

Remote server communication and application delivery

Alan Newman

Revision 1.0, December 2004

1. Introduction

This article wraps up the previous two articles (*Creating MIDlets for Multiple UI* and *Game development across the MIDP versions*) covering the development of “DemoVaders” and consolidates the application.

This final article in the “DemoVaders” series looks at providing a network element to the application so that high scores can be sent to and retrieved from a remote server. Such features create an opportunity for a feeling of community, by allowing users to store their scores on a server and then compare them to their peers.

We shall also look at how the MIDlet would be delivered to an end-user, by describing the configuration of the server. Not only is this an important method of generating a revenue stream, but it also gives us the ability to test over-the-air (OTA) installations locally before submitting the application to a commercial distribution partner.

Finally we shall outline the “Java Verified” process (www.javaverified.com), an application-testing framework that provides a unified testing standard across the industry. Java Verified helps strengthen the trust between the developer, the distributor and the consumer. We shall explain why developers might want to submit their applications to Java Verified as a means of creating new revenue streams

2. MIDlet networking

2.1. Overview

The DemoVaders application provides us with a perfect reason to communicate with a remote server. The game generates scores, which the user may want to submit to a high-score table. This could help form some sort of community and allow the user to compare their performance to that of others. Typically network operator portals may offer this facility to users.

In the same way we separated the display of the data from the core application logic in the two previous papers, we have again adopted a similar model. The submission and retrieval of scores to and from the server is broken down in to three `MIDlet` classes and one server-side Java servlet:

- 1) `Score.java` – is the main worker class for remote server communication. This provides the methods with which the score UI classes interact with the server.
- 2) `NameEntryForm.java` - is a simple high-level `lcdui` class that captures the user’s name and score at the end of the game’s only level. This data is then passed on to the remote server via the `Score.java` class.

- 3) `HighScoreCanvas.java` - as its name suggests, is a low-level `lcdui` class. It is a `Runnable` canvas that retrieves the high score data from the server and then draws it to the display.
- 4) `HighScores.java` - is a simple `Servlet` class operating on the remote server. Its job is to wait for `HTTP` requests from the `MIDlet` and then return the high scores when requested over `HTTP`.

The MIDP 2.0 specification extends the network communication defined in MIDP 1.0. HTTP 1.1 networking was defined as mandatory in MIDP 1.0. MIDP 2.0 makes mandatory HTTPS networking (SSL). The use of low-level IP networking was optional for MIDP 1.0 implementations, where as for MIDP 2.0 this has been specified as a recommended practice. Symbian OS v7.0s phones provide implementations of sockets, server sockets, secure sockets and datagram protocols. All these protocols can be accessed using the Generic Connection Framework (GCF), which was first defined under CLDC 1.0.

To demonstrate how the `HttpConnection` works, our application will examine how a developer might submit a `POST` to a remote web server, and then retrieve information from that server using a `GET` request.

2.2. Code explanation

In the first instance we aim to submit a user's score at the end of the only level of our sample application `DemoVaders`. This is achieved by creating an `HTTP POST` command. The score is submitted to a Java servlet installed within Tomcat on the remote server. This value is stored along with the user's name in memory within an `ArrayList`.



Figure 2.1

This `POST` is instigated when the user enters their name in an `lcdui` form, which appears at the end of the level. When the user presses the `OK` button, a thread is started within the `HighScoreCanvas.java` class that opens an `HttpConnection` to the server. The `POST` is made and the score is stored in the server.

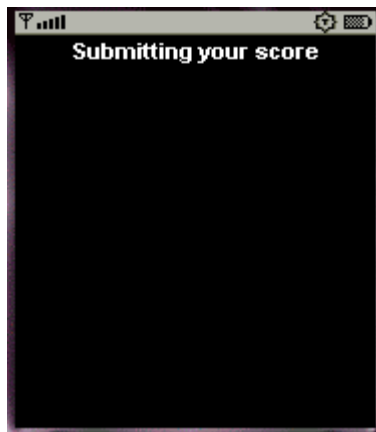


Figure 2.2

In the meantime the user is notified of this process by the display of a message on screen.



Figure 2.3

Once the score has been submitted, the thread makes a request to the servlet's GET method and retrieves the scores stored in memory. These values are loaded by the MIDlet, which reads the servlet's `OutputStream` as an `InputStream`. It then stores those values in memory and draws them to the display.

Figure 2.4

2.2.1. Sending a POST request to the remote web server

The `sendHttpRequest()` method is where the main part of the communication with the remote server is carried out.

