**Telnet using Java**

This article will demonstrate how using the SSH Factory Telnet component you can establish an interactive Telnet session with a Telnet server. This article will also serve as a prerequisite to an upcoming article titled [Scripting Telnet sessions using Java](http://www.jscape.com/articles/scripting_telnet_sessions_using_java.html) which will demonstrate how to automate Telnet commands in a batch like process. To see what else iNet Factory has to offer [Download a FREE 30 day SSH Factory Evaluation](http://www.jscape.com/sshfactory/dload.html).

**Overview of Telnet**

The Telnet client component provides a method for communicating with a Telnet server. The process for establishing an interactive session with a Telnet server using the Telnet component is as follows:

1. Creating a new Telnet instance
2. Implementing a TelnetListener
3. Registering a TelnetListener
4. Establishing a connection
5. Performing option negotiation
6. Receiving data
7. Sending data
8. Releasing a connection

Each of these processes is described in the sections below.

**Creating a new Telnet instance**

To create a new Telnet instance ensure that the com.jscape.inet.telnet package is included in your import statements and create a new Telnet instance providing the Telnet server hostname as an argument.

Telnet telnet = **new**Telnet(hostname);

**Implementing a TelnetListener**

The Telnet component is one of the few components in SSH Factory which MUST have a registered event listener. The reason for this will be clear later. The following is a sample implementation of the TelnetListener interface. A TelnetListener instance must be created and registered with the Telnet instance prior to invoking the Telnet#connect method to ensure that all data sent by the Telnet server is captured.

**public class**MyTelnetListener **implements**TelnetListener {  
  
  **public void**connected(TelnetConnectedEvent event) {  
  }  
  
  **public void**disconnected(TelnetDisconnectedEvent event) {  
  }  
  
  **public void**doOption(DoOptionEvent event) {  
  }  
  
  **public void**dontOption(DontOptionEvent event) {  
  }  
  
  **public void**willOption(WillOptionEvent event) {  
  }  
  
  **public void**wontOption(WontOptionEvent event) {  
  }  
  
  **public void**doSubOption(DoSubOptionEvent event) {  
  }  
  
  **public void**dataReceived(TelnetDataReceivedEvent event) {  
  }  
}

**Registering a TelnetListener**

To register a listener invoke the Telnet#addTelnetListener method providing a TelnetListener instance as its argument.

Telnet telnet = **new**Telnet(hostname);  
MyTelnetListener listener = **new**MyTelnetListener();  
telnet.addTelnetListener(listener);

**Establishing a connection**

Once a Telnet instance has been created and a TelnetListener registered you may establish a connection to the Telnet server by invoking the Telnet#connect method.

Telnet telnet = **new**Telnet(hostname);  
MyTelnetListener listener = **new**MyTelnetListener();  
telnet.addTelnetListener(listener);  
telnet.connect();

**Performing option negotiation**

Upon establishing a connection the process of option negotiation will begin. Option negotiation is a communications process for the Telnet client and the Telnet server to come up with a set of agreed upon protocols. An example of option negotiation is agreeing upon the terminal emulation (e.g. vt100, xterm, dumb) to use in the Telnet session.

Option negotiation as its name implies is optional and may be initiated by either the client or server. This does not mean however that option negotiation may be ignored. For example, in the event that the server attempts to perform option negotiation the client must respond by either accepting or rejecting the option request. Capturing option negotiation data from the Telnet server is accomplished using the TelnetListener class. In order to capture this data you must register an instance of the TelnetListener class with the Telnet instance as shown earlier.

In performing option negotiation there are four (5) Telnet protocol commands that can be used by the client and server.

1. DO OPTION - Requests to enable an option.
2. DONT OPTION - Refuses offer to enable an option.
3. WILL OPTION - Offers to enable an option.
4. WONT OPTION - Refuses request to enable an option.
5. SUB OPTION - For option subnegotiation

For the purposes of this article we will refuse all options both requested and offered by the Telnet server. This in effect will give us a basic Telnet client that is capable of exchanging data with the Telnet server. In order to capture and refuse options requested or offered by the Telnet server you will need to overload the TelnetListener#doOption and TelnetListener#willOption methods as follows:

**public void**doOption(DoOptionEvent event) {  
  // refuse any options requested by Telnet server  
  telnet.sendWontOption(event.getOption());      
}  
  
  
**public void**willOption(WillOptionEvent event) {  
  // refuse any options offered by Telnet server  
  telnet.sendDontOption(event.getOption());  
}

**Receiving data**

Once option negotiation has been completed you may receive data sent by the Telnet server. To receive data, overload the TelnetListener#dataReceived method as follows:

**public void**dataReceived(TelnetDataReceivedEvent event) {  
  // print data received from Telnet server to console  
  System.out.println(event.getData());  
}

**Sending data**

To send data to the Telnet server you will first obtain an OutputStream from the Telnet instance. Given this OutputStream you can then send data to the Telnet server as follows:

// get output stream  
output = telnet.getOutputStream();

// sends all data entered at console to Telnet server  
**while**((input = reader.readLine()) != **null**) {  
  **if**(connected) {  
    ((TelnetOutputStream) output).println(input);  
  } **else**{  
    **break**;  
  }  
}

In the example above the input comes from the console and is redirected to the Telnet server using the TelnetOutputStream#println method. The TelnetOutputStream class is used as it automatically appends a \r\n (carriage return line feed) to the end of the data sent. This is required by the Telnet server for it to know when it may begin processing the data received.

