

# MIDAS components

By Dave Oldfield

With the decision by Inprise to reduce the licence fee for MIDAS 3, there has been a resurgence of interest in this technology. However, there are some issues in using it, one of which is that MIDAS 3 can be stateless. In fact, I think it is safer to assume it is always stateless - it uses the best advantages of the technology and the client software is immune to modifications to the application server if techniques such as pooling or MTS are used. Being stateless means that, every time the client wishes to download a new packet of records, the current state of the client must be first transferred to the application server. In this article, I will show how this can be done and present some simple components to automate the process. Some familiarity with MIDAS is assumed.

To transfer the state from the client to the application server, or vice versa, Delphi provides two events, `BeforeGetRecords` and `AfterGetRecords`, both of them on the client dataset and provider. These events are declared to be of type `TRemoteEvent`, which is defined as:

```
procedure(Sender: TObject; var OwnerData:
           OleVariant) of object
```

Any state information to be transferred can be encoded into the `OwnerData` parameter. As an `OleVariant` virtually any data can be passed. Therefore, when the client dataset needs to fetch data from the application server, the following sequence occurs.

1. The client dataset calls a `BeforeGetRecords` event, where it provides the state information as the `OwnerData` parameter. For example, you could assign the value of the last record downloaded.
2. The provider on the application server also has a `BeforeGetRecords` event, where it can respond to, or change, the state information before it creates the data packet. So, if `OwnerData` contains the last record downloaded, the provider's dataset could move to this record, like so:

```
if not VarIsEmpty(OwnerData) then
  Dataset.Locate(KeyField, OwnerData, []);
```

3. After creating the data packet, the provider then calls the `AfterGetRecords` event, where it can encode new state information into the `OwnerData` parameter to be returned back the client dataset. For example, to assign the value of the last record downloaded:

```
if (KeyField <> '') then OwnerData :=
  Dataset.FieldValues[Self.KeyField];
```

4. The client dataset receives an `AfterGetRecords` event, where it can respond to the state information returned by the provider, probably by storing the value ready for the next time data is to be downloaded.

Although the above sequence is quite straightforward, with many client datasets in an application writing two events for each, and two for their corresponding providers, soon gets tedious. A better solution is to write some components

that automate the process. Obviously, two components are needed - a `TClientDataset` and a `TDataSetProvider` descendant - here I have called them `TStatusClientDataset` and `TStatusDataSetProvider`.

As always, when deriving new components from those in Delphi, the first step is to look at the VCL source code to ascertain what events or methods we can override. Looking at the code for a `TClientDataset` (in `DBClient.pas`) there is a virtual protected method `DoGetRecords` which calls the two `Before / After GetRecord` event handlers. We could implement these two events and assign them to the event properties at run-time in the component's constructor, as in the following:

```
constructor TStatusClientDataSet.Create(AOwner:
                                       TComponent);
begin
  inherited;
  BeforeGetRecords := DoBeforeGetRecords;
  AfterGetRecords := DoAfterGetRecords;
end;
```

The problem with doing this is that we lose the advantage of implementing these events at design-time. For instance, you may wish to display which records have been downloaded or to override the automatically produced state information for some reason. What we need to do is store a reference to any design-time implementation of these events and call them in our run-time implementation. In the new constructor below, `OldBeforeGetRecords` is a new private variable that stores a reference to the design-time instance of `BeforeGetRecords`. Then we point `BeforeGetRecords` at the new procedure, `DoBeforeGetRecords`, which is where we will provide the state information. Similarly for `AfterGetRecords`.

```
constructor TStatusClientDataSet.Create(AOwner:
                                       TComponent);
begin
  inherited;
  OldBeforeGetRecords := BeforeGetRecords;
  BeforeGetRecords := DoBeforeGetRecords;
  OldAfterGetRecords := AfterGetRecords;
  AfterGetRecords := DoAfterGetRecords;
end;
```

We can implement our new event `DoBeforeGetRecords` as in the following listing. After assigning the state information to the `OwnerData` parameter the stored reference to the original `BeforeGetRecords` is called if it has been assigned. The `DoAfterGetRecords` method is implemented in a similar way.

```
procedure
TStatusClientDataSet.DoBeforeGetRecords(Sender:
                                       TObject; var OwnerData: OleVariant);
begin
  OwnerData := StateData;
  if Assigned(OldBeforeGetRecords) then
    OldBeforeGetRecords(Sender, OwnerData);
end;
```

To implement the TStatusDatasetProvider is roughly the same process but with a slight change. There is a public method in an ancestor class, TCustomProvider, called GetRecords which calls the Before and After get record events. But instead of calling BeforeGetRecords directly, there is a virtual method DoBeforeGetRecords which calls it. As this is a virtual method, we might as well as override it. The AfterGetRecords event needs to be handled as we did in the client dataset, by storing a reference to the design time event. Bringing all this together results in the two components below. Note there is a public property in TStatusClientDataSet called StateData which stores the state information. Similarly, in TStatusDatasetProvider there is a published property, KeyField, which is the field name to be used to locate the next record to be downloaded. Remember to add DbClient and Provider to the Uses clause of the unit.

