarcadero.com.