Under Construction: Delphi 4 And Java

by Bob Swart and Hubert Klein Ikkink

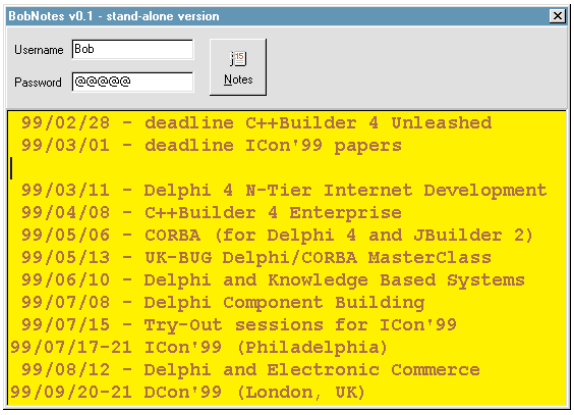
Last month we explored CORBA, and I claimed that CORBA was a good platform-independent and language-independent method of communication. This time, we'll focus specifically on protocols for communicating between Delphi 4 and Java, starting with CGI and sockets. Next month we'll return to CORBA again.

BobNotes

First, let's define the 'business case' for this article. In these busy times, I can no longer rely solely on my memory to keep appointments, deadlines and my sanity at the same time. Instead, I find myself making notes on small pieces of paper (and losing them almost immediately). Since I'm at least *near* a computer at all times (if not on top of one), why not make use of this potential and create an application that can present my latest list of *Things To Do* at all times?

Given that I have at least four 'working' machines (one at work, one at home, one in the attic and one in the garage), I need a way to share my *To Do* notes on all the machines. And the only network they have in common is, of course, the internet.

The standalone version of the *To Do* notes, which I have called BobNotes, can be seen in Figure 1. I could use it to store diary notes like deadlines, dates for masterclasses, etc.



➤ *Figure 1: Standalone version of BobNotes v0.1.*

So, this month, we'll not only focus on a Delphi application, but also implement a small Java Applet to login, show and/or update simple text (my *To Do* notes). The Java Applet, running inside any JDK 1.1 compliant internet browser (like Netscape Navigator 4.x or Internet Explorer 4.x), communicates with the Delphi server-side application that stores the *To Do* notes for every user.

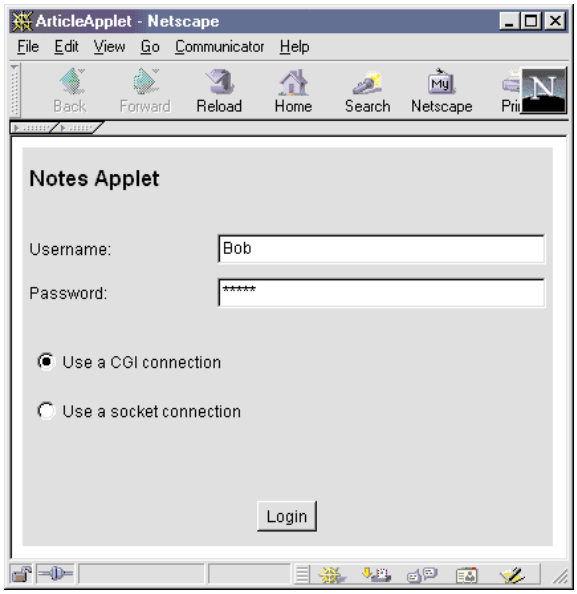
Mr.Haki's Java Applet

The Java Applet is made by Hubert A Klein Ikkink (aka Mr.Haki) using JBuilder 2, and requires a JDK 1.1 compliant browser. Java Applets cannot store anything on the current (local) client machine, so they rely on server-side applications as storage devices for our *To Do* notes.

The Applet consists of basically three different screens: a login screen, a screen to show the notes and a screen to add new notes. So when a user loads the HTML page with the Java Applet he or she must first log in using a username and password. Besides filling in the username and password the user can also choose the communication protocol to be used by the Applet. Today we will look at communication between the Java Applet and server-side Delphi applications with CGI and with sockets (so we won't see the actual implementation of the Java Applet: email Hubert at hubert@bolesian.nl if you want a copy of the Applet source code).

