Shell Tutorial

@version@ (@date@)

by Dieter Wimberger

Table of contents

1. About

This document describes how to write a Shell implementation and points out important issues with the implementation.

2. Basics

To be able to understand this tutorial, you should first try to get comfortable with the following elements of the API:

net.wimpi.telnetd.shell.Shell
The interface that you will need to implement.
net.wimpi.telnetd.event.ConnectionListener
The *Shell* interface extends this interface to enforce the handling of connection events. A separate section of this tutorial will describe event handling in more detail.
net.wimpi.telnetd.io.BasicTerminalIO
The base class for Terminal I/O. A separate section of this tutorial will describe more about terminal I/O issues.
net.wimpi.telnetd.net.ConnectionData
A class which gives you access to connection specific references and information. If your application becomes more sophisticated, you might probably want to make use of this instance.

Throughout the terminal you will see that there are probably more classes/interfaces and material that you should become familiar with.

Also make sure that you check out the rest of the deployment and configuration documentation, to make sure you know how to configure and startup with your shell.

3. Connection Events

As this is a vital point of the shell implementation, we will discuss it first. By implementing the shell interface you are automatically enforced to implement the ConnectionListener interface. This is not very difficult, but requires some background to understand the behavior at runtime.
There are following connection events:

CONNECTION_LOGOUTREQUEST
Occurs when a connection requested disgraceful logout by sending a <Ctrl>-<D> key combination.
CONNECTION_BREAK

Occurs when the connection sent a NVT BREAK signal.
CONNECTION_IDLE
Occurs if a connection has been idle exceeding the configured time to warning.
CONNECTION_TIMEDOUT
Occurs if a connection has been idle exceeding the configured time to warning and the configured time to timedout.

Each event has it's handling method, as defined by the interface, which will be called by the ConnectionManager of the respective listener. This implies, that the handling routine you write should return control as fast as possible.

Warning:
You should carefully consider what strategy you use for event handling as the connection thread will be blocked when reading from the I/O.

A possible strategy would be to flag or queue the event, interrupt the blocked connection thread in a controlled fashion and make it handle events before reading from the I/O again. Another possible strategy is a thread pool for handling events.
Logically this depends on your application, as well as the event type.

4. Implementing the Shell Interface

You have to start with defining a class that implements the interface:

```
public class SimpleShell
    implements Shell {
```

In many cases you will want to have some reference to the I/O and the connection.

```
private Connection m_Connection;
private BasicTerminalIO m_IO;
```

An important part of the implementation is a factory method that will allow the shell manager to create instances of your shell.

Warning:
This method should be a static method of the class and is part of the Shell implementation contract not defined in the interface. If it is not encountered when loading the shell class, an exception will be thrown.

```
public static Shell createShell() {
  return new SimpleShell();
}//createShell
```

Note:
If you might want to recycle shell instances, you can do this independent of the ShellManager through this factory method.

The key method of the shell is the *run(Connection con)* method which will be called by the connection to pass control to your application (i.e. shell implementation), once the connection has been established.

```
public void run(Connection con) {
  m_Connection = con;
  m_IO = m_Connection.getTerminalIO();
  //register the connection listener
  m_Connection.addConnectionListener(this);

  //your shell routines

}
```

We will come back to this method later with some example.

Now what is missing is the ConnectionListener implementation mentioned beforehand. The following code snippet provides a skeleton dummy implementation:

```
public void connectionTimedOut(ConnectionEvent ce) {
  m_IO.write("CONNECTION_TIMEDOUT");
  m_IO.flush();
  //close connection
  m_Connection.close();
}//connectionTimedOut

public void connectionIdle(ConnectionEvent ce) {
  m_IO.write("CONNECTION_IDLE");
  m_IO.flush();
}//connectionIdle

public void connectionLogoutRequest(ConnectionEvent ce) {
  m_IO.write("CONNECTION_LOGOUTREQUEST");
  m_IO.flush();
}//connectionLogout

public void connectionSentBreak(ConnectionEvent ce) {
  m_IO.write("CONNECTION_BREAK");
  m_IO.flush();
}//connectionSentBreak
```

So far so good. Now, to have a skeleton that is a simple running example, we will add some shell output to the run method:

```
m_IO.eraseScreen();  //erase the screen
m_IO.homeCursor();  //place the cursor in home position
m_IO.write("SimpleShell. Thanks for connecting.\r\n");  //some output
m_IO.flush();  //flush the output to ensure it is sent
```

