Using eclipse to debug applications on NonStop

Cor Geboers
cor.geboers@hp.com
EMEA NonStop Competence Center
Hewlett-Packard

The goal of this document is to give you an introduction on how you can use the OpenSource tool eclipse to debug Java applications running on the HP NonStop Server. It assumes that you have a basic knowledge of eclipse and tells how to configure the server side to be able to connect to the remote server. See my other document “Eclipse development for NonStop” if you want to have an introduction to eclipse.

I have chosen 3 different scenarios that should help you debug most Java applications that you might encounter:

- Debugging a stand-alone application
- Debugging an NSJSP servlet
- Debugging a WebLogic Server application

**Debugging a stand-alone application**

This example assumes you are familiar with eclipse, at least so you can create a stand-alone Java application. In my example I have a class called “DebugMe” that does nothing fancy, just accepting a parameter and printing out a line for xx times. The source looks like this in the eclipse IDE:

![DebugMe source code in Eclipse IDE](image)

*Figure 1*
As you can see, I have a breakpoint enabled at line 39. This is done by double clicking in the vertical bar left of the source. The blue dot shows me where the actual breakpoint is. Hovering over the dot shows a tooltip with some extra information. (See figure 1)
After sending the jar file with the class over to the nonstop system, I run the program as follows:

```java
java -classpath ./debug_standalone.jar nscc.cg.debug.sa.DebugMe -count=5
```

This shows:

```
received this value for x: 0
received this value for x: 1
received this value for x: 2
received this value for x: 3
received this value for x: 4
```

To run this program in debugging mode I have to give it the correct parameters. Instead of putting them on the command line I use the `JAVA_OPTIONS` environment variable, which allows you to change options for Java without the need to give them as JVM arguments. While it is not very important in this case, it is very handy when you use shell scripts that start your java program. Instead of having to edit the shell script, I can now just change the value of my environment variable.

This is how I do it: (not: everything should occur on 1 line)

```
export _JAVA_OPTIONS="-Xdebug -Xnoagent -Djava.compiler=none
-Xrunjdwp:server=y,transport=dt_socket,address=16123,suspend=y"
```

Explanation of the parameters:

- `-Xdebug` Enable debugging mode
- `-Xnoagent` Disable the “traditional” debugging mode, use the JPDA agent
- `-Djava.compiler=none` Disable bytecode compilation
- `-Xrunjdwp:` JPDA specific connection information and settings
  - `server=y` Instructs the JVM to listen for the external debugger (eclipse) to attach at the specified address
  - `transport=dt_socket` Tell the debugger to use the network connection (tcp/ip)
  - `address=16123` If `dt_socket` is used, this is the TCP/IP listen port
  - `suspend=y` Suspend the program at startup. This means that the program should wait for the remote debugger to connect to it, before continuing.

To run my program, I do the same as above, or alternatively if I have an executable jar file, that is, where I have a manifest that tells me which class to call automatically, I can do this:

```
java -jar debug_standalone.jar -count=10
```
But let's stick with the example above and call the same command again:

```
java -classpath ./debug_standalone.jar nscc.cg.debug.sa.DebugMe -count=5
```

```
Picked up _JAVA_OPTIONS: -Xdebug -Xnoagent -Djava.compiler=none -Xrunjdwp:server=y,transport=dt_socket,suspend=y,address=16123
```

The line which starts with “Picked up” shows me which flags have been accepted by the JVM. Notice that now there is no output from the program, it is waiting for the debugger to connect to it. So next I have to configure my debugger in the eclipse IDE. I do this as follows:

On the menu choose **Run -> Debug ...** This will show this screen:

![Screen shot of Eclipse Debug configuration](image)

--- 4 ---
Select “Remote Java Application” and either do a right mouse-click on this item and select “New”, or click on the “New” button to get this screen:

![Figure 3](image)

As name use the proposed one (DebugMe) or specify whatever you like. The connection type defaults to Standard, which corresponds to the dt_socket we specified above. The Host is the name or IP address of our NonStop Server. Make sure that you use the one that corresponds to the TCP/IP process you use when you run the Java program. The Port is the port specified in the address field in the runjwdp options. Since I used 16123 there, I copy that value here.

Start debugging by clicking on the “Debug” button on the bottom of the dialog. Eclipse will ask you if you want to use the “Debug perspective”. Click yes to have a better overview of all windows:

![Figure 4](image)
My screen looks now like this:

![Debugging Window](image)

**Figure 5**

As you can see, the debugger started my program and halted at the first breakpoint it encountered. In this case it is the start of my program, but it might be completely different in your environment.

The window at the bottom-left contains the source of the debugging program. Remember, blue dots are breakpoints, where the debugger will halt the code. The windows above it (called "Debug") contains an overview of all the running threads in the Java program. The window at the top-right ("Variables") contains a tab for the breakpoints and one for the variables from the current scope. You will see this change whenever a new method is called for example. Currently it points to the variables known in the main method. There is an integer count and a String[] args.
The “Variables” window allows you to look at the contents of these variables. When you continue with the program, using the “Step” commands, this window will emphasize all changes by coloring them red:

![Figure 6](image)

Stepping can be done by either click on the buttons in the Debug-toolbar, or by using function keys F5 (step in) and F6 (step over).

![Figure 7](image)

Press “resume” to continue with the program and hit the next breakpoint. From here the control can be taken over and you can inspect/display all variables. It is enough to hover the pointer over a variable in the source code to see its value popup as a tooltip:

![Figure 8](image)

This simple example is usually not what you want to debug, but the same technique can be used to debug servlets and EJBs as I will show in the next topics.
Debugging servlets in NSJSP 3.0

Using the same technology as above, I can instrument the NSJSP servlet container to debug it with eclipse. Instead of using the parameters for debugging on the commandline or in the _JAVA_OPTIONS environment variable, I include them in my "servlet.config" file, which is used at startup of the iTP Webserver.

Here is the actual text I used:

```
# Server $server_objectcode {
  CWD $env(NSJSP_HOME)
  Env CLASSPATH=$JVCP:$USRCP
  Env JAVA_HOME=$env(JAVA_HOME)
  Env JREHOME=$env(JAVA_HOME)/jre
  Env TCPIP_RESOLVER_ORDER=HOSTFILE-DNS
  MapDefine =TCPIP^PROCESS^NAME /G/ZSAM1
  Maxservers 1
  Maxlinks 250
  Numstatic 1

# Linkdepth 25
  Env TANDEM_RECEIVEDEPTH=25

# File locations to direct standard input, output and error.
# Stdin /dev/null
  Stdout $root/logs/servlet.log
  Stderr $root/logs/servlet_error.log
# This is the actual Arglist used to start up the NSJSP Container.
# Note: all this should be on one line
# Arglist -Xdebug -Xnoagent -Djava.compiler=none
  -Xrunjdwp:server=y,transport=dt_socket,suspend=n,address=16124
  -Xnoclassgc –Xmx64m -Xss128k -Dbrowserdebug=false
  -Djdbc.drivers=com.tandem.sqlmp.SQLMPDriver $NSJSP_SECMGR $NSJSP_SECMGR_POLICY -Djava.endorsed.dirs=$env(NSJSP_ENDORSED_DIRS)
  -Dcatalina.home=$env(NSJSP_HOME)
  -Djava.io.tmpdir=$env(NSJSP_HOME)/temp com.tandem.servlet.NSJSPBootstrap -config $env(NSJSP_HOME)/conf/iTP_server.xml start
}
```

Lines in blue are what I actually changed from the default. Here is some explanation to why I needed to do this.

1. **Maxservers 1 and Numstatic 1:**
   Since we try to debug this environment, we need to be sure that our eclipse IDE connects to the correct process. Remember that we are now listening to a TCP/IP socket. Although PTCP/IP and TCP/IPv6 allow you to have multiple listeners on the same port, we would not be sure if the IDE connects to the same instance where my client or browser calls the servlet. By setting these values to 1, there is no doubt possible.
2. **-Xdebug -Xnoagent -Djava.compiler=none -Xrunjdwp:server=y,transport=dt_socket,suspend=n,address=16124**
   This looks exactly the same as in the standalone example, no? Except for one small change: `suspend=n`. This tells the JVM not to wait until the debugger connects, but to continue working and just listen to the port (16124). This is necessary, because the servlet engine itself is a Java program that needs to initialize itself before it can actually accept requests from clients. If you have a servlet engine that hangs during startup when debugging, check this parameter.

3. **Filemap /debug $server_objectcode**
   Not really needed for debugging, but it needs to be there, so the httpd knows that it should forward requests for the /debug context to the servlet engine and not process it.

   As an extra, here is the context I used in my application from the file iTP_server.xml:

   ```xml
   <Context path="/debug" docBase="/home/cor/mywork/debug/servlet" reloadable="true" debug="0"/>
   ```

   This tells the servlet engine where it should look for the files belonging to this web application.
Restart your webserver to take the new parameters and switch to your eclipse IDE. I used a breakpoint in my code at the first output statement, but you can use whatever you like:

**Figure 9**
Select the menu Run -> Debug ... and fill in the correct parameters:

![Debug Configuration](image)

Figure 10

The only difference with the previous example is that I used port 16124 for my servlet instead of the 16123 in my stand-alone example.
Switch to the debug perspective and you will see the DebuggeeServlet appear in your Debug window:

![Debug window showing active threads](image)

**Figure 11**

As you can see, there are a lot of threads active already. These were started by the Servlet engine to do different work. Next go to your favorite browser and call the servlet with a simple GET method:

Once this page is called, the code will activate the breakpoint that you set in the source and you need to switch to the eclipse IDE. In my case it shows this screen:

![Debugging Screen](image)

**Figure 12**

I have now the parameters param1 and param2 in the String variables as you can see from the Variables window on the top-right in figure 12. The Debug window shows which thread is active (not surprisingly it is doGet()) and the arrow on the left of my source code shows where in the code I am located.

Next use the Step commands, just as before to go over the code and look at what is done. At the end of the doGet method, control will be given back to the Servlet engine, so don’t forget to do "Resume" (F8), otherwise you will start debugging the Tomcat code and that is usually not what you want to do.
And finally your browser will show this:

![Figure 13](image)

This is class nscc.cg.debug.servlet.DebuggeeServlet, using the GET method

Parameter 1 = tester
Parameter 2 = 333
Debugging a WebLogic Server application

The third example here shows how to debug a servlet and an EJB running inside a WLS 8.1 SP2 environment. If you have read the previous sections, it will turn out to be very similar. After all I am not talking about rocket science here ;-) 

Since WLS is usually started with a shell script I use the \_JAVA\_OPTIONS variable again, to avoid that I have to change the script in between different sessions. For this experiment I created a small servlet (called DebugMeServlet) and an even simpler EJB (simply called DebugMe). The servlet simply accepts 2 parameters (as before) and forwards them to the “business” method in the stateless EJB, which concatenates the strings after converting them to uppercase. Nothing fancy and surely not the way you want to write this in a real application. Remember the code here is just for amusement of real programmers.

After creating an ear file with both applications, I deployed it in the WebLogic Server and can access them. To get my WLS server in debugging mode I use the \_JAVA\_OPTIONS variable:

```
export _JAVA_OPTIONS="$_JAVA_OPTIONS -Xdebug -Xnoagent
-Djava.compiler=none
-Xrunjdwp:server=y,transport=dt_socket,suspend=n,address=16125"
```

Again everything should be on one line ! The statement looks the same as with the NSJSP case, suspend=n, because we want the WLS server to do all it’s initialization stuff before we continue our work. Use your favorite startup script to start the server and switch to your eclipse IDE. I have set a breakpoint at some line in my servlet and another in the bean:

--- 15 ---
Similar for the EJB:

**Figure 15**

Use "Run -> Debug ..." to create a connection for the remote machine:

**Figure 16**
Again I used another port, but there is nothing different. Click on the “Debug” button and the IDE will connect with the remote server.

**Figure 17**

Use your browser again to connect to the environment:

```
```

As soon as you hit enter you will see that the eclipse IDE shows the breakpoint in the servlet.
Click on Resume (or press F8) and the next breakpoint (in the EJB) will be shown:
Just as before, use the necessary step commands to examine your source and the variables. Don’t forget to do “Resume” at the end, because you don’t want to debug the WLS server!
Resources

Eclipse: http://www.eclipse.org
CDT   http://www.eclipse.org/CDT