In essence, the Generic Connection Framework is used to create a connection to the remote server. Once this connection has been opened we create and write the information that will form the request to the web server. This request information consists of the header information and the body of the message. The headers tell the server what sort of communication to expect; in

this case we shall be making a POST request. Headers are set using the `setRequestProperty()` method. The header name/value pairs follow the HTTP 1.1 format.

We then open an `OutputStream`, down which we send we will send our data. Once the data stream has been written, it is important to remember to `flush()` this data or else the server will not receive it.

```
public static String sendHttpPostRequest(String url, String value) throws IOException
{
    HttpURLConnection connection=null;
    InputStream is=null;
    OutputStream os=null;
    StringBuffer buffer=new StringBuffer();

    int response=0;
    try {
        // open the URL and set up the connection for POSTing.
        connection=(HttpURLConnection)Connector.open(url);
        connection.setRequestMethod(HttpURLConnection.POST);
        connection.setRequestProperty
            ("CONTENT-TYPE", "application/x-www-form-urlencoded");
        connection.setRequestProperty
            ("User-Agent", "Profile/MIDP 2.0 Configuration/CLDC-1.0");
        connection.setRequestProperty("Content-Language","en-US");
        // open the outputstream and send the POST data to the server.
        os=connection.getOutputStream();
        os.write(value.getBytes());
        os.flush();

        // now get the response
        response=connection.getResponseCode();
        System.out.println("Connection: "+response);
        if(response!=HttpURLConnection.HTTP_OK) {
            throw new IOException("Http response code:"+response);
        }

        is=connection.getInputStream();

        int chars;
        //read input
        while ((chars=is.read())!=-1) {
            buffer.append((char)chars);
        }
        System.out.println("Servlet Response: "+buffer.toString());
    }
    catch(Exception e) {
        buffer.setLength(0);
        buffer.append(POST_STATUS_FAILED);
    }
    finally {
        if(is!=null) {
            is.close();
        }
        if(os!=null) {
            os.close();
        }
        if(connection!=null) {
            connection.close();
        }
    }
    return buffer.toString();
}
```

Once the POST has been made, we read in the response code using the `getResponseCode()` method. If all goes well, an `HttpURLConnection.HTTP_OK` will be

returned. Once this has been confirmed we open an `InputStream` to read the response from the server. We have kept it simple here and the server responds with an “OK” string (as well as the `HTTP_OK`). This string is written to the servlet’s output stream by the servlet.

A character stream is read in from the `InputStream` and written to a buffer which is passed back to the calling method. After this, we need to ensure the resources no longer required are returned to the application manager, so we `close()` the input and output streams as well as the connection.

This method of sending data to a web server means the server receives the information as a block, rather than the more traditional parameter data. The servlet’s `doPost()` method needs to be aware of this. We have set the servlet up to receive information to its own `BufferedReader`, `HttpRequest.getReader()`, rather than make use of the `HttpRequest.getParameter()` method. We shall provide more detail on this in section 2.2.5.

The data sent is prepared by a method higher up the chain, `sendScore()`. It calls `sendHttpPostRequest()` by passing in the server name and the web page, in this case the servlet name.

```
public String sendScore(String score, String userName)
{
    // set the POST value
    StringBuffer value=new StringBuffer();
    value.append("score=");
    value.append(score);
    value.append("\nusername=");
    value.append(userName);

    try {
        // send it to the send POST method.
        return sendHttpPostRequest(HIGH_SCORE_URL+HIGH_SCORE_PAGE,
            value.toString());
    }
    catch(IOException io) {
        return POST STATUS FAILED;
    }
}
```

We have prepared the data in name/value pairs, separated by a carriage return. This means we are actually sending the data on different lines and the servlet’s buffered reader can read the data in line-by-line from the stream. This data is processed by the server and stored in memory, ready to be served as a high score table.

2.2.2. Sending a GET request to remote web server

Once the scores have been submitted to the server, the thread within `HighScoreCanvas.java` then moves on. It updates the display to show the user it has submitted the scores (Figure 2.2) and is now retrieving the scores from the server (Figure 2.3) using an HTTP GET request method.

