Technical Note

Rep++ Serialization: Implementing the Undo Command

Overview

Rep++ RowsetTree and Rowset objects support the serialization/deserialization process of their content to or from a number of formats. This capability can be used to save the content of a RowsetTree or Rowset in a file or buffer for later retrieval. The serialized content is perfectly safe for interprocess and even internetwork communications. As a result, it can be retrieved and restored (i.e. deserialized) in the same or in a different application, which may or may not run on the same computer.

The capability to serialize and deserialize Rep++ objects has created a number of possible uses:

- Use Rep++ objects as Web services' parameters.
- Save data to and load data from the file system directly.
- Create standard data structures to send or receive a Rep++ RowsetTree or Rowset to or from an application that is not Rep++-aware.
- Keep and restore different snapshots of data to implement an Undo command.

This article describes how to use the Rep++ serialization/deserialization capability to implement a simple Undo command. Applies to **Rep++ 8**.

Rep++ Serialization: Implementing the Undo Command

Serializing and deserializing Rep++ objects

You can serialize and deserialize RowsetTree and Rowset objects in a buffer or in a file using a binary or XML format. The objects provide the following serialization/deserialization methods.

Method	Description
LoadFromBuf	Loads the content of a RowsetTree or Rowset object from a binary byte array.
LoadFromFile	Loads the content of a RowsetTree or Rowset object from a binary file.
LoadXMLFromBuf	Loads the content of a RowsetTree or Rowset object from a string containing XML markup.
LoadXMLFromFile	Loads the content of a RowsetTree or Rowset object from an XML file.
SaveToBuffer	Saves the content of a RowsetTree or Rowset object to a byte array in binary format.
SaveToFile	Saves the content of a RowsetTree or Rowset object to a file in binary format.
SaveXMLToBuf	Saves the content of a RowsetTree or Rowset object to a string in XML format.
SaveXMLToFile	Saves the content of a RowsetTree or Rowset object to an XML file.

The binary byte array is a proprietary format for storing Rep++ data structures. The following parameters can be used when serializing/deserializing a binary format (please see the Rep++ documentation in the Visual Studio® Help Viewer for details):

- **Compress** — When this parameter is not specified, its default value is true.
  - Set it to true when serializing to indicate that the binary content should be compressed.
  - Set it to true when deserializing to indicate that the binary content is compressed and thus needs to be decompressed before the deserialization.
- **ChangedOnly** — true to serialize only the new, deleted or modified lines. When this parameter is not specified, its default value is false.
- **LangNeutral** — true to serialize or deserialize content in a language-independent format. The meaning of this property is whether to serialize or deserialize the internal (neutral) or external (end user formated) field values. When this parameter is not specified, its default value is false.

The following parameters can be used when serializing/deserializing the XML format (the set of parameters depend on the object and method):