This is a very basic implementation of these components. For your own applications, you may need to transfer more state information than just a key value. Extra state information you may wish to transfer could be a session id that was returned to the client when logging on. Also, choosing the key field name of TStatusdatasetProvider could be done so that it automatically chooses the primary key. But, the techniques presented here are a good place to start.

```

type
  TStatusClientDataSet = class(TClientDataSet)
  private
    FStateData: OleVariant;
    OldBeforeGetRecords: TRemoteEvent;
    OldAfterGetRecords : TRemoteEvent;
    procedure DoBeforeGetRecords(Sender:
      TObject; var OwnerData: OleVariant);
    procedure DoAfterGetRecords(Sender:
      TObject; var OwnerData: OleVariant);
  public
    constructor Create(AOwner: TComponent);
      override;
    property StateData: OleVariant read
      FStateData write FStateData;
  end;

  TStatusDatasetProvider=class(TDatasetProvider)
  private
    FKeyField: string;
    OldAfterGetRecords: TRemoteEvent;
    procedure DoAfterGetRecords(Sender:
      TObject; var OwnerData: OleVariant);
  protected
    procedure DoBeforeGetRecords(Count:
      Integer; Options: Integer;
      const CommandText: WideString; var
      Params, OwnerData: OleVariant); override;
  public
    constructor Create(aOwner: TComponent);
      override;
  published
    property KeyField: string read FKeyField
      write FKeyField;
  end;

  procedure Register;
  implementation

  constructor TStatusClientDataSet.Create(AOwner:
    TComponent);
  begin
    inherited;
    FStateData:= Unassigned;
    {Intercept..}

```

```

    OldBeforeGetRecords:= BeforeGetRecords;
    BeforeGetRecords:= DoBeforeGetRecords;
    OldAfterGetRecords:= AfterGetRecords;
    AfterGetRecords:= DoAfterGetRecords;
  end;

  procedure
  TStatusClientDataSet.DoBeforeGetRecords(Sender:
    TObject; var OwnerData: OleVariant);
  begin
    OwnerData:= StateData;
    {Opportunity to handle..}
    if Assigned(OldBeforeGetRecords) then
      OldBeforeGetRecords(Sender, OwnerData);
  end;

  procedure
  TStatusClientDataSet.DoAfterGetRecords(Sender:
    TObject; var OwnerData: OleVariant);
  begin
    {Opportunity to handle..}
    if Assigned(OldAfterGetRecords) then
      OldAfterGetRecords(Self, OwnerData);
    StateData:= OwnerData;
  end;

  constructor
  TStatusDatasetProvider.Create(aOwner:
    TComponent);
  begin
    inherited;
    {Intercept..}
    OldAfterGetRecords:= AfterGetRecords;
    AfterGetRecords:= DoAfterGetRecords;
  end;

  procedure
  TStatusDatasetProvider.DoBeforeGetRecords(Count,
    Options: Integer;const CommandText:
    WideString; var Params, OwnerData:
    OleVariant);
  begin
    inherited; //..calls BeforeGetRecords
    if Dataset.Active
      and not VarIsEmpty(OwnerData) then
      Dataset.Locate(Self.KeyField,OwnerData,[]);
  end;

  procedure
  TStatusDatasetProvider.DoAfterGetRecords(Sender:
    TObject; var OwnerData: OleVariant);
  begin
    if Dataset.Active and (Self.KeyField <> '')
      then
      OwnerData:=
      Dataset.FieldValues[Self.KeyField];
    {Opportunity to handle..}
    if Assigned(OldAfterGetRecords) then
      OldAfterGetRecords(Self, OwnerData);
  end;

  procedure Register;
  begin
    RegisterComponents('Midas',[TStatusClientDataSet,
      TStatusDatasetProvider]);
  end;

```

---

*Dave is a Delphi/InterBase developer interested in contract work. He has recently returned to his native parts but, although Sheffield-based, is still a regular at BUG London meetings - come and see him present MIDAS on November 21st - and is happy to work in the South. You can contact him on 0114 235 6571 or [dave@infersys.demon.co.uk](mailto:dave@infersys.demon.co.uk).*