The URL is passed to the `sendHttpGetRequest()` method. As previously, this is used to open an `HttpConnection`. The headers on this connection to the server are set in order to

prepare the server for the incoming request. As well as setting an `HttpConnection.GET` header, we have also set the `Content-Language` and `User-Agent` headers. The latter will show in the web server's log files, which may be useful for analysis later on in the MIDlet product lifecycle.

```
public static String sendHttpRequest(String url) throws IOException
{
    HttpConnection connection=null;
    InputStream is=null;
    StringBuffer buffer=new StringBuffer();

    int response=0;
    try {
        // open the URL and set up the connection for POSTing.
        connection=(HttpConnection)Connector.open(url);

        // set the request headers.
        connection.setRequestMethod(HttpConnection.GET);
        connection.setRequestProperty("User-Agent",
            "Profile/MIDP 2.0 Configuration/CLDC-1.0");
        connection.setRequestProperty("Content-Language", "en-US");

        // now get the response and test it
        response=connection.getResponseCode();
        if(response!=HttpConnection.HTTP_OK) {
            throw new IOException ("Http response code:"+response);
        }

        // open the stream
        is=connection.openInputStream();

        int chars;
        //read input and store in the buffer
        while ((chars=is.read())!=-1) {
            buffer.append((char)chars);
        }
    }
    catch(Exception e) {
        buffer.setLength(0);
        buffer.append(POST_STATUS_FAILED);
    }
    finally {
        try{
            if(is!=null){
                is.close();
            }
            if(connection!=null){
                connection.close();
            }
        }
        catch (Exception ignoreException) {}
    }

    return buffer.toString();
}
```

Once the MIDlet has established that the connection is `HTTP_OK`, it proceeds by opening the connection's `InputStream`. The application is now ready to read in the stream of data, in this case the high score data, sent by the server. This input stream is `read()` to a buffer and passed back to the calling method where it is turned into something more readable to the user.

We read these scores into a `Vector`, which stores the data as a `Score` object.

```
public static Score[] getHighScores()
{
    Score[] scores=new Score[10];

    try {
        String buffer=sendHttpRequest(HIGH SCORE URL);

        int index=0;
        int start=0;
        int element=0;
        int length=buffer.length();

        Vector result=stringTokenizer(buffer, "\n");

        for (element=0;element<result.size();element++) {
            scores[element]=parseScoreData
                ((String)result.elementAt(element));
        }
    }
    catch(Exception io)
    {
        System.out.println(io.toString());
    }

    return scores;
}
```

To do this, we have written a tokenizer method called `stringTokenizer`, which shreds the buffer data. The buffer data was sent out by the servlet with each username/score pair being printed on a new line. This means we can easily separate them in the MIDlet. MIDP 2.0 does not have its own regular expression methods and is the reason for making our own. The string buffer is passed to the tokenizer method and returns the data, now separated into a `Vector` of `String` objects.

```
private static Vector stringTokenizer(String tokenString, String token)
{
    Vector vtrResult=new Vector();

    int index=0;
    int start=0;
    int length=tokenString.length();
    while ((index=tokenString.indexOf(token,start))!=-1 || start<length) {
        vtrResult.addElement(tokenString.substring(start,index));
        start=index+3;
    }

    return vtrResult;
}
```

We then parse the score data vector and store it as a `Score` array.

```
private static Score parseScoreData(String valuePair)
{
    int index=valuePair.indexOf(",");
    Score score=new Score();
    score.setUserName(valuePair.substring(0,index));
    score.setScore(valuePair.substring(index+1, valuePair.length()));

    return score;
}
```

This array is iterated and the values drawn to the display by the `HighScoreCanvas.java` class.

2.2.3. The high score user interface

The user interface for the server communication is made up from two classes. One of them is a high-level `lcdui` class, called `NameEntryForm.java`, while the other, called `HighScoreCanvas.java`, manages the server communication.

NameEntryForm.java

This is a very simple class called by the application's layer manager class. It consists merely of one `StringItem` and a `TextField`. When the game's demonstration level finishes, the layer manager tells the `MIDlet` class, `DemoMIDlet` to set as current the `NameEntryForm`, which is a `Displayable` object. It receives the score from the layer manager and prompts the user to enter his name, as can be seen in Figure 2.1. Once the user enters a name in the box on this form, it is passed along with the score to the `HighScoreCanvas`.

HighScoreCanvas.java

This is a `Runnable` class that allows the `MIDlet` to communicate with the server, while informing the user what is currently happening.