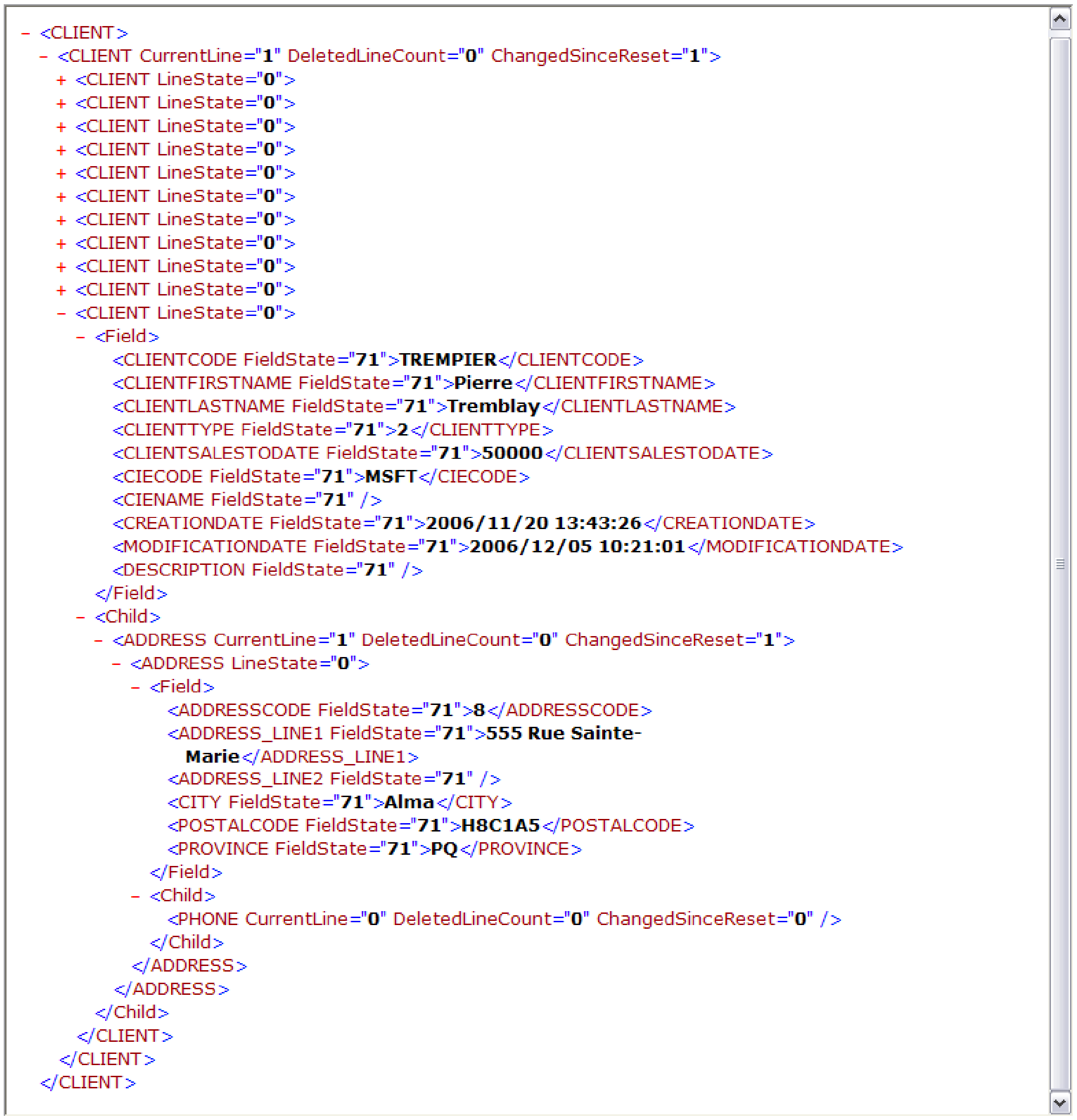
- **ChangedOnly** — true to serialize only the new, deleted or modified lines. When this parameter is not specified, its default value is false.
- **RowsetTreePrefix** — String prefix for the XML element of a RowsetTreeDef.

- **RowsetPrefix** — String prefix for the XML element of a RowsetDef.
- **FieldPrefix** — String prefix for the XML element of a field.
- **Flags** — Serialization options.

Important

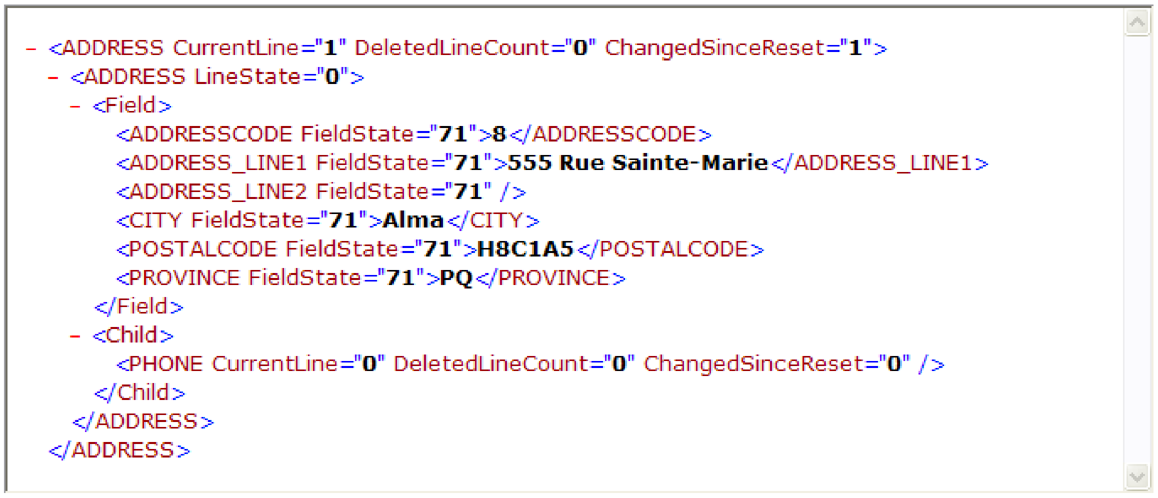
Please note that serializing or deserializing using the XML format is slower than using the binary format.