Now this is not very exciting, but we have our first example (and Shell skeleton):

```
package your.package;

public class SimpleShell
```

```
    implements Shell {

  private Connection m_Connection;
  private BasicTerminalIO m_IO;

  public void run(Connection con) {
    m_Connection = con;
    m_IO = m_Connection.getTerminalIO();
    //register the connection listener
    m_Connection.addConnectionListener(this);

    m_IO.eraseScreen();  //erase the screen
    m_IO.homeCursor();  //place the cursor in home position
    m_IO.write("Dummy Shell. Thanks for connecting.\r\n");  //some output
    m_IO.flush();  //flush the output to ensure it is sent
  }//run

  public void connectionTimedOut(ConnectionEvent ce) {
    m_IO.write("CONNECTION_TIMEDOUT");
    m_IO.flush();
    //close connection
    m_Connection.close();
  }//connectionTimedOut

  public void connectionIdle(ConnectionEvent ce) {
    m_IO.write("CONNECTION_IDLE");
    m_IO.flush();
  }//connectionIdle

  public void connectionLogoutRequest(ConnectionEvent ce) {
    m_IO.write("CONNECTION_LOGOUTREQUEST");
    m_IO.flush();
  }//connectionLogout

  public void connectionSentBreak(ConnectionEvent ce) {
    m_IO.write("CONNECTION_BREAK");
    m_IO.flush();
  }//connectionSentBreak

   public static Shell createShell() {
    return new SimpleShell();
  }//createShell

}//class SimpleShell
```

5. Configuring and Running

Now that we have a simple implementation, let's see it in action. The following is a combined properties file that defines a sample listener which will run the SimpleShell we have just created.

```
#Unified telnet proxy properties
```

```
#Daemon configuration example.
#Created: ??/??/2005 you


############################
# Telnet daemon properties #
############################

####################
# Terminals Section #
####################

# List of terminals available and defined below
terminals=vt100,ansi,windoof,xterm

# vt100 implementation and aliases
term.vt100.class=net.wimpi.telnetd.io.terminal.vt100
term.vt100.aliases=default,vt100-am,vt102,dec-vt100

# ansi implementation and aliases
term.ansi.class=net.wimpi.telnetd.io.terminal.ansi
term.ansi.aliases=color-xterm,xterm-color,vt320,vt220,linux,screen

# windoof implementation and aliases
term.windoof.class=net.wimpi.telnetd.io.terminal.Windoof
term.windoof.aliases=

# xterm implementation and aliases
term.xterm.class=net.wimpi.telnetd.io.terminal.xterm
term.xterm.aliases=

##################
# Shells Section #
##################

# List of shells available and defined below
shells=simple

# shell implementations
shell.simple.class=your.package.SimpleShell

####################
# Listeners Section #
####################
listeners=std


# std listener specific properties

#Basic listener and connection management settings
std.port=6666
std.floodprotection=5
std.maxcon=25
```

```
# Timeout Settings for connections (ms)
std.time_to_warning=3600000
std.time_to_timedout=60000

# Housekeeping thread active every 1 secs
std.housekeepinginterval=1000

std.inputmode=character

# Login shell
std.loginshell=simple

# Connection filter class
std.connectionfilter=none
```

Now we assume that we saved the above in a properties file named *test.properties*. You can now startup the telnetd as follows (again we assume you have a JRE/JDK installed and the java VM binary in the PATH):

```
java -classpath telnetd.jar:commons-logging.jar:log4j.jar
net.wimpi.telnetd.TelnetD -D -Dlog4j.configuration=<your URL for
log4j.properties> <your URL for test.properties>
```

Using telnet to login, you should see something like the following:

```
[Fangorn:~] wimpi$ telnet localhost 6666
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

Then the screen will be erased, and you should end up with the following:

```
Simple Shell. Thanks for connecting.
Connection closed by foreign host.
[Fangorn:~] wimpi$
```

6. Terminal I/O

As mentioned in the overview (../deployment/) there are elements in the library that can help you with the I/O. The most basic I/O is **net.wimpi.telnetd.io.BasicTerminalIO**. You can directly use it to manipulate the terminal screen, you can wrap it into basic InputStream, Reader, OutputStream or Writer implementations as well as design your own I/O classes on top that help you most with your application.

Another option provided by the library is the toolkit implementation that has been started in the *net.wimpi.telnetd.io.toolkit* package. Work on it is still in progress, and contributions would be more than welcome. Some documentation/how-to will probably follow somewhen. What you can do to check out it's functionality (respectively what it does), is to run the *net.wimpi.telnetd.shell.DummyShell* implementation (in character mode!) and press *t* at the prompt. This will start you into a small demo of the implemented elements.

Note:

There is a target called *runit* that should startup a daemon with a listener that has the DummyShell set as it's login shell. In most cases you can advance between active elements using *ENTER*, except for the full screen editor, where you will have to use *TAB*.

Probably it is possible to adapt some of the code of projects you can find online (like jcurzez, Java JNI Courses, etc.).

6.1. Styled Output

The implementation has support that helps you with creating styled output (bold, colors etc.). If the terminal negotiated with a specific connection supports it, style escape codes specific to the terminal will be sent.

The mechanism is rather simple, adding markups to strings that will be translated into escape sequences the moment the string is written to the connection. The utility class **net.wimpi.telnetd.io.terminal.ColorHelper** contains definitions, as well as helper methods to add them properly to strings you pass in (see API docs).

7. ConnectionData and Shell Switching

It is possible to obtain some basic information about a connection from the shell implementation. This is done by obtaining an *net.wimpi.telnetd.net.ConnectionData* instance from the acutal connection. The following code snippet is an example:

```
ConnectionData cd = m_Connection.getConnectionData();
m_IO.write("Connected from: " +
    cd.getHostName() + "[" + cd.getHostAddress() + ":"
    + cd.getPort() + "]" + BasicTerminalIO.CRLF);
m_IO.write("Guessed Locale: " +
    cd.getLocale() + BasicTerminalIO.CRLF);
m_IO.write(BasicTerminalIO.CRLF);

m_IO.write("Negotiated Terminal Type: " +
    cd.getNegotiatedTerminalType() + BasicTerminalIO.CRLF);
m_IO.write("Negotiated Columns: " + cd.getTerminalColumns() +
    BasicTerminalIO.CRLF);
m_IO.write("Negotiated Rows: " + cd.getTerminalRows() +
        BasicTerminalIO.CRLF);
```

A shell might switch or allow to switch to another shell (the same environment will be available to any shell, so you can pass parameters or references between shells without problem).

Note:

Don't forget about the connection event listening. In most cases you might want to unregister the actual shell and register the one you are switching to.

The following code snippet represents a simple example:

```
if(m_Connection.setNextShell("simple2")) {
  m_Connection.removeConnectionListener(this);
  m_IO.write("Switching to Simple2Shell" + BasicTerminalIO.CRLF);
} else {
  m_IO.write("Could not set shell to switch to.");
}
```

8. The Full Example

The SimpleShell class with the use of the ConnectionData instance, as well as the environment. Will switch to Simple2Shell (follows below):

```
package your.package;

import net.wimpi.telnetd.io.BasicTerminalIO;
import net.wimpi.telnetd.net.Connection;
import net.wimpi.telnetd.net.ConnectionData;
import net.wimpi.telnetd.event.ConnectionEvent;

public class SimpleShell
    implements Shell {

  private Connection m_Connection;
  private BasicTerminalIO m_IO;

  public void run(Connection con) {
    m_Connection = con;
    m_IO = m_Connection.getTerminalIO();
    //register the connection listener
    m_Connection.addConnectionListener(this);

    m_IO.eraseScreen();  //erase the screen
    m_IO.homeCursor();  //place the cursor in home position

    //output connection data
    ConnectionData cd = m_Connection.getConnectionData();
    m_IO.write("Connected from: " + cd.getHostName() +
        "[" + cd.getHostAddress() + ":" + cd.getPort() + "]"
        + BasicTerminalIO.CRLF
    );
    m_IO.write("Guessed Locale: " + cd.getLocale() +
        BasicTerminalIO.CRLF);
    m_IO.write(BasicTerminalIO.CRLF);
    //output negotiated terminal properties
    m_IO.write("Negotiated Terminal Type: " +
        cd.getNegotiatedTerminalType() + BasicTerminalIO.CRLF);
```

```
    m_IO.write("Negotiated Columns: " + cd.getTerminalColumns() +
        BasicTerminalIO.CRLF);
    m_IO.write("Negotiated Rows: " + cd.getTerminalRows() +
        BasicTerminalIO.CRLF);
    //add environment variable to pass between shells
    cd.getEnvironment().put("key1","value1");
    cd.getEnvironment().put("key2", "value2");
    cd.getEnvironment().put("key3", "value3");
    cd.getEnvironment().put("key4", "value4");

    if(m_Connection.setNextShell("simple2")) {
      m_Connection.removeConnectionListener(this);
      m_IO.write("Switching to Simple2Shell" + BasicTerminalIO.CRLF);
    } else {
      m_IO.write("Could not set shell to switch to.");
    }
    m_IO.flush();  //flush the output to ensure it is sent
  }//run

  public void connectionTimedOut(ConnectionEvent ce) {
    m_IO.write("CONNECTION_TIMEDOUT");
    m_IO.flush();
    //close connection
    m_Connection.close();
  }//connectionTimedOut

  public void connectionIdle(ConnectionEvent ce) {
    m_IO.write("CONNECTION_IDLE");
    m_IO.flush();
  }//connectionIdle

  public void connectionLogoutRequest(ConnectionEvent ce) {
    m_IO.write("CONNECTION_LOGOUTREQUEST");
    m_IO.flush();
  }//connectionLogout

  public void connectionSentBreak(ConnectionEvent ce) {
    m_IO.write("CONNECTION_BREAK");
    m_IO.flush();
  }//connectionSentBreak

  public static Shell createShell() {
    return new SimpleShell();
  }//createShell

}//class SimpleShell
```

The following code is for the *Simple2Shell*, to show that switching really works, and that the environment has not changed during the swtich.