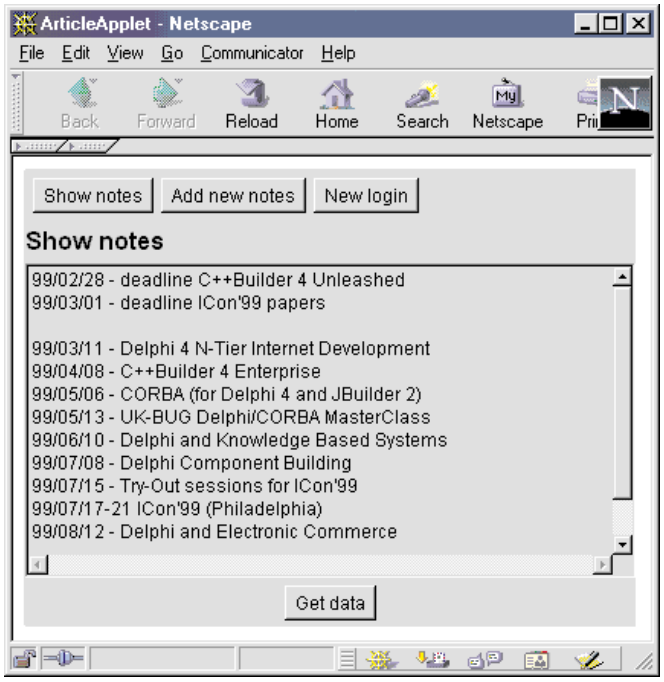
After selecting the communication protocol the user can get notes from the server or add new notes and save them to the server. Figures 3 and 4 show the two options available. Figure 4 shows the result after pressing the `Get data` button.

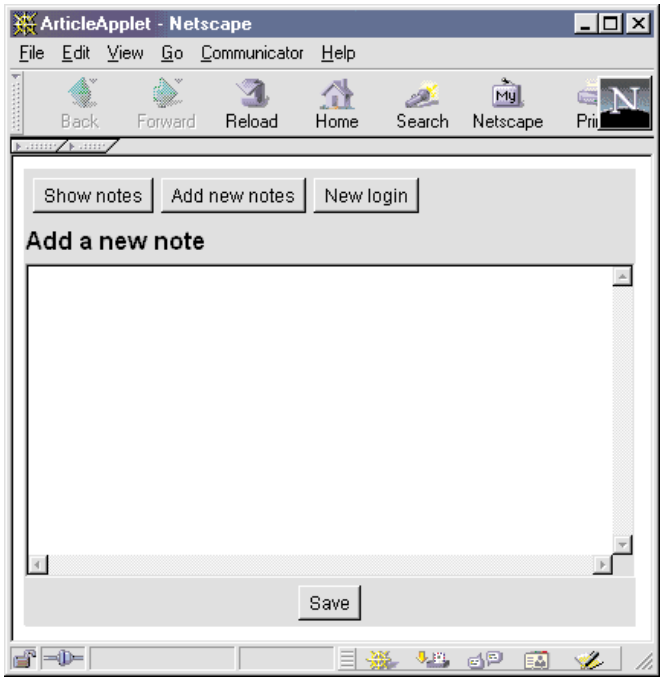
➤ *Figure 2*



Dr.Bob's Delphi List

Now that we have a Java client Applet that can show (and update) *To Do* notes for a given person, it's time to focus on the Delphi server-side application that stores the individual notes. For this, we only need a simple database setup with a set of strings for each user. Since I know that not everyone has access to a WinNT web server that includes the BDE, we will not implement the Delphi program using the BDE, but instead build a simpler implementation using .INI files. The layout of the .INI file is the same for each user:





➤ *Above: Figure 3*

➤ *Right: Figure 4*

```
[bob]
password=swart
lines=3
1=Write Under Construction #44
2=Finish 2 C++Builder 4
  Unleashed chapters
3=Download Delphi 4
  Update Pack #3
```

The reason why I use a password field is of course to guard my *To Do* notes against illegal updates by someone else. The `lines` key contains the number of lines in the list, which are numbered from 1 to the value of `lines`. A pretty easy format, but it is sufficient for a fair amount of users each of which might have about a dozen *To Do* notes.