The class is instantiated with the username and score. The thread is spawned and the process of communicating with the server is started. By creating this canvas as a `Runnable` object it means we can thread the processes that take time and resource, while keeping the user interface responsive and the user informed. As each step in the remote communication is made, the `Canvas.paint()` method displays the latest version of `strMessage`.

```
. . .
public HighScoreCanvas(String userName, String userScore){

    this.setFullScreenMode(true);
    this.userName=userName;
    this.userScore=userScore;
}

synchronized void start()
{
    thread=new Thread(this);
    thread.start();
}

public void run()
{
    try
    {
        // tell the user what is happening
        repaint();

        // create a new Score object and send the score
        Score score=new Score();
        score.sendScore(userScore, userName);

        // update the display to inform the user
        strMessage=GETTING HIGH SCORES;
        repaint();

        // get the high scores from the server
```

```

scores=score.getHighScores();

// once loaded render them on the display
strMessage=HIGH_SCORE_TABLE;
isHighScoreLoaded=true;
repaint();

}
catch(Exception ie)
{
    System.out.println(ie.toString());
}
}

synchronized void stop()
{
    . . .
}

public void drawHighScores(Graphics g, int x, int y)
{
    // draw the high scores to the graphics object

    . . .
}

public void paint(Graphics g)
{
    // paint to display

    . . .
}
}

```

Once the calls to the server have been made, the high scores are drawn to the display using the `drawHighScores()` and `paint()` methods, as seen in figure 2.4.

2.2.4. The remote server application

This paper would not be complete without a look at the server application passing data between the server and our `MIDlet`. It is really a rather simple servlet that services the `POST` and `GET` HTTP methods sent by the `MIDlet`, with its implementation of `doPost()` and `doGet()` methods. In a fully blown enterprise application, this servlet would serve as the interface to the larger server-side application, which would be made up of EJBs and other server-side components such as a database.

The `doGet()` method is responsible for responding to the `MIDlet`'s requests for data. The scores are stored, in memory, by the servlet as an `ArrayList` containing server-side `Score` objects. They are written to the `HttpServlet`'s output stream, each different username/score value pair being written on a new line. The `MIDlet` reads this stream and parses the data before displaying them on screen.

```

package com.symbian.score;

import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class HighScore extends HttpServlet

```

```

{
    private static String strHighScoreFile;
    private static ArrayList aScores;

    public void init(ServletConfig config) throws ServletException
    {
        if(aScores==null) {
            aScores=new ArrayList();
        }
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // get the scores in the text file
        response.setContentType("text/html");

        // draw the scores to the output
        getHighScores(response.getWriter());
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // set the content type
        response.setContentType("text/html");

        // create the output stream
        PrintWriter out = response.getWriter();

        // create the reader for the POST input
        BufferedReader br=request.getReader();

        // create a temporary store for the values
        Hashtable hParameters=new Hashtable();

        try {
            // read the POST input
            String score="";
            String userName="";

            String buffer;
            while((buffer=br.readLine())!=null) {
                getParameter(buffer, hParameters);
            }

            // store the values in memory for serving up later
            score=(String)hParameters.get("score");
            userName=(String)hParameters.get("username");

            addScore(score, userName);
            out.print("OK");
        }
        catch(Exception e) {
            out.print("failed");
        }
    }

    public void getParameter(String pair, Hashtable hParameters)
    {
        hParameters.put(pair.split("=")[0], pair.split("=")[1]);
    }

    public void addScore(String score, String userName)
    {
        // for demo purposes keep the high score table fresh
        doKeepFresh(aScores);

        // store the new submission in memory - an ArrayList
        aScores.add(new Score(userName, Long.parseLong(score)));
    }
}

```

```

}

public void doKeepFresh(ArrayList aScores)
{
    // as this is a demonstration pick 5 random submissions
    // to remove if we have 10 of more in store
    if(aScores.size()>=10) {
        for(int i=0;i<5;i++) {
            aScores.remove((int)Math.round((Math.random()*9));
        }
    }
}

public synchronized static void getHighScores(PrintWriter out)
{
    // sort the high scores in to order, value then alphabetical
    Collections.sort(aScores);

    // create an iterator
    Iterator iScores=aScores.iterator();
    Score score;

    // write the scores to the output.
    while (iScores.hasNext()){
        score=(Score)iScores.next();
        out.println(score.getUserName()+" "+score.getUserScore());
    }
}
}

```

The `doPost()` method is responsible for receiving the data from the MIDlet and storing it in the `ArrayList`. It might be better for an application storing larger amounts of data to use a database for this process; however, for this example we just store the values in memory.