Here is a sample of an XML-serialized RowsetTree containing the list of clients in the contact management demo application.



```
- <CLIENT>
  - <CLIENT CurrentLine="1" DeletedLineCount="0" ChangedSinceReset="1">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    + <CLIENT LineState="0">
    - <CLIENT LineState="0">
      - <Field>
          <CLIENTCODE FieldState="71">TREMPIER</CLIENTCODE>
          <CLIENTFIRSTNAME FieldState="71">Pierre</CLIENTFIRSTNAME>
          <CLIENTLASTNAME FieldState="71">Tremblay</CLIENTLASTNAME>
          <CLIENTTYPE FieldState="71">2</CLIENTTYPE>
          <CLIENTSALESTODATE FieldState="71">50000</CLIENTSALESTODATE>
          <CIECODE FieldState="71">MSFT</CIECODE>
          <CIENAME FieldState="71" />
          <CREATIONDATE FieldState="71">2006/11/20 13:43:26</CREATIONDATE>
          <MODIFICATIONDATE FieldState="71">2006/12/05 10:21:01</MODIFICATIONDATE>
          <DESCRIPTION FieldState="71" />
        </Field>
      - <Child>
        - <ADDRESS CurrentLine="1" DeletedLineCount="0" ChangedSinceReset="1">
          - <ADDRESS LineState="0">
            - <Field>
                <ADDRESSCODE FieldState="71">8</ADDRESSCODE>
                <ADDRESS_LINE1 FieldState="71">555 Rue Sainte-
                  Marie</ADDRESS_LINE1>
                <ADDRESS_LINE2 FieldState="71" />
                <CITY FieldState="71">Alma</CITY>
                <POSTALCODE FieldState="71">H8C1A5</POSTALCODE>
                <PROVINCE FieldState="71">PQ</PROVINCE>
              </Field>
            - <Child>
                <PHONE CurrentLine="0" DeletedLineCount="0" ChangedSinceReset="0" />
              </Child>
            </ADDRESS>
          </ADDRESS>
        </Child>
      </CLIENT>
    </CLIENT>
  </CLIENT>
```

The following Figure is a sample content of an XML-serialized Rowset that contains the address of the expanded client from the previous Figure.



```
- <ADDRESS CurrentLine="1" DeletedLineCount="0" ChangedSinceReset="1">
  - <ADDRESS LineState="0">
    - <Field>
        <ADDRESSCODE FieldState="71">8</ADDRESSCODE>
        <ADDRESS_LINE1 FieldState="71">555 Rue Sainte-Marie</ADDRESS_LINE1>
        <ADDRESS_LINE2 FieldState="71" />
        <CITY FieldState="71">Alma</CITY>
        <POSTALCODE FieldState="71">H8C1A5</POSTALCODE>
        <PROVINCE FieldState="71">PQ</PROVINCE>
      </Field>
    - <Child>
        <PHONE CurrentLine="0" DeletedLineCount="0" ChangedSinceReset="0" />
      </Child>
    </ADDRESS>
  </ADDRESS>
```

Implementing a simple Undo command

Undoing changes is a standard feature found in almost all commercial applications. The Undo command allows the end user to ignore recent changes and restore an older state. The principle is quite simple:

1. Take a snapshot of the original state.
2. Let the user make changes.
3. According to end user, do one:
    - If the end user clicks Save, then overwrite the snapshot of the original state with a snapshot of the current state.
    - If the end user clicks Undo, then overwrite the snapshot of the current state with the snapshot of the original state.

As you can see, the Undo command relies heavily on the presence of a mechanism to take and restore snapshots of an application's state. This can be achieved using the Rep++ serialization and deserialization capability. Rep++ enables you to:

1. Take a snapshot of a RowsetTree or Rowset by serializing its content to a buffer or file.
2. Restore a snapshot by deserializing a buffer or a file into a RowsetTree or Rowset.