Two base routines are needed. One called `GetLines` to get the *To Do* notes from the INI file, and one `SetLines`, to assign new *To Do* notes and write them to the INI file. In both cases, the username and password need to be passed as arguments too (and the username and password should be correct, of course). The two routines are implemented in a unit called `IniMod`, which can be seen as a replacement DataModule or even WebModule (but then based on an INI file, and only the two `Get/Set` APIs). See Listing 1 for the details.

Communication

Now that we have the Java client Applet and the Delphi server-side application, it's time to focus on the communication between the two. An easy way to communicate is the Common Gateway Interface (CGI) protocol where the Java Applet calls the Delphi server-side CGI application, passing data on the URL using the `GET` protocol, and interprets the results that are returned.

Alternatively, we can use TCP/IP to open a socket between the Delphi server and Java Applet, and communicate directly between the two.

Finally, last time we showed that CORBA can be used as communication protocol, which would mean (in this case) transforming the Delphi server into a CORBA server implementing CORBA methods (and the Java Applet into a CORBA client, calling these

➤ *Listing 1: IniMod storage unit.*

```
unit IniMod;
interface
uses SysUtils, Classes;
type
  ELoginFailed = class(Exception);
  procedure GetLines(const User, Passw: String; Lines:
    TStrings); // raises ELoginFailed
  procedure SetLines(const User,Passw: String; Lines:
    TStrings); // raises ELoginFailed
implementation
uses IniFiles;
var  IniFile: TIniFile = nil;
procedure GetLines(const User,Passw: String;
  Lines: TStrings);
var i: Integer;
begin
  if (User <> '') and (IniFile.ReadString(User, 'password',
    '') = Passw) then begin
    Lines.Clear;
    for i:=1 to IniFile.ReadInteger(User,'lines',0) do
      Lines.Add(IniFile.ReadString(User,IntToStr(i),''))
  end else
    raise ELoginFailed.Create('Login failed.')
end {GetLines};
procedure SetLines(const User,Passw: String;
  Lines: TStrings);
var i: Integer;
begin
  if (User <> '') and (IniFile.ReadString(User, 'password',
    '') = Passw) then begin
    IniFile.EraseSection(User);
    { reset password }
    IniFile.WriteString(User,'password',Passw);
    { linescount }
    IniFile.WriteInteger(User,'lines',Lines.Count);
    for i:=1 to Lines.Count do
      IniFile.WriteString(User,IntToStr(i),Lines[Pred(i)])
  end else
    raise ELoginFailed.Create('Permission denied.')
end {SetLines};
initialization
  IniFile := TIniFile.Create('.\BobNotes.ini');
finalization
  IniFile.Free;
  IniFile := nil
end.
```

```
program CGI;
{$APPTYPE CONSOLE}
uses DrBobCGI, SysUtils, Classes, IniMod;
var
  Notes: TStringList = nil;
  User, Password: String;
  Get: Boolean;
  i: Integer;
begin
  writeln('Content-type: text/plain');
  writeln;
  User := Value('User');
  Password := Value('Password');
  Notes := TStringList.Create;
  Get := Value('Notes') = '';
  if not Get then
    Notes.Text := Value('Notes');
  try
    try
      if Get then begin
        GetLines(User,Password,Notes);
        for i:=1 to Notes.Count do writeln(Notes[Pred(i)])
      end else begin
        SetLines(User,Password,Notes);
        writeln('OK')
      end
    except
      on E: Exception do writeln(E.Message)
    end
  finally
    Notes.Free
  end
end.
```

➤ *Listing 2: Delphi CGI Server.*

methods). In theory, using JMIDAS for JBuilder 2, we can also decide to use DCOM (MIDAS) instead of CORBA. CORBA (and possibly DCOM) connections between Delphi and JBuilder applications will be covered next time; for now we focus on 'low-level' CGI and sockets instead.

Delphi CGI

The Delphi server-side application can be called with three possible arguments: `User=`, `Password=` and `Notes=`. The Java Applet can call the Delphi CGI application in two ways. If only the username and password are passed, then the current *To Do* notes are returned. If, as well as `User` and `Password`, the `Notes` variable is also passed, then the *To Do* notes on the server are updated with the value of `Notes`. In both cases, an error will be returned if the `Password` doesn't match the `User` (and no, there's still no way to set or reset a given password).