The main thing to note from the `doPost()` method is that we are not using the more conventional method of taking the `HttpRequest` object and accessing its parameters using `getParameter()`. This is because the POST data is being sent as a block and has to be read by an input stream. For this we have used a `BufferedReader` created using the `HttpRequest.getReader()` method. We then read the input stream to a buffer and store the values in a `Hashtable`. The hash table data is then accessed and the name/value pairs are stored as `Score` objects in the `ArrayList`.

The `Score` object can be seen below and it gives us a `Comparable` object in which to store the high score data in the `ArrayList`. `ArrayList` being a collection provides us with an easy to use `sort()` method, allowing us to pass the data to the MIDlet in descending high score, then alphabetical order, removing the need for the mobile application to do any further processing with the data. We may as well use the processing power where it is more prevalent.

```

package com.symbian.score;

import java.util.*;

public class Score implements Comparable
{
    String userName;
    long userScore;

    Score(String userName, long userScore)

```

```

{
    this.userName=userName;
    this.userScore=userScore;
}

public String getUserName()           {return userName;}
public long getUserScore()            {return userScore;}

public void setUserName(String s)     {userName=s;}
public void setUserScore(long l)      {userScore=l;}

public int compareTo(Object object)
{
    final int BEFORE=-1;
    final int EQUAL=0;
    final int AFTER=1;

    Score score=(Score)object;
    if(getUserScore()==score.getUserScore()) {
        return getUserName().toLowerCase().compareTo
            (score.getUserName().toLowerCase());
    }
    else {
        if(getUserScore()>score.getUserScore()) {
            return BEFORE;
        }
        else if(getUserScore()<score.getUserScore()) {
            return AFTER;
        }
        return EQUAL;
    }
}
}

```

2.2.5. Network communication using SocketConnection

So far we have only demonstrated one way in which a Symbian OS phone might communicate over a network. As we have outlined in our introduction, Symbian OS phones may also use sockets as a means of network communication. Sockets make use of the Generic Connection Framework to access the network, using the `Connector.open()` factory method.

The connection is established in the following manner. The `open()` method receives the address of the remote server. The address is formatted as follows:

```
socket://<domainname.com>:<port number>
```

The `open()` is passed this string and the prefix of the string indicates to the connection which protocol type will be used when it is opened. As another example, for secure sockets we would use the following format:

```
ssl://<domainname.com>:<port number>
```

Once the connection has been established it is possible to open an output stream, `DataOutputStream`, down which server communication can be made. Conversely, to read the response from the server we create a `DataInputStream`.

One possible use for socket communication might be to create a MIDlet that connects to a POP3 mailbox and read our mail to the mobile phone.

```
public void getEmail()
{
    // specify the domain name and POP3
    // port number on our remote server.
    String connectionString = "socket://myemailserver.com:110";

    SocketConnection connection=null;

    DataOutputStream dos=null;
    DataInputStream dis=null;

    final String SERVER_ERROR="Server error";

    try {
        // establish a socket connection with remote server
        connection = (SocketConnection)Connector.open(connectionString);

        // create DataOutputSteam on top of the socket connection
        dos = connection.openDataOutputStream();

        // send the request to the server
        // remember to flush() the request.
        ...

        // create DataInputStream on top of the socket connection
        dis = connection.openDataInputStream();

        // retrieve the response from the server
        ...

    }
    catch (Exception e) {
        ...
    }
    finally
    {
        // free up I/O streams and close the socket connection
        try {
            if (dis != null) {
                dis.close();
            }
            if (dos != null) {
                dos.close();
            }
            if (connection != null) {
                connection.close();
            }
        }
        catch (Exception ignoreException) {}
    }
}
```

The server responds, sending back the requested data. We have by this point created a `DataInputStream` on the connection and read the inbound data stream in, character by character.

2.2.6. Networking and the Security Model

Networking APIs are regarded as security-sensitive under the MIDP 2.0 security model since opening a connection incurs billable events to the end-user. In order to ensure backward compatibility with MIDP 1.0, the MIDP 2.0 specification defines access rights for `HTTP` and `HTTPS` connections for both trusted and untrusted MIDlet suites. In the case of trusted MIDlet

suites, the MIDP 2.0 security model allows automatic access to HTTP (and HTTPS) connections. Trusted applications are those that are signed as trusted with a certificate and installed into a trusted protection domain. The Java Verified process is one example of how an ISV would create a Trusted Third Party application. According to the MIDP 2.0 specification, untrusted MIDlet suites can only open an `HTTPConnection` with explicit permission from the user.