The task of taking and restoring snapshots boils down to the following two methods:

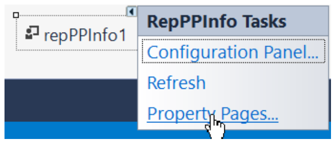
```
private bool TakeSnapShot(RepPP.RowsetTree rsTree, out Byte[] arrSnapshot) {
  bool    bResult;

  arrSnapshot = null;
  bResult = (rsTree.SaveToBuf(out arrSnapshot) ==
              (int)RepPP.ErrorCode.sdNoErr);
  return(bResult);
}

private bool RestoreSnapShot(RepPP.RowsetTree rsTree, Byte[] arrSnapshot) {
  bool bResult = false;

  if (arrSnapshot != null && arrSnapshot.Length != 0) {
    rsTree.Empty();
    bResult = (rsTree.LoadFromBuf(arrSnapshot) ==
                 (int)RepPP.ErrorCode.sdNoErr);
  }
  return(bResult);
}
```

To illustrate the implementation of your simple Undo command, you will use the contact management demo system to build a Windows Forms application that will:

1. Load the data.
2. Take a snapshot.
3. Change the data. The application will do the following:
   - Modify a client.
   - Delete a client.
   - Add an address for a client.
4. Undo your changes by restoring the snapshot taken in step 2.

Creating a standard Windows® Forms project connected to Rep++

1. Create a standard Windows® project.
2. Add a reference to RepPP.dll (the Rep++ class library for .NET).
3. From the toolbox, drop a Rep++ Toolkit 8.0 RepPPInfo control on the form.
4. Open the **RepPPInfo Tasks** menu and click **Property Page**.



5. Select the connection and system for the Democlient program.

Implementing the Undo command

1. Add the TakeSnapShot and RestoreSnapShot methods to the code of the default form.
2. Add the DisplayData helper method that will be used to display the state of the data at different stages:

```
private void DisplayData(string strTitle, RepPP.RowsetTree rsTreeClients) {
    RepPP.Application app;
    RepPP.Rowset       rowsetClients;
    RepPP.Rowset       rowsetAddresses;
    RepPP.Field        fldFirstName;
    string             strMessage;

    app = rsTreeClients.Application;
    rowsetClients = rsTreeClients.RootRowset;
    strMessage    = "The number of clients: ";
    strMessage   += rowsetClients.UndelLineCount;
    fldFirstName  = rowsetClients.Fields["ClientFirstName"];
    strMessage   += "\nThe first name of the first client: ";
    strMessage   += fldFirstName.GetValue(0, true);
    rowsetClients.SelectLine(1, true, true);
    rowsetAddresses = app.RowsetDefs["Address"].ActiveRowset;
    strMessage     += "\nThe number of addresses of the second client: ";
    strMessage     += rowsetAddresses.UndelLineCount;
    MessageBox.Show(strMessage, strTitle);
}
```

3. Add a button to the default form and handle its click event to create the Rep++ application object and load the list of clients:

```
private void button1_Click(object sender, EventArgs e) {
  RepPP.Application   app;
  RepPP.RowsetTreeDef rsTreeDef;
  RepPP.RowsetTree    rsTreeClients;
  RepPP.Rowset        rowsetClients;
  RepPP.Field         fldFirstName;

  using (app = RepPP.Application.CreateFromRes()) {
      rsTreeDef = app.RowsetTreeDefs["CLIENT"];
      rsTreeDef.BuildSqlCommand();
      using (rsTreeClients = rsTreeDef.RowsetTrees.Create()) {
          rsTreeClients.ReadFromDb();
          rowsetClients = rsTreeClients.RootRowset;
          fldFirstName  = rowsetClients.Fields["CLIENTFIRSTNAME"];
          DisplayData("Original data", rsTreeClients);
    }
  }
}
```

4. Add the following code (in bold) to simulate different types of data modification:

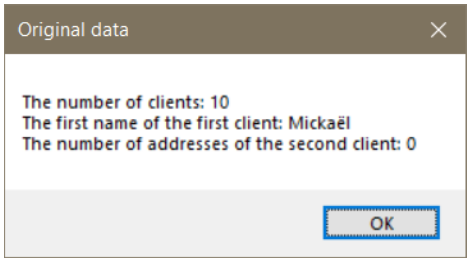
```
private void button1_Click(object sender, EventArgs e) {
  RepPP.Application   app;
  RepPP.RowsetTreeDef rsTreeDef;
  RepPP.RowsetTree    rsTreeClients;
  RepPP.Rowset        rowsetClients;
  RepPP.Rowset        rowsetAddresses;
  RepPP.Field         fldFirstName;

  using (app = RepPP.Application.CreateFromRes()) {
      rsTreeDef = app.RowsetTreeDefs["CLIENT"];
      rsTreeDef.BuildSqlCommand();
      using (rsTreeClients = rsTreeDef.RowsetTrees.Create()) {
          rsTreeClients.ReadFromDb();
          rowsetClients = rsTreeClients.RootRowset;
          fldFirstName  = rowsetClients.Fields["CLIENTFIRSTNAME"];
          DisplayData("Original data", rsTreeClients);

          // Modify a Client
          fldFirstName.SetValue(0, true, "Modified");
          // Delete a Client
          rowsetClients.SelectLine(4);
          rowsetClients.DeleteLine();
          // Add an Address
          rowsetClients.SelectLine(1, true, false);
          rowsetAddresses = app.RowsetDefs["Address"].ActiveRowset;
          rowsetAddresses.AddBlankLine();
          DisplayData("Modified data", rsTreeClients);
```

```
    }
  }
}
```