```
package your.package;

import java.util.Hashtable;
import java.util.Enumeration;
```

```
import net.wimpi.telnetd.io.BasicTerminalIO;
import net.wimpi.telnetd.net.Connection;
import net.wimpi.telnetd.net.ConnectionData;
import net.wimpi.telnetd.event.ConnectionEvent;

public class Simple2Shell
    implements Shell {

  private Connection m_Connection;
  private BasicTerminalIO m_IO;

  public void run(Connection con) {
    m_Connection = con;
    m_IO = m_Connection.getTerminalIO();
    //register the connection listener
    m_Connection.addConnectionListener(this);

    m_IO.write("Simple2Shell" + BasicTerminalIO.CRLF);
    //output stored environment variables
    ConnectionData cd = m_Connection.getConnectionData();
    Hashtable env = cd.getEnvironment();
    for(Enumeration enum = env.keys(); enum.hasMoreElements();) {
      String key = (String) enum.nextElement();
      m_IO.write(key + "=" + env.get(key) + BasicTerminalIO.CRLF);
    }
    m_IO.write("Goodbye!" + BasicTerminalIO.CRLF);
  }//run

  public void connectionTimedOut(ConnectionEvent ce) {
    m_IO.write("CONNECTION_TIMEDOUT");
    m_IO.flush();
    //close connection
    m_Connection.close();
  }//connectionTimedOut

  public void connectionIdle(ConnectionEvent ce) {
    m_IO.write("CONNECTION_IDLE");
    m_IO.flush();
  }//connectionIdle

  public void connectionLogoutRequest(ConnectionEvent ce) {
    m_IO.write("CONNECTION_LOGOUTREQUEST");
    m_IO.flush();
  }//connectionLogout

  public void connectionSentBreak(ConnectionEvent ce) {
    m_IO.write("CONNECTION_BREAK");
    m_IO.flush();
  }//connectionSentBreak

  public static Shell createShell() {
    return new Simple2Shell();
  }//createShell
```

```
}//class Simple2Shell
```

To run this example you have to modify the above configuration properties to include the second shell we wrote:

```
#Unified telnet proxy properties
#Daemon configuration example.
#Created: ??/??/2005 you


############################
# Telnet daemon properties #
############################

####################
# Terminals Section #
####################

# List of terminals available and defined below
terminals=vt100,ansi,windoof,xterm

# vt100 implementation and aliases
term.vt100.class=net.wimpi.telnetd.io.terminal.vt100
term.vt100.aliases=default,vt100-am,vt102,dec-vt100

# ansi implementation and aliases
term.ansi.class=net.wimpi.telnetd.io.terminal.ansi
term.ansi.aliases=color-xterm,xterm-color,vt320,vt220,linux,screen

# windoof implementation and aliases
term.windoof.class=net.wimpi.telnetd.io.terminal.Windoof
term.windoof.aliases=

# xterm implementation and aliases
term.xterm.class=net.wimpi.telnetd.io.terminal.xterm
term.xterm.aliases=

##################
# Shells Section #
##################

# List of shells available and defined below
shells=simple,simple2

# shell implementations
shell.simple.class=your.package.SimpleShell
shell.simple2.class=your.package.Simple2Shell

####################
# Listeners Section #
####################
listeners=std
```

```
# std listener specific properties

#Basic listener and connection management settings
std.port=6666
std.floodprotection=5
std.maxcon=25


# Timeout Settings for connections (ms)
std.time_to_warning=3600000
std.time_to_timedout=60000

# Housekeeping thread active every 1 secs
std.housekeepinginterval=1000

std.inputmode=character

# Login shell
std.loginshell=simple

# Connection filter class
std.connectionfilter=none
```

To make the tutorial complete here the command for running the example:

```
java -classpath telnetd.jar net.wimpi.telnetd.TelnetD test.properties
```

As well as an output of the example:

```
[Fangorn:~] wimpi$ telnet localhost 6666
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

Then the screen will be erased, and you should end up with the following:

```
Connected from: localhost[0:0:0:0:0:0:0:1:53635]
Guessed Locale: en

Negotiated Terminal Type: VT100
Negotiated Columns: 130
Negotiated Rows: 24
Switching to Simple2Shell
Simple2Shell
key2=value2
key1=value1
key4=value4
key3=value3
Goodbye!
Connection closed by foreign host.
[Fangorn:~] wimpi$
```

Note:

Logically the data values from the above output are likely to differ when you are running it (i.e. the address, the guessed locale, the terminal type etc.)