The Delphi CGI server application is implemented in Listing 2, using the unit `DrBobCGI`.

Note that the content-type is set to `text/plain`, and the actual content is either `OK` (after we've set the new *To Do* notes), or the current list of *To Do* notes: one item on each line.

The Delphi CGI application can be called directly by the Java Applet as a URL call, like this:

```
http://www.drbob42.com/cgi-bin/
  CGI.exe?User=Bob&Password=swart
```

which, since no argument `Notes` is passed, will return the current *To Do* notes for user `Bob` with password `swart`.

```
private URL constructURL(String user, String pwd, String lines)
  throws MalformedURLException
{
  // Create a StringBuffer for a String-like representation of the URL
  StringBuffer constructURL = new StringBuffer();
  // Add the name of the CGI app
  constructURL.append("http://www.drbob42.com/cgi-bin/CGI.exe");
  // Start with the parameter part (or GET part) of the CCI app
  constructURL.append("?");
  // Add the username part
  constructURL.append("User");
  constructURL.append("=");
  // The URLEncoder.encode() method takes care of formatting variables so
  // they are ready to be sent to the CGI (replacing spaces with %20, etc.)
  constructURL.append(URLEncoder.encode(user));
  // Add the password part
  constructURL.append("&");
  constructURL.append("Password");
  constructURL.append("=");
  constructURL.append(URLEncoder.encode(pwd));
  // Adding the lines part (if not null, else we are already done)
  if (lines != null)
  {
    constructURL.append("&");
    constructURL.append("Notes");
    constructURL.append("=");
    constructURL.append(URLEncoder.encode(lines));
  }
  // Return the new URL object
  return new URL(constructURL.toString());
}
```

➤ *Listing 3: Java constructURL*

Java CGI

To communicate with the CGI application from the Java Applet, we must be able to construct the URL shown above for both getting and setting notes on the server. Luckily Java already contains a URL class. This class is capable of storing a URL and eventually opens a connection to this URL using streams. So what we have to do right now is construct a URL object, which will incorporate the CGI executable, and add the username, password and (optionally) new notes as parameters of this CGI executable. Listing 3 contains the Java method `constructURL` to take care of this.

The method `constructURL` takes three arguments: `username`, `password` and `lines`. These three arguments are used to construct the URL object. If the `lines` argument contains no value (is null) we do not add that part to the URL. A nice utility method we are using is the `URLEnocder.encode()` method. This method is capable of formatting any `String` object to a MIME format called `x-www-form-urlencoded`. This way we can be sure what we are sending can be understood by the CGI.exe. The CGI application will not know the difference between accessing the application through a simple browser or through our Java Applet.

Once we have constructed the URL it is time to get information from the URL. We can use the `openStream()` method of the URL class to open a stream to the URL. This means we are opening a one-way 'tunnel' from the CGI application to our Applet. Any output the CGI application will create will be sent directly to the Applet. And in our Java Applet we can collect the data and do with it what we want. For example show the notes (being information sent by the CGI) to the user. The code in Listing 4 takes care of opening the

```
private String sendAndReceive(String user, String pwd,
  String lines)
  throws ConnectionException
{
  String result = null;
  // Input stream reader for connection to CGI app
  BufferedReader in = null;
  try {
    // Create a URL object based on user and pwd
    URL url = constructURL(user, pwd, lines);
    // Open a stream connection to the CGI application
    in = new BufferedReader(new InputStreamReader(
      url.openStream()));
    // StringBuffer to store data received from the CGI app
    StringBuffer buf = new StringBuffer();
    // Line contains a single line received from the CGI app
    String line;
    // Loop for receiving data from the CGI app until no
    // more data available
    while ((line = in.readLine()) != null)
    { // Store the received data in the StringBuffer
      buf.append(line);
      // Add an end-of-line
      buf.append("\r\n");
    }
    // Convert to received data to the result String
    result = buf.toString();
  }
  catch (MalformedURLException mfuex) {
    // The URL object isn't correct, so we throw
    // a new ConnectionException
    throw new ConnectionException(mfuex.getMessage());
  }
  catch (IOException ioex) {
    // Error reading from CGI app, so we throw
    // a new ConnectionException
    throw new ConnectionException(ioex.getMessage());
  }
  finally {
    // Closing the input stream reader
    if (in != null) {
      try {
        in.close();
      }
      catch (IOException ioex) {
        // Don't handle it
      }
    }
  }
  return result;
}
```

➤ *Listing 4: Java CGI Client*

stream to the CGI application and reading data from it.