The level of access for other networking APIs is not defined by the MIDP 2.0 specification, but depends on the security policy in force on the phone. The MIDP 2.0 specification includes an optional addendum, the “Recommended Security Policy for GSM/UMTS Compliant Devices” (RSP), which does define the access capabilities to these non-HTTP(S) networking APIs. The RSP specifies, among other things, a set of protection domains and the security policy relating to those domains.

Implementation of the RSP is not required for MIDP 2.0 compliance, but it is required for compliance with the “Java Technology for the Wireless Industry” specification (JTWI, JSR 185). Symbian OS, Version 8.0 includes a JTWI compliant Java implementation.

Under the RSP, MIDlet suites bound to the Trusted Third Party protection domain can open a `SocketConnection` with the explicit permission of the end-user, the default access level being set to “Session” (access granted until the MIDlet terminates). Untrusted MIDlets can open a `SocketConnection` with user permission, but in this case the default access level setting is “Oneshot” (permission required every time the API is invoked). Also note the JTWI imposes an additional restriction on which ports an untrusted MIDlet suite can access using a `SocketConnection`. Attempting to access any of the following ports 80, 8080 and 443 using a `SocketConnection` opened from an untrusted MIDlet suite will throw a `SecurityException`.

3. Serving mobile content from a web server

Once the `MIDlet` has been successfully completed and bug tested, the developer needs to ensure they can distribute the application to their customers. The most common method for doing this is over-the-air (OTA). It is essential that testing be carried out to make sure the `MIDlet` will install on the target phones, by delivery over-the-air. During development the developer will most likely use the infrared port or Bluetooth link for testing. However, this doesn’t guarantee the MIDlet will install at all over-the-air. The JAD and JAR files have to be exactly in sync with each other, and the JAD file has to be created according to the MIDP 2.0 specification. The specification is very precise and the JAD file has to reflect exactly the attributes of the JAR file.

Over-the-air distribution is convenient for the user and is a way to maximize revenue streams if the application has been distributed to content providers and network operators to good effect.

Another reason this should be tested thoroughly, is that this is the first test carried out by the Java Verified process. This process is called “pre-testing”, and it determines whether the `MIDlet` will install OTA. Failure to do so will result in a further iteration of testing and the

possibility of incurring greater costs. Therefore make sure this works. We shall be providing an overview of the Java Verified process in the next section of this paper.

Tables 3.1 and 3.2 provide two sample web pages that can be accessed by a Symbian OS phone. These have been set up to facilitate the downloading of our MIDlet.

3.1. Web server MIME type configuration

Before we connect the mobile phone to these pages, we need to check the configuration of the server to ensure that it can serve JAD and JAR files as downloadable objects. These settings are known as MIME types and are there for the web server to recognize the JAD and JAR file types as downloadable files. In Apache, the following lines should be added to one of two places: Either the master configuration file: `httpd.conf`, or alternatively the `.htaccess` file.

```
AddType text/vnd.sun.j2me.app-descriptor jad
AddType application/java-archive jar
```

We also need to tell Apache how to deal with WML files, to serve them as text. This is done by adding the following line:

```
AddType text/vnd.wap.wml wml
```

There is one remaining side issue that should be addressed as well. We noticed some unexpected results during the development of the socket connection to the remote server. We found the default virtual server, as specified by Apache, was not the one where our servlet was being served. This was despite specifying the virtual server's domain name in the socket connection. This resulted in the default web page, on the default virtual server at the same IP address, being served instead.

3.2. WML and XHTML files for serving the MIDlet

The creation of the web pages is a simple process. Tables 3.1 and 3.2 provide both a WML and XHTML sample page that can be used to test the OTA installation of a MIDlet. These files are placed under the `htdocs` directory of Apache. The user can now browse these pages, as if looking for an application to download. In practice, a network operator, content aggregator or distribution partner would host these pages.

This is a simple WML page with a link to the JAD file:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-// WAPFORUM// DTD WML 1.1// EN"
    "http://www.wapforum.org/DTD/wml 1.1.xml">
<wml>
  <card id="card1" title="Download Java Application">
    <p align="center">
      To download click below:
    </p>
    <p>
      <a href="DemoVaders.jad">DemoVaders</a>
    </p>
  </card>
```

```
</wml>
```

Table 3.1 - a sample WML page

The XHTML file works in the same way as the WML file. The Nokia 6600 and Sony Ericsson P900 will recognize both XHTML and WML file formats.

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-// W3C// DTD XHTML Basic 1.0// EN"
"http://www.w3.org/TR/xhtml1-basic/xhtml1-basic10.dtd">
<html xmlns="http:// www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
<head>
<title> Download Java Application </title>
</head>
<body>
<a href="DemoVaders.jad">DemoVaders</a><br/>
</body>
</html>
```

Table 3.2 - a sample XHTML page

Once the phone has recognized and validated the JAD file information against the contents of the JAR file, download and installation will commence.