5. To simulate the implementation of an Undo command, take a snapshot of the original data (i.e. right after reading it from the database) and then restore the snapshot of the original data to ignore any intermediate changes:

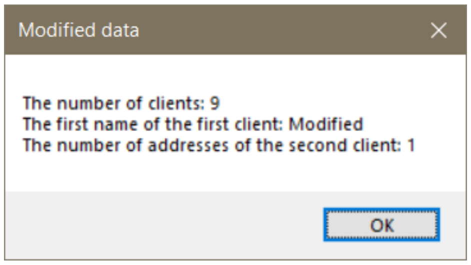
```
private void button1_Click(object sender, EventArgs e) {
  RepPP.Application   app;
  RepPP.RowsetTreeDef rsTreeDef;
  RepPP.RowsetTree    rsTreeClients;
  RepPP.Rowset        rowsetClients;
  RepPP.Rowset        rowsetAddresses;
  RepPP.Field         fldFirstName;
  Byte[]              arrSnapshot;

  using (app = RepPP.Application.CreateFromRes()) {
      rsTreeDef = app.RowsetTreeDefs["CLIENT"];
      rsTreeDef.BuildSqlCommand();

      using (rsTreeClients = rsTreeDef.RowsetTrees.Create()) {
          rsTreeClients.ReadFromDb();
          rowsetClients = rsTreeClients.RootRowset;
          fldFirstName  = rowsetClients.Fields["CLIENTFIRSTNAME"];
          DisplayData("Original data", rsTreeClients);

          // Take Client Snapshot
          if (!TakeSnapShot(rsTreeClients, out arrSnapshot)) {
              throw new ApplicationException("Cannot take Snapshot!");
          }

          // Modify a Client
          fldFirstName.SetValue(0, true, "Modified");
          // Delete a Client
          rowsetClients.SelectLine(4);
          rowsetClients.DeleteLine();
          // Add an Address
          rowsetClients.SelectLine(1, true, false);
          rowsetAddresses = app.RowsetDefs["Address"].ActiveRowset;
          rowsetAddresses.AddBlankLine();
          DisplayData("Modified data", rsTreeClients);

          // Restoring Client Snapshot
          if (!RestoreSnapShot(rsTreeClients, arrSnapshot)) {
              throw new ApplicationException("Cannot restore Snapshot!");
          }
          DisplayData("Restored data", rsTreeClients);
        }
    }
}
```

6. Build the project and run the application. The following three message boxes indicate:
   - The original data.



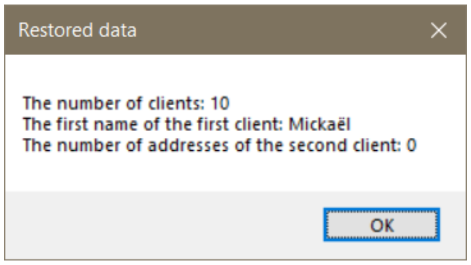
```
The number of clients: 10
The first name of the first client: Mickaël
The number of addresses of the second client: 0
```

- The modified data.



```
The number of clients: 9
The first name of the first client: Modified
The number of addresses of the second client: 1
```

- The restored data (to confirm that the Undo command was performed successfully).



```
The number of clients: 10
The first name of the first client: Mickaël
The number of addresses of the second client: 0
```

Advanced Undo commands

The previous section described the basic building blocks of an Undo command. The same basic building blocks will allow you to develop a number of sophisticated Undo commands. For example:

- You can take a snapshot after every change the user makes and thus implement a text editor-like Undo command.
- You can also take and restore a partial snapshot of data. For example, you can implement separate Undo commands for clients and addresses by using Rowset serialization instead of RowsetTree serialization.