Two things are important to notice. First of all we wrap the stream from the URL in a `BufferedReader` object. The `BufferedReader` object is optimised for reading character data and because we are using only character data here it is a good thing to use this class. The second important thing is the loop structure for reading in the data from the stream. We are executing a `readLine()` method (which returns everything up to an end of line symbol) repeatedly until the response is null. In other words up to the end of the stream.

To invoke this method from our Applet is quite easy. If we want to get data from the server we will invoke this method with the username and password of the user and leave the `lines` argument null. And to add new notes to the server we can invoke the same method, only this time we will use the `lines` argument for the new notes, see Listing 5.

Sockets

A CGI application is only activated for each request, and gets terminated right after it has serviced the request. Of course, an ISAPI DLL remains in memory, but is still basically a stateless protocol. Sockets enable the client and the server to open a communication 'channel' (called socket) and keep it open for as long as necessary.

```
// Reading notes from the CGI app
String fromURL = sendAndReceive("Bob", "swart", null);
// Sending new notes to the CGI app
sendAndReceive("Bob", "swart", "99/02/29 - Nonday");
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
Element                    Description          Text
-----------------------------------------------------------
First element:             Command              PUT, GET
Second element:            Username             username
Third element:             Password             password
Fourth element: (optional) Notes                the notes
```

But before we see how to use sockets for data communication between two different machines, it is maybe a good idea to first explain what sockets are. Basically a socket is a peer-to-peer communication endpoint. This endpoint has a network address and a port number. Sockets communicate using TCP/IP and this means we can use sockets also over the internet. A socket address consists of two parts: an IP address (or name) and a port number. The IP address is the address of the machine where the socket will be set up. For example on our machine this IP address would be 127.0.0.1, or we can use the name *localhost*. A port is the entry point of the application. A 16-bit number represents the port number. Well known services already use up some of the port numbers available on a machine, like an HTTP web server that uses port 80 to listen for requests. If we are developing applications with sockets on our own we don't want to use these reserved port numbers. So how do we know what port numbers can be used and can't be used? The answer can be found within the SERVICES file in our Windows folder. This file contains a list of port numbers which are reserved and shouldn't be used by us, to avoid conflicts.

A pitfall to be aware of when using sockets is the existence of firewalls. Firewalls can be set up to only allow requests for certain socket port numbers, like port 80 for a web server. This could jeopardise the communication of our own developed applications, because the firewall will not allow the use of other port numbers. So we need to keep this in mind when programming with sockets.

Sockets give us total freedom to communicate between different applications on different machines, but in order for the different applications to understand what is being sent it is important to define a protocol for the communication. This protocol describes what elements will be sent, in which order, and what to expect in return. In the CGI example we already used a pre-defined protocol to communicate: HTTP. We will define (update) a new protocol for

our application so the server can distinguish between a GET data request and an ADD data request, see Listing 6.

The different elements are separated by a semi-colon (;) and are sent as a single string of characters.

Delphi Server Socket

The Delphi TServerSocket component (from the Internet tab) can be used to manage multiple client connections (like the Java Applet). A Socket Server application must be 'in the air' at all times, so we'll create an empty form with a TMemo component (to show debug lines of text) and a single TServerSocket component. The latter has a few properties that must be set at design-time. It's important to specify the port number, which is set to 4242 in our case. As ServerType, we can specify either stNonBlocking or stThreadBlocking. The latter spawns a new thread for each socket connection accepted by the server socket. We'll use stNonBlocking to handle all reading and writing over the socket connections asynchronously. Client connections are handled in a single execution thread. OnClientRead and OnClientWrite events are called when the client socket tries to send or receive data.

