

**QuickServer**  
v 1.3

Volume

**1**



<http://www.quickserver.org>

# Developers Guide

**Akshathkumar Shetty**

QuickServer v 1.3.1

# Developers Guide

---

Date : 10 March 2004  
Author : Akshathkumar Shetty [ [akshath@quickserver.org](mailto:akshath@quickserver.org) ]  
Contributions By :  
    Sean Stallbaum [ [Sean.Stallbaum@nationalcity.com](mailto:Sean.Stallbaum@nationalcity.com) ]  
Version : 0.3  
Target : Developers having a good Java knowledge.  
Keywords : QuickServer, ServerSocket, Object, Socket,  
Library, Java, multithreading, multi-client,  
thread, pool.

© 2003-2004 Akshathkumar Shetty  
akshath@quickserver.org  
<http://www.quickserver.org> - <http://quickserver.sourceforge.net>

---

# Contents

<b>CONTENTS.....</b>	<b>II</b>
<b>1. INTRODUCTION.....</b>	<b>4</b>
NEED FOR QUICKSERVER .....	4
BASIC ARCHITECTURE.....	5
MAJOR FEATURES .....	6
WHAT'S NEW IN 1.3.1? .....	7
REQUEST TO DEVELOPERS.....	7
CREDITS .....	7
WEBSITE & EMAIL .....	7
<b>2. INSTALLATION.....</b>	<b>8</b>
PREREQUISITES.....	8
INSTALLATION.....	8
TESTING THE INSTALLATION.....	9
BAT FILES IN EXAMPLES FOLDER [WINDOWS 9X ONLY] .....	10
CODE FOR THIS BOOK .....	10
<b>3. BUILDING BASIC ECHOSERVER .....</b>	<b>11</b>
CODING .....	11
RUNNING AND TESTING.....	13
<b>4. ADDING AUTHENTICATION.....</b>	<b>15</b>
<b>5. USING CLIENTDATA .....</b>	<b>19</b>
MAKING CLIENTDATA POOLABLE.....	23
<i>Basics of PoolableObjectFactory.....</i>	<i>24</i>
<b>6. USING REMOTE ADMIN SUPPORT .....</b>	<b>27</b>
ADDING YOUR OWN COMMAND .....	30
<b>7. USING LOGGING AND CUSTOMISATION .....</b>	<b>35</b>
SIMPLE LOGGING.....	36
ADVANCED LOGGING .....	39
<b>8. XML CONFIGURATION .....</b>	<b>41</b>
<b>9. DATA MODES AND DATA TYPES.....</b>	<b>45</b>
<b>10. COMMUNICATION WITH OBJECT MODE.....</b>	<b>46</b>

<b>11. COMMUNICATION WITH BYTE MODE .....</b>	<b>47</b>
<b>12. XML BASED JDBC MAPPING .....</b>	<b>48</b>

## 1. Introduction

*Java library to create multi-threaded multi-client TCP servers.*

**Q**uickServer is a free, open source Java library for quick creation of robust and multi-threaded, multi-client TCP server applications. It provides an abstraction over the `java.net.ServerSocket` class; it eases the creation of powerful server applications. It has been designed and implemented by Akshatkumar Shetty.

Example programs demonstrating the use of the library can be found with the QuickServer distribution [examples folder]. Latest examples, documentation is also available through the website <http://www.quickserver.org> or <http://quickserver.sourceforge.net>

This guide is for any developer who wants to learn and use QuickServer. You should have basic knowledge of coding in java and a basic knowledge of networking and sockets will help.

---

### ICON KEY



Valuable information



Tip & Tricks



Java Code



Caution

Icon Keys used this guide uses the following icon keys for better visual appeal.

### ***Need for QuickServer***

In any programming language, socket programming is no small feat to any programmer, and creating a multi-threaded multi-client server socket is a nightmare for most programmers. We always waste time in writing the same code each time we build new software, which handles multiple socket connections. So I have made a library - QuickServer, to handle a creation of multi-threaded, multi-client server applications for Java.