4. Java Verified

While we are wrapping up this series of papers, it seems prudent to give developers an outline of the Java Verified process. The process has been created with the aim of creating trusted content for use by end-users, network operators, and mobile phone manufacturers. For developers it represents the chance to ship their application with the Java Powered™ and Java Verified Program Logos.

Content with the Java Powered™ Logo will carry more weight with distribution partners and end-users alike. This will in turn provide increased opportunities through global marketing strategies such as application catalogues run by the Java Verified member companies. Potential distribution partners will feel more willing to take trusted content.

4.1. Unified Testing Initiative (UTI)

The Java Verified process uses the Unified Testing Initiative (UTI), a testing framework created by Sun, Motorola, Nokia, Siemens and Sony Ericsson, the 'member companies'.

The UTI has been developed from the member companies existing testing programs to form a single process providing testing, promotion and distribution for independent software vendors (ISVs). The initiative, being an open industry-driven process, means input comes from a variety of companies within the mobile application market. Interested parties such as phone manufacturers, network operators, technology providers, publishers, developers and managed service providers, in fact any one interested in developing the mobile Java platform, may contribute to its future direction.

The goal as stated by the UTI is “to grow supply and demand for mobile Java applications by driving the development, promotion and distribution of high-quality mobile applications for the Java platform.”

As more companies contribute to the development of the test framework, the testing itself will become more comprehensive.

4.2. The Java Verified process

The following provides a brief outline of the process. More information is of course available on the Java Verified site (www.javaverified.com).

In the first instance the developer needs to read and execute the Java Verified Program Agreement, found here:

<http://leasequote.sun.com:8080/nextance/JavaVerified.html>

This has to be completed for each application that will be submitted for testing. A general publicity release form also has to be completed. This enables the developer to take advantage of any marketing opportunities offered by the member companies. A list of the currently supported devices can be downloaded from the site -

<http://www.javaverified.com/docs/DeviceMatrix.pdf>. Assuming the devices within the matrix are applicable to the developer’s application, complete and submit the registration form - <http://www.javaverified.com/developer/main>. The test criteria should also be examined to make sure the developer is aware of what standard the applications will need to reach to pass the tests. The criteria can be found at this address - http://www.javaverified.com/docs/Unified_Test_Criteria.pdf.

4.2.1. Pre-testing

In the first instance, the application is tested to make sure the JAD and JAR files are aligned. This process will, among other things, determine what APIs have been used, whether it conforms to the JTWI specification (JSR 185) and the scope of devices it is able to run on. When the application has passed the pre-testing process it is given a ticket number, which will be used to identify the application for the rest of the testing process. This process will be repeated until the pre-testing criteria have been met.

4.2.2. Test provider

Once the pre-testing process has been completed, the developer needs to find a suitable test provider. The developer will need to form a business relationship with a test house, so consideration to location should be taken into account. Once a testing agreement has been reached, including testing and re-testing costs (set independently), the test provider will request the application’s ticket number. The test provider will use the previously allocated ticket number to obtain the application from the program, rather than receive it directly from the application developer. This step has been implemented in such a way so as to re-enforce the application’s integrity throughout the process.

4.2.3. Application testing

The criteria for testing can be downloaded from the Java Verified website, http://www.javaverified.com/docs/Unified_Test_Criteria.pdf. It sets out all the testing conditions that need to be met by the application. It is an iterative process, and the application will be tested to cover every version of firmware for the devices the MIDlet is targeted at.

The document provides in depth detail of the process, but in summary the following areas are covered by the testing process:

4.2.3.1 MIDP

The MIDP test covers areas such as application launch, a test primarily concerned with the MIDlet lifecycle. Will it launch? Will it stop when required? The user interface and user interaction are then assessed. The functionality of the MIDlet is analyzed. Does it do what it sets out to do? The process compares the help files and user manuals to the actual operation of the application. The operation of the application with regard to the hardware and the other software on the phone is assessed. In other words will the MIDlet be disruptive to the normal running of the phone?