Apart from OnClientRead and OnClientWrite, we'll also implement the OnAccept, OnClientConnect and OnClientDisconnect event handlers to be able to log when a client connection is accepted by our Socket Server, when a connection is made and when a client is disconnected again, respectively. The OnClientRead event handler is the most interesting (see Listing 7), as this is the one receiving the client request, parsing the Socket.ReceiveText and sending back the results (like OK or the contents of the *To Do* notes) using the Socket.SendText method. Note that at the end of the last Socket.SendText we need to send an END_OF_TRANSMISSION token, so the (Java) client knows that no more lines (from the *To Do* list) are to be expected.

Java Client Socket

Java contains a Socket class we can use to set up socket connection. The constructor of the Socket class takes two arguments: name of the host machine and the port number. The Delphi server application will be running on a machine called www.drbob42.com and will use port number 4242. So we can use these values as arguments of our socket creation. After we have created the socket we create input and output streams with the sockets. So we are creating a unique 'tunnel' between the Java Applet and the Delphi server-side application. We can use this tunnel to send back and forth data. And this tunnel will exist as long as both the client-side socket and the server-side sockets are open.

Listing 8 shows the code that makes sure we open a socket connection.

We saw earlier that we had to define a protocol to communicate

➤ *Listing 7: Delphi Socket Server.*

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ScktComp, StdCtrls;
type
  TForm1 = class(TForm)
    ServerSocket1: TServerSocket;
    Memo1: TMemo;
    procedure ServerSocket1Accept(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure ServerSocket1ClientConnect(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure ServerSocket1ClientDisconnect(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure ServerSocket1ClientWrite(Sender: TObject;
      Socket: TCustomWinSocket);
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
uses IniMod;
procedure TForm1.ServerSocket1Accept(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add('Client accepted');
end;
procedure TForm1.ServerSocket1ClientConnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add('Client connected');
end;
procedure TForm1.ServerSocket1ClientDisconnect(Sender:
  TObject; Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add('Client disconnected');
end;
procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
const
  Sep = ';';
var
  i: Integer;
  Str: String;
  Lines: TStringList;
  User,Password: ShortString;
begin
  Memo1.Lines.Add('Client read');
  Str := Socket.ReceiveText;
  Lines := TStringList.Create;
  try
    Memo1.Lines.Add(Str);
    if Pos('PUT',Str) = 1 then begin
      { PUT }
      Delete(Str,1,4);
      User := Copy(Str,1,Pos(Sep,Str)-1);
      Delete(Str,1,Pos(Sep,Str));
      Password := Copy(Str,1,Pos(Sep,Str)-1);
      Delete(Str,1,Pos(Sep,Str));
      Lines.Text := Str;
      try
        IniMod.SetLines(User,Password,Lines);
        Lines.Text := 'OK'
      except
        on E:Exception do
          Lines.Text := E.Message
      end
    end else begin
      { GET }
      Delete(Str,1,4);
      User := Copy(Str,1,Pos(Sep,Str)-1);
      Delete(Str,1,Pos(Sep,Str));
      Password := Str; { strip CR/LF ?? }
      Str := ''; { no notes }
      try
        IniMod.GetLines(User,Password,Lines)
      except
        on E:Exception do
          Lines.Text := E.Message
      end
    end; { GET }
    for i:=0 to Pred(Lines.Count) do
      Socket.SendText(Lines[i] + #13#10);
    Socket.SendText('END_OF_TRANSMISSION' + #13#10);
  finally
    Lines.Free
  end
end;
procedure TForm1.ServerSocket1ClientWrite(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add('Client write');
end;
end.
```