## **Basic Architecture**

QuickServer divides the application logic of its developer over four classes,



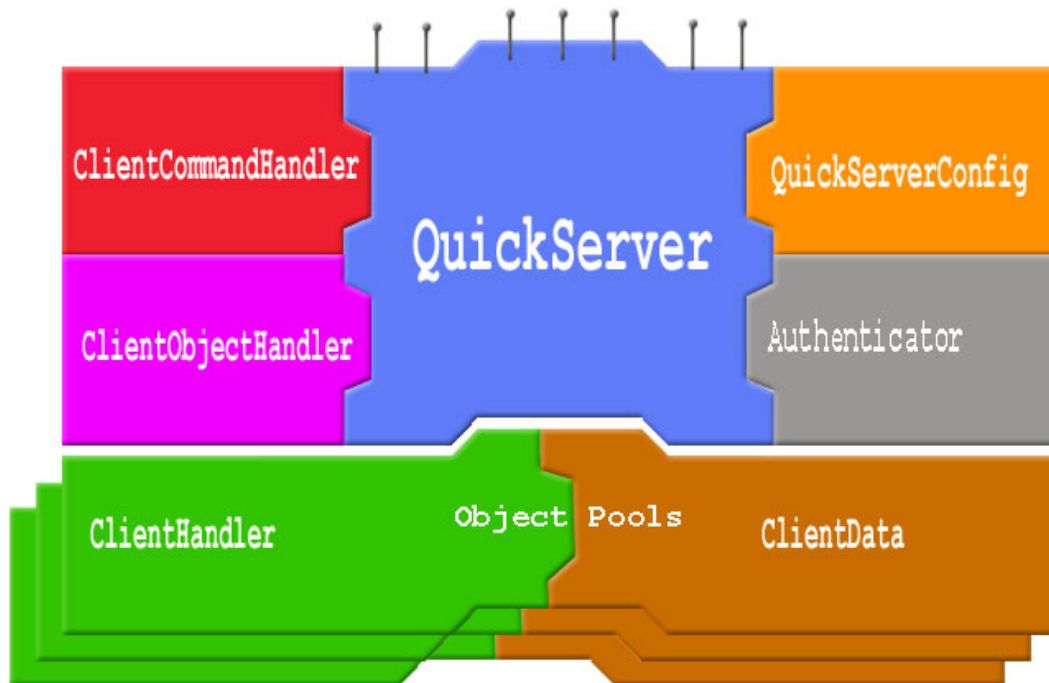
- ClientCommandHandler  
Handle client interaction - String commands.
- ClientObjectHandler [Optional Class]  
Handle client interaction - Object commands.
- Authenticator [Optional Class]  
Used to authenticate a client.
- ClientData [Optional Class]  
Client data carrier (support class)

The diagram below shows the basic architecture of QuickServer library. The seven spokes on the QuickServer block are the seven Service methods.

- `java.lang.String info()`
- `int getServiceState()`
- `boolean initService(java.lang.Object[] config)`
- `boolean startService()`
- `boolean resumeService()`
- `boolean suspendService()`
- `boolean stopService()`

Of the four components connected to QuickServer block only ClientCommandHandler class is the absolutely necessary class.

QuickServerConfig object is constructed by `initService()` method of the Service interface, that QuickServer implements, after reading configuration from XML file. QuickServerConfig is then used to configure QuickServer.



**Figure 1-1**

ClientHandler thread object is used from the pool of object for every client connected; optional ClientData class is associated with the ClientHandler class. ClientHandler object contain references to ClientCommandHandler, ClientObjectHandler(optional), Authenticator(optional) objects contained with in the main QuickServer.

NOTE: QSAdminServer is not shown in this diagram, it is a composed QuickServer within the main QuickServer.

See the [architecture.pdf](#) for more on architecture of QuickServer.

### **Major Features**

- Create multi-threaded, multi-client TCP server applications.
- Clear separation of server, protocol and authentication logic.
- Remote administration support with GUI.
- Restart or Suspend the server without killing connected clients.
- In build pools for reusing of Threads and most used Objects.
- Full logging support [Java built in Logging].
- Support for sending and receiving Strings.
- Support for sending and receiving Bytes.
- Support for sending and receiving serialized java objects.
- Support for xml configuration.

- Support for xml Based JDBC Mapping
- Support for Service Configurator pattern.
- Specify maximum number of clients allowed.
- Easy methods to identify and search any specific client.
- Nice easy examples come with the distribution - FTPServer, CmdServer, EchoWebServer, ChatServer.

### ***What's New in 1.3.1?***

- Added BYTE mode of communication.
- Added a new interface ClientIdentifiable and methods to identify and search a client
- Added new method to ClientHandler to find the time when Client was assigned to ClientHandler.
- Improved logging in ClientHandler to log all outgoing and incoming communications at FINE level.
- Improved error detection for Authenticator implementations.

### ***Request to Developers***

If you would like to contribute to the development of QuickServer please do get in touch with me. I am always on the lookout for people who can contribute to make this library even better.

If you use QuickServer in your development and if you would like to share your experience with the QuickServer community, please feel free to email me the details or do post it in the QuickServer Forums. If possible do mail across the source code of your project. If found interesting your project will be showcased on the powered projects section of QuickServer's homepage. Thanks.

### ***Credits***

Thanks to everyone who helped me in this project. Thanks to all users who sent their valuable comments and suggestion.

### ***Website & Email***

URL : <http://www.quickserver.org>  
URL : <http://quickserver.sourceforge.net>  
Email : [akshath@quickserver.org](mailto:akshath@quickserver.org)



## 2. Installation

### *Prerequisites*

#### **For QuickServer v 1.2 and above**

- Recommended JVM : v1.4 or above.
- Minimum JVM : v1.3 {Not Tested}.
- Java Logging API (any one)
  - java.util.logging package [Comes with JDK 1.4]
  - Lumberjack library [<http://javalogging.sourceforge.net/>]
- XML parser (any one)
  - SAX (Simple API for XML 2.0) [Comes with JDK 1.4]
  - JAXP (Java API for XML Parsing) 1.1 [Comes with JDK 1.4]
  - Xerces [<http://xml.apache.org/xerces2-j>]
  - Crimson [<http://xml.apache.org/crimson>]
- Jakarta Commons Components {Digester, Pool}
  - This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>). Jar files are shipped with library along with BeanUtils, Collections, Logging. [<http://jakarta.apache.org/commons/components.html>]  
Apache Software License is included in the file "apache\_license.txt" for your convenience.

#### **For QuickServer v 1.1 and bellow**

- JDK 1.3 or above

### *Installation*

- 1) Using the setup or compressed archive extract QuickServer into a directory of your choice say c:\QuickServer
- 2) Add the "c:\QuickServer\dist\QuickServer.jar" file to the CLASSPATH
- 3) If you are using JDK 1.4 or above skip Step to 'Testing the installation'.
- 4) Installing the Lumberjack Java logging API for JVM bellow 1.4
  - Download Lumberjack library from <http://javalogging.sourceforge.net/>
  - Extract the jar file to using following command  
jar -vxf Lumberjack-X.X.X.X.jar

- Now go to the extracted dir say c:\Lumberjack-0.9.3.2\ and give the following command  
    java -jar lib\logging.jar
- 5) Installing XML parsers

### **Testing the installation**

1. Open the command prompt.
2. Navigate to the installation folder c:\QuickServer\examples by typing command like:

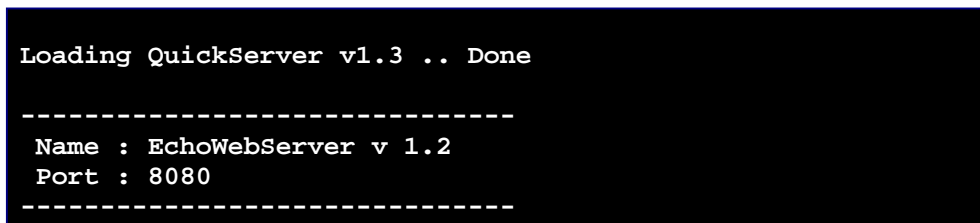
```
cd c:\QuickServer\examples
or
cd /QuickServer/examples/
```

3. Type one of the following commands:

```
java echowebserver.EchoWebServer
or
java com.ddost.net.server.QuickServer
conf\EchoWebServer.xml
```

It reads configuration from xml file using QuickServer. Do not click on the EchoWebServer.bat it will work even if you don't set the path.

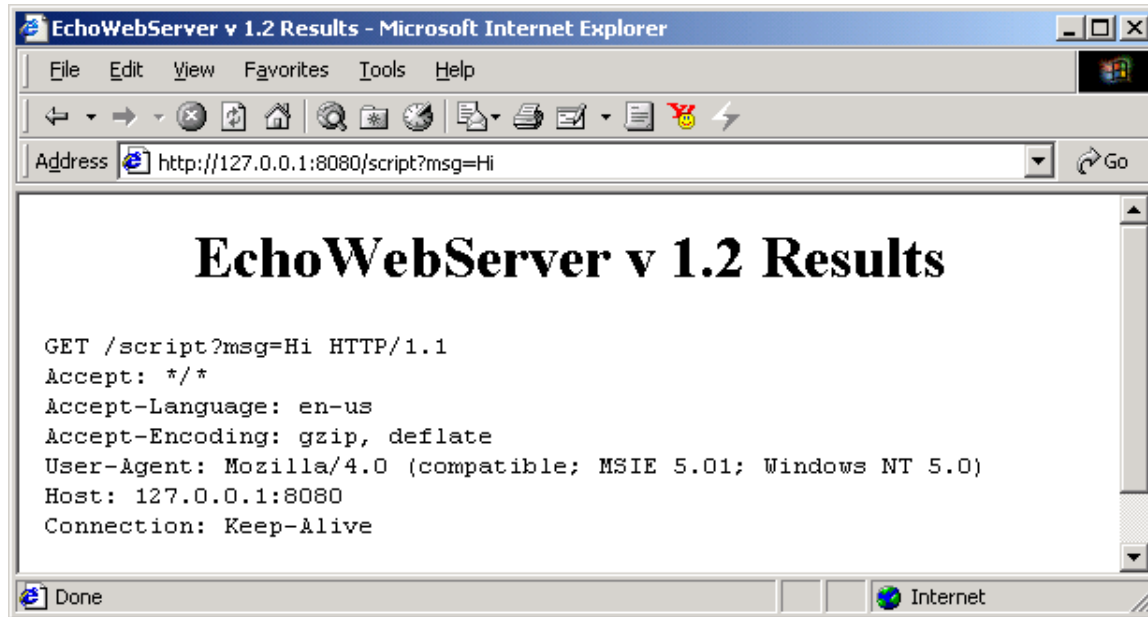
It should display output similar to the figure 2-1 below:



```
Loading QuickServer v1.3 .. Done
-----
Name : EchoWebServer v 1.2
Port : 8080
-----
```

**Figure 2-1**

4. Open your browser to: <http://127.0.0.1:8080/script?msg=Hi>. You should get the response similar to this.



### ***Bat files in Examples folder [Windows 9x only]***

Some when you click on the EchoWebServer.bat, you get some thing like this then follow the instruction given bellow

```
Out of environment space
Exception in thread "main" java.lang.NoClassDefFoundError:
echowebserver/EchoWebServer
```

**Figure 2-3**

Close the window, and right click on the file **EchoWebServer.bat** select properties Click on 'Memory' tab, and Change 'Initial environment' to '4096' or above. Click OK.



Try this step again but now instead of clicking **EchoWebServer.bat** click on the MSDOS icon.

### ***Code for this book***

To use this book effectively you will also need the code, you must have received it with the book, if not do download it from QuickServer.org and extract it to c:\Quickserver\tutor.

## 3. Building Basic EchoServer

A good way to learn how to use QuickServer library is to study the examples provided with it. And even better way is to writing a server using the QuickServer library. That's what we will be doing in the rest of this guide. Though the server that we build is not that useful, it will demonstrate all the features that QuickServer library provides you with. We start by building a basic server and then keep on adding a new feature in each chapter.

### Coding

So what is the server we will be building? It is called EchoServer – It's a simple TCP server that echoes back any line of input that user sends with and **'Echo :'** prefixed.

- Create a director at some location [ say c:\projects\java\ ] called **'echoserver'**.
- Now lets create the main class of echoserver **'EchoServer.java'** save it in the directory **'echoserver'** 📁

```
01 package echoserver;
02
03 import org.quickserver.net.*;
04 import org.quickserver.net.server.*;
05
06 import java.io.*;
07
08 public class EchoServer {
09     public static void main(String s[]) {
10         QuickServer myServer =
11             new QuickServer("echoserver.EchoCommandHandler");
12         myServer.setPort(4123);
13         myServer.setName("EchoServer v 1.0");
14         try {
15             myServer.startServer();
16         } catch(AppException e){
17             System.err.println("Error in server : "+e);
18         }
19     }
20 }
```

Listing 3 - 1

When we define [Line: 10 & 11] the QuickServer object myServer we pass it a String object that says which class the QuickServer must load and use as its command handler for any client.

This class is a user created class that implements org.quickserver.net.server.ClientCommandHandler interface. Next, [Line: 12] we set the port that the server will listen on and then set the Application's name [Line: 13].

Finally, we actually start the server [Line: 15].

- Now lets create the command handler class for our echoserver, that handles commands sent to echo server, '**EchoCommandHandler.java**' save it in the directory '**echoserver**'.

```
01 // EchoCommandHandler.java
02 package echoserver;
03
04 import java.net.*;
05 import java.io.*;
06 import org.quickserver.net.server.ClientCommandHandler;
07 import org.quickserver.net.server.ClientHandler;
08
09 public class EchoCommandHandler implements ClientCommandHandler {
10
11     public void gotConnected(ClientHandler handler)
12         throws SocketTimeoutException, IOException {
13         handler.sendClientMsg("+++++++");
14         handler.sendClientMsg("| Welcome to EchoServer v 1.3 |");
15         handler.sendClientMsg("|           Send 'Quit' to exit |");
16         handler.sendClientMsg("+++++++");
17     }
18     public void lostConnection(ClientHandler handler)
19         throws IOException {
20         handler.sendSystemMsg("Connection lost : " +
21             handler.getSocket().getInetAddress());
22     }
23     public void closingConnection(ClientHandler handler)
24         throws IOException {
25         handler.sendSystemMsg("Closing connection : " +
26             handler.getSocket().getInetAddress());
27     }
28
29     public void handleCommand(ClientHandler handler, String command)
30         throws SocketTimeoutException, IOException {
31         if(command.equals("Quit")) {
32             handler.sendClientMsg("Bye ;-);");
33             handler.closeConnection();
34         } else {
35             handler.sendClientMsg("Echo : "+command);
36         }
37     }
38 }
```

## Listing 3 - 2

This class implements the `org.quickserver.net.server.ClientCommandHandler` interface [Line: 9] as required by `QuickServer`.

When client gets connected [Line: 11], `gotConnected()` function of the `ClientCommandHandler` is called, so in this method we send some welcome text to the client [Line: 13-16]. The text is sent the client using the `sendClientMsg()` method of the passed `ClientHandler` object. We also display [log] the `INetAddress` of the connected client to the consol [Line: 20-21, 25-26] using `sendSystemMessage()` method of the passed `ClientHandler` object.

The `handlerCommand()` method [Line:29] is the focal method in the `ClientCommandHandler` interface because this method is called every time server receives any line of input from the client. In our implementation, we check to see if the sent command is equal to 'Quit' [Line: 31] and if it is then we send some text back to user indicating server will close connection and close the connection [Line: 33], else we send same text back to the user with 'Echo : ' prefixed to it.

### ***Running and Testing***

Now lets compile our new program...

- Go to command prompt (cmd.exe)
- Traverse to the base directory:  
`cd c:\projects\java`
- Compile the code  
`javac echoserver\*.java`
- Run the server [if no errors when compiling]  
`java echoserver.EchoServer`
- You should get something like this

```
Loading QuickServer v1.3 .. Done
-----
Name : EchoServer v 1.0
Port : 4123
-----
```

**Figure 3-1**

- Testing to see if our server is working, as we want it to. For this step we can use any socket communication software like telnet. But I like `SocketTest` – a java swings base tool.  
Download it from <http://www.ddost.com/soft/sockettest/>

Start the socket communication program [double click on sockettest.jar file or from the command prompt type the following command `java -jar sockettest.jar`]

In the open window connect to the localhost on port 4123.

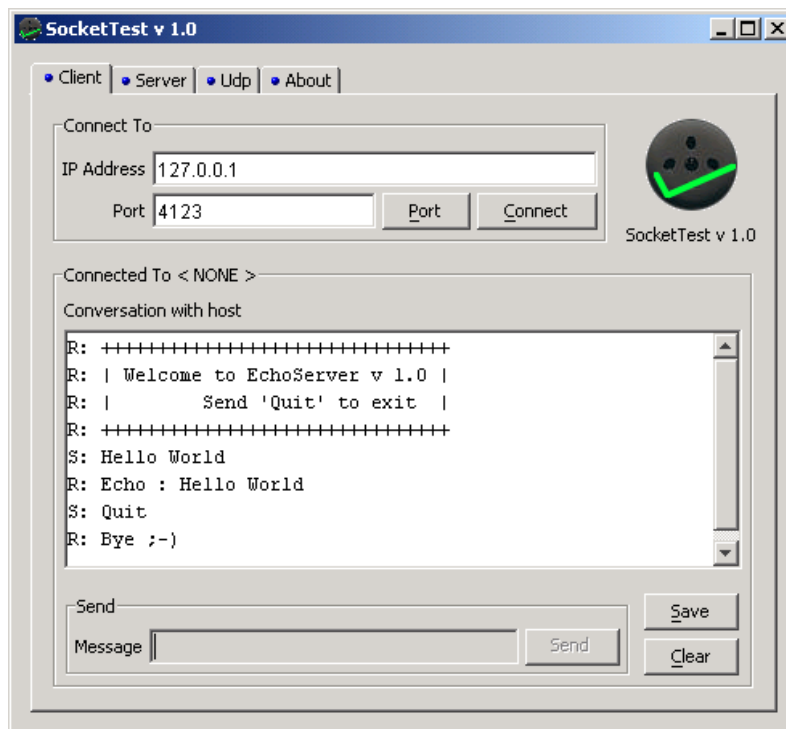
For SocketTest

- Enter IP as 127.0.0.1
- Port as 1234
- Click on 'Connect' button.

For telnet use command:

- open localhost 4123

Once connection is made you should see a banner like the figure below.



**Figure 3-2**

Try sending some strings to the server and see what it send back. Send 'Quit' to disconnect from the server.

## 4. Adding Authentication

In the last chapter we created a basic server. Now lets add some authentication to it. If we look in the documentation [docs folder] of `org.quickserver.net.server.QuickServer` you will notice it has a method like

```
public void setAuthenticator(java.lang.String
                             authenticator)
```

Further reading of the documentation tells us that **authenticator** is the fully qualified name of the class that implements

`org.quickserver.net.server.Authenticator` class.

There are two implementation of Authenticator class, namely

```
org.quickserver.net.server.QuickAuthenticator
```

This class is used to authenticate a client when it connects to QuickServer. Only single instance of this class will be used per QuickServer to handle all authentications. (Recommended implementation).

```
org.quickserver.net.server.ServerAuthenticator
```

This class is used to authenticate a client when it connects to QuickServer Every client connected will have separate instance of this class which will be used to handle the authentication.

Lets click on the `QuickAuthenticator` it shows us a simple example of how we can write our own authenticator class for QuickServer, so lets us write one for our EchoServer.

So how will our authenticator authenticate our client, to keep it simple, client who supplies a password that is same as the username is authenticated.

1. Go to the directory created in the previous chapter [say `c:\projects\java\echoserver`].
2. Now lets create the authenticator class of echoserver '**EchoServerQuickAuthenticator**' save it in the directory '**echoserver**'





```

01 package echoserver;
02
03 import org.quickserver.net.server.*;
04 import java.io.*;
05
06 public class EchoServerQuickAuthenticator extends QuickAuthenticator {
07
08     public boolean askAuthorisation(ClientHandler clientHandler)
09         throws IOException {
10         String username = askStringInput(clientHandler, "User Name :");
11         String password = askStringInput(clientHandler, "Password :");
12
13         if(username==null || password ==null)
14             return false;
15
16         if(username.equals(password)) {
17             sendString(clientHandler, "Auth OK");
18             return true;
19         } else {
20             sendString(clientHandler, "Auth Failed");
21             return false;
22         }
23     }
24 }

```

#### Listing 4 - 1

This class extends the `org.quickserver.net.server.QuickAuthenticator` class. [Line: 6].

In the method `askAuthorisation()` [Line: 8] we ask username from the client [Line: 10] by using `askStringInput()` method inherited from `QuickAuthenticator` class. The same method `askStringInput()`, will read the client input [Line: 10]. If the password matches the username supplied [Line: 16] we return true indicating client was authorised else we return false [Line: 21].

Now lets tell QuickServer to use our new authenticator as its `Authenticator`. Edit the java file `EchoServer.java` created in the previous chapter to look like this [Changes shown in **bold**].

```

01 package echoserver;
02
03 import com.ddost.net.*;
04 import com.ddost.net.server.*;
05
06 import java.io.*;
07
08 public class EchoServer {
09
10     public static void main(String s[]) {
11

```

```

12     QuickServer myServer =
13         new QuickServer("echoserver.EchoCommandHandler");
14     myServer.setAuthenticator(
15         "echoserver.EchoServerQuickAuthenticator");
16     myServer.setPort(4123);
17     myServer.setName("EchoServer v 1.0");
18     try {
19         myServer.startServer();
20     } catch(AppException e){
21         System.err.println("Error in server : "+e);
22     }
23 }
24 }
25

```

**Listing 4 - 2**

Now lets compile our changed program. Run it as described the previous chapter. When you try to connect this time it will ask you for a username and password. If you supply user name equals to the password it will log you in If you try wrong password for more that five times it will disconnect the client printing '-ERR Max Auth Try Reached'. This can be changed using setMaxAuthTry() and setMaxAuthTryMsg() method of QuickServer class. Bellow is a sample output.

```

R: ++++++
R: | Welcome to EchoServer v 1.0 |
R: | Send 'Quit' to exit |
R: ++++++
R: User Name :
S: user
R: Password :
S: user
R: Auth OK
S: Hello
R: Echo : Hello

```

Some time when authorising we may need to close the client connection instead of letting him try again we can do this in two ways

→ Throw a AppException like this from askAuthorisation() method like this

```

String username = askStringInput(clientHandler,
    "User Name :");
if(username!=null &&
    username.equalsIgnoreCase("QUIT")) {
    sendString(clientHandler, "Logged out.");
    throw new AppException("Quit");
}

```

→ Get reference to `ClientHandler` and then close the connection. Like this

```
String username = askStringInput(clientHandler,
    "User Name :");
if(username!=null &&
    username.equalsIgnoreCase("QUIT")) {
    sendString(clientHandler, "Logged out.");
    ClientHandler ch = getClientHandler();
    ch.closeConnection();
    return false;
}
```

**Note:** `ClientHandler` object can provide many useful information about client connected like ip address, refer to the API docs for more information.

**Warning :**

- ☛ Do not store any client related information in the authenticator class, if you need you must put it in the `ClientData` class – covered in the next chapter.
- ☛ You must also make sure the `askAuthorisation()` method is thread safe.

## 5. Using ClientData

There have been a lot of references to the ClientData class in the previous chapters, though if we examine the QuickServer API docs it has no method in it.

Since the ClientCommandHandler and ServerAuthenticator class can't hold any client data we use ClientData class to store any client information between to function class to handleCommand() or askAuthorisation().

To demonstrate how this is a useful feature you need to know, lets say our EchoServer will greet the user with his username if he sends 'Hello' to it. And if user sends 'Hello' more than ones it will replay to him telling him he has told Hello n times including this Hello. So lets define out ClientData class to store the username and count of hello told to the server by the client.

1. Go to the directory created in the Chapter 3 [say c:\projects\java\echoserver].
2. Now lets create the ClientData class of echoserver '**EchoServerData.java**' save it in the directory '**echoserver**'

```
01 //---- EchoServerData.java ----
02 package echoserver;
03
04 import org.quickserver.net.server.*;
05 import java.io.*;
06
07 public class EchoServerData implements ClientData {
08     private int helloCount;
09     private String username;
10
11     public void setHelloCount(int count) {
12         helloCount = count;
13     }
14     public int getHelloCount() {
15         return helloCount;
16     }
17
18     public void setUsername(String username) {
19         this.username = username;
20     }
21     public String getUsername() {
```

```
22     return username;
23 }
24 }
25 //--- end of code ---
```

**Listing 5 - 1**

Now lets tell QuickServer to use our EchoServerData as its ClientData class.

Edit the java file EchoServer.java created in the previous chapter to look like this [Changes shown in **bold**].

```
01 package echoserver;
02
03 import org.quickserver.net.*;
04 import org.quickserver.net.server.*;
05
06 import java.io.*;
07
08 public class EchoServer {
09     public static void main(String s[]) {
10
11         String cmd = "echoserver.EchoCommandHandler";
12         String auth = "echoserver.EchoServerQuickAuthenticator";
13         String data = "echoserver.EchoServerData";
14
15         QuickServer myServer = new QuickServer(cmd);
16         myServer.setAuthenticator(auth);
17         myServer.setClientData(data);
18
19         myServer.setPort(4123);
20         myServer.setName("Echo Server v 1.0");
21         try {
22             myServer.startServer();
23         } catch (AppException e){
24             System.out.println("Error in server : "+e);
25         }
26     }
27 }
28
```

**Listing 5 - 2**

In the above we have put our configuration information into string variables and then used these variable to set the QuickServer.

3. Now we need to modify our Authenticator class i.e., EchoServerAuthenticator class so that it stores the username in the ClientData object. Bellow is the modified code:

```
01 package echoserver;
02
03 import org.quickserver.net.server.*;
04 import java.io.*;
05
06 public class EchoServerQuickAuthenticator extends QuickAuthenticator {
07
08     public boolean askAuthorisation(ClientHandler clientHandler)
09         throws IOException {
10         String username = askStringInput(clientHandler, "User Name :");
11         if(username!=null && username.equalsIgnoreCase("QUIT")) {
12             sendString(clientHandler, "Logged out.");
13             //close the connection
14             clientHandler.closeConnection();
15             return false;
16         }
17
18         String password = askStringInput(clientHandler, "Password :");
19
20         if(username==null || password ==null)
21             return false;
22
23         if(username.equals(password)) {
24             sendString(clientHandler, "Auth OK");
25             //store the username in ClientData
26             EchoServerData data = (EchoServerData)clientHandler.getClientData();
27             data.setUsername(username);
28             return true;
29         } else {
30             sendString(clientHandler, "Auth Failed");
31             return false;
32         }
33     }
34 }
```

### Listing 5 - 3

4. Now we need to modify our ClientCommandHandler implementation class i.e., EchoCommandHandler.java so that it greet the user with his username if he sends 'Hello' to it. And if user sends 'Hello' more than ones it will replay to him telling him he has told Hello n times including this Hello.

Bellow is the modified code

```
01 // EchoCommandHandler.java
02 package echoserver;
03
04 import java.net.*;
05 import java.io.*;
06 import org.quickserver.net.server.ClientCommandHandler;
07 import org.quickserver.net.server.ClientHandler;
08
09 public class EchoCommandHandler implements ClientCommandHandler {
10
11     public void gotConnected(ClientHandler handler)
12         throws SocketTimeoutException, IOException {
13         handler.sendClientMsg("+++++++");
14         handler.sendClientMsg("| Welcome to EchoServer v 1.0 |");
15         handler.sendClientMsg("| Note: Password = Username |");
16         handler.sendClientMsg("| Send 'Quit' to exit |");
17         handler.sendClientMsg("+++++++");
18     }
19     public void lostConnection(ClientHandler handler)
20         throws IOException {
21         handler.sendSystemMsg("Connection lost : " +
22             handler.getSocket().getInetAddress());
23     }
24     public void closingConnection(ClientHandler handler)