**Releasing a connection**

To release an established connection simply invoke the Telnet#disconnect method as follows:

telnet.disconnect();

**Example**

Below is the fully functional source code for an interactive Telnet client written using the SSH Factory Telnet component. This same example is available in the SSH Factory evaluation download in the examples directory.

001 **import**com.jscape.inet.telnet.\*;  
002 **import**java.io.\*;  
003   
004 **public class**TelnetExample **extends**TelnetAdapter {  
005   
006   **private**Telnet telnet = **null**;  
007   **private**OutputStream output = **null**;  
008   **private static**BufferedReader reader = **null**;  
009   **private boolean**connected = **false**;  
010   
011   **public**TelnetExample(String hostname) **throws**IOException {  
012   
013     String input = **null**;  
014     // create new Telnet instance  
015     telnet = **new**Telnet(hostname);  
016   
017     // register this class as TelnetListener  
018     telnet.addTelnetListener(**this**);  
019   
020     // establish Telnet connection  
021     telnet.connect();  
022     connected = **true**;  
023   
024     // get output stream  
025     output = telnet.getOutputStream();  
026   
027     // sends all data entered at console to Telnet server  
028     **while**((input = reader.readLine()) != **null**) {  
029       **if**(connected) {  
030         ((TelnetOutputStream) output).println(input);  
031       } **else**{  
032         **break**;  
033       }  
034     }  
035   }  
036   
037   /\*\* Invoked when Telnet socked is connected.  
038    \* @see TelnetConnectedEvent  
039    \* @see Telnet#connect  
040    \*/  
041   **public void**connected(TelnetConnectedEvent event) {  
042     System.out.println("Connected");  
043   }  
044   
045   /\*\*   
046    \* Invoked when Telnet socket is disconnected. Disconnect can  
047    \* occur in many circumstances including IOException during socket read/write.  
048    \* @see TelnetDisconnectedEvent  
049    \* @see Telnet#disconnect  
050    \*/  
051   **public void**disconnected(TelnetDisconnectedEvent event) {  
052     connected = **false**;  
053     System.out.print("Disconnected.  Press enter key to quit.");  
054   }  
055   
056   /\*\*  
057    \* Invoked when Telnet server requests that the Telnet client begin performing specified <code>TelnetOption</code>.  
058    \* @param event a <code>DoOptionEvent</code>  
059    \* @see DoOptionEvent  
060    \* @see TelnetOption  
061    \*/  
062   **public void**doOption(DoOptionEvent event) {  
063     // refuse any options requested by Telnet server  
064     telnet.sendWontOption(event.getOption());  
065   }  
066   
067   /\*\*  
068    \* Invoked when Telnet server offers to begin performing specified <code>TelnetOption</code>.  
069    \* @param event a <code>WillOptionEvent</code>  
070    \* @see WillOptionEvent  
071    \* @see TelnetOption  
072    \*/  
073   **public void**willOption(WillOptionEvent event) {  
074     // refuse any options offered by Telnet server  
075     telnet.sendDontOption(event.getOption());  
076   }  
077   
078   /\*\*  
079    \* Invoked when data is received from Telnet server.  
080    \* @param event a <code>TelnetDataReceivedEvent</code>  
081    \* @see TelnetDataReceivedEvent  
082    \*/  
083   **public void**dataReceived(TelnetDataReceivedEvent event) {  
084     // print data received from Telnet server to console  
085     System.out.print(event.getData());  
086   }  
087   
088   /\*\*  
089    \* Main method for launching program  
090    \* @param args program arguments  
091    \*/  
092   **public static void**main(String[] args) {  
093     **try**{  
094       reader = **new**BufferedReader(**new**InputStreamReader(System.in));  
095   
096       // prompt user for Telnet server hostname  
097       System.out.print("Enter Telnet server hostname (e.g. 10.0.0.1): ");  
098       String hostname = reader.readLine();  
099   
100       // create new TelnetExample instance  
101       TelnetExample example = **new**TelnetExample(hostname);  
102     } **catch**(Exception e) {  
103       e.printStackTrace(System.out);  
104     }  
105   }  
106   
107 }

1. Line 1. Add necessary import statements.
2. Line 4. Implement the TelnetAdapter class to capture events fired by the Telnet class. TelnetAdapter is an abstract implementation of the TelnetListener class.
3. Line 15. Create a new Telnet instance using provided hostname.
4. Line 18. Register Telnet event listener.
5. Line 21. Establish connection with Telnet server.
6. Line 25. Get OutputStream for writing data to Telnet server.
7. Line 28. Read data from console and send to Telnet server.
8. Line 41. Capture when connection is established.
9. Line 51. Capture when connection is released.
10. Line 62. Refuse options requested by Telnet server.
11. Line 73. Refuse options offered by Telnet server.
12. Line 83. Print data received from Telnet server to console.
13. Line 92. main() method for starting Telnet client