```
Socket socket;
BufferedReader in;
PrintWriter out;
try
{
  // Open a socket on www.drbob42.com, port 4242
  socket = new Socket("www.drbob42.com", 4242);
  // Create the input and output streams
  in = new BufferedReader(new inputStreamReader(socket.getInputStream()));
  out = new PrintWriter(socket.getOutputStream());
}
catch (IOException ioex)
{
  // Something went wrong, throw ConnectionException object
  throw new ConnectionException(ioex.getMessage());
}
```

➤ *Listing 8: Java Socket Connection.*

between the client and server over the socket. So our method responsible for communicating must construct and use the protocol. We will notice some similarities between this method code and the one we used for the CGI communication, see Listing 9.

When the protocol is created we will first check if we want to get notes from the server or want to add notes to the server by checking the lines argument. If this argument is null, we want to get data else we want to put data on the server. This command is the first element of the protocol. Next follows the proposed delimiter, a semi-colon. Then the two elements username and password are added (each separated by the delimiter). And finally, if we are putting new data on the server, we add this data as the last element.

The protocol we construct is then sent to the server application with the `out.print()` method. Once the data is sent we can wait for the response from the server. We will read in data from the server until we reach the string `END_OF_TRANSMISSION`, which denotes the end of the data.

To use this method for communication we invoke it the same way we did for the CGI communication, see Listing 10.

Stuffing...

There's one potential problem with sockets. In the `OnClientRead` event handler of the Delphi Socket Server, our code expects to receive the complete request all in one go. However, for a `PUT` with more than a few different *To Do* items, we might not get everything in one packet, and will need to listen to two or more `OnClientRead` events. This is not implemented at this time (take a closer look at my `TBSMTP` and `TBPOP3` components discussed in Issues 35 and 36 if you want to see some `TClientSocket` implementations that use this technique).

Apart from this problem, there's one more catch using sockets on WinNT, and that's the fact that the ScktSrvr application (from Delphi4\Bin) must be running before you can do any communication using sockets. Just something to remember in case it doesn't work right away...

Conclusions

We've seen that we can communicate between Delphi (Server) Applications and Java (Client) Applets in different ways, using the standard CGI protocol or using sockets. The underlying architecture of Delphi and Java is quite similar, as only the implementation details differ.

➤ *Listing 10*

```
// Reading notes from the socket app
String fromURL = sendAndReceive("Bob", "swart", null);
// Sending new notes to the socket app
sendAndReceive("Bob", "swart", "99/02/29 - Nonday");
```

➤ *Listing 9: Java Socket Client.*

```
private String sendAndReceive(String user, String pwd,
  String lines)
    throws ConnectionException
{
  // String to store result
  String result = null;
  try
  {
    // Create the "protocol" to be sent to the server
    StringBuffer protocol = new StringBuffer();
    // If lines is null we only interested in getting notes,
    // else we are adding new notes
    if (lines != null)
    {
      protocol.append("PUT");
    }
    else
    {
      protocol.append("GET");
    }
    // The delimiter used between elements
    protocol.append(";");
    // Username
    protocol.append(user);
    protocol.append(";");
    // Password
    protocol.append(pwd);
    // We are adding new notes so add those to the protocol
    if (lines != null)
    {
      protocol.append(";");
      protocol.append(lines);
    }
    // Send protocol to the server
    out.print(protocol);
    out.flush();
    // Read response from the server.
    // If we are getting notes this will contain notes, else
    // a status code.
    StringBuffer buf = new StringBuffer();
    String line = in.readLine();
    boolean reading = true;
    while (reading)
    {
      // If end of transmission reached we will leave the
      // loop
      if (line.equalsIgnoreCase("END_OF_TRANSMISSION")) {
        reading = false;
      // Else we continue to add String to the result
      }
      else
      {
        buf.append(line);
        buf.append("\r\n");
        // Read next line
        line = in.readLine();
      }
    }
    result = buf.toString();
  }
  catch (IOException ioex)
  { // Error in reading and writing to socket
    throw new ConnectionException(ioex.getMessage());
  }
  return result;
}
```

Next Time

Next time, we take BobNotes a little bit further, and wrap it up as a Delphi CORBA Server Object. We will then generate a Java CORBA Client Object to communicate with the Delphi Server, and cover the pitfalls along the way. Finally, we'll tell a little bit more about the DCOM for Java bridge called JMIDAS, *so stay tuned...*

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a technical consultant and webmaster using Delphi, JBuilder and C++Builder for Bolesian and a freelance technical author.

Hubert A Klein Ikkink (aka Mr.Haki) has been using Java and JBuilder since they were invented and has wide commercial experience in the development and deployment of Java applications. Hubert is the webmaster for Mr.Haki's JBuilder Machine at www.drbob42.com/JBuilder. He writes and speaks regularly on Java and JBuilder.