MIDlets that are network-enabled are tested for secure transmissions and the handling of network failures. This includes amicably dealing with server-side as well as network failures. Also localization is examined. Have languages and date/time formats been taken into account by the developer?

4.2.3.2 JTWI

The JTWI section of the testing is more specific. It covers those application features under JSR 185. The new security model is one of these areas. Many of the tests revolve around the correct behavior of the four security domains: manufacturer, operator, trusted third party and untrusted. Those MIDlet suites whose signature cannot be authenticated as manufacturer, operator or trusted third party will be considered untrusted.

The MIDP 2.0 features such as the push registry and the wireless messaging API are also tested. Does the MIDlet respond to registered alarms or incoming requests to activate? The process also examines the wireless messaging API functionality of the application.

4.2.3.3 MMAPI & Bluetooth technology

This final section looks at the operation of the Bluetooth link and the Mobile Media API. How does the application behave when the Bluetooth link is switched on and off, for example?

4.2.3.4 Digital signing

Once the application has passed the testing process, it is sent to GeoTrust (<http://www.geotrust.com>) for digital signing. This provides testing integrity and proves the application has passed the Java Verified process. The application of the digital signature means the MIDlet may be installed, authenticated and operated within the trusted third-party domain.

The application is returned to the developer and it may not be changed after this process has been completed. If it is, then the digital signature will no longer be valid.

Now that the application has been certified, use can be made of the Java Powered Logo™. It can be placed on the splash screen or an “about” screen for example and it can be used once the Java Verified program agreement has been signed. The logo must have been integrated within the application prior to pre-testing. Adding it after the testing process will invalidate the certification.

4.3. Marketing opportunities

Once the Java Verified process has been completed the developer is eligible to apply to member companies for inclusion in marketing catalogues, such as the Nokia sales channels, as well those offered by Siemens, Motorola and Sun.

4.4. The testing houses

One important part of the Java Verified process is of course the testing houses. As we mentioned before developers will have to form business relationships with testing houses. Currently, there are four testing partners associated with Java Verified: BabelMedia, CapGemini, NSTL and RelQ.

Unsurprisingly, these companies specialize in the testing of applications and have many automated processes that speed up the process. We spoke to BabelMedia to see what the process meant to them and to the developers they have worked with. We spoke to Paul Munford, Communications director, to find out how he viewed the program.

One of the interesting aspects of the program for developers is improving the route-to-market. For small developers this can be especially troublesome as it takes time and money building relationships with many partners. “For smaller developers, who may not wish or are unable to maintain relationships with multiple publishers or portals, it’s ideal as Java Verified act as a content aggregator or portal,” explained Paul Munford. This is a reasonable view to take. Investing in the program allows the developer to concentrate on the business of creating Java content.

“You don’t have to sell the content once it’s approved, it is available for any operator to download and sell on your behalf, so you reach markets you may have been totally unaware of”, he added. “So it does ease the route to market – rather than try and sell to people, people are browsing your wares instead.”

One of the main concerns for the consumer, and of course the distribution partners, is to be able to get hold of quality products. We wondered what affect this process had on the technical aspects of Java products. “On a technical level, all Java Verified applications meet standards set down by Sun to make the consumer experience a successful one, so yes, product quality is improved and there is even a degree of consistency across applications from different publishers.”

At the end of the day, this will boast consumer confidence in Java products, which is good for all mobile Java developers.

5. Conclusion

The ability to add a community aspect to stand alone applications adds another dimension to Java content. The network operators benefit from the added airtime use and the longevity of the application is increased as users seek to beat their friends and family to the top of the high score tables.

But gaming is not the all-consuming use of MIDP. Enterprises would also benefit from being able to send data to remote staff or enable them to submit sales figures, update stock tables and also keep up-to-date with appointments. The ability of Symbian OS phones to communicate using a diverse range of networking protocols gives the application developer a broader outlook to mobile development.

As the mobile Java market reaches a certain level of maturity, it is only right that it is able to provide a standard of trust which consumers can rely on when making choices over which product to purchase. Without quality products, the confidence in the overall offering will diminish and that will affect the market as a whole. Remember the early days of WAP? Java will not fail to deliver in the way early WAP phones did, as it offers the user a richer form of content. For the platform to be taken seriously by the enterprise it needs to be seen to offer mature, well-tested and well-positioned content. The Java Verified process offers the framework with which this can be achieved so it is important for developers to be aware of how it works what it can offer them.

[Back to Developer Library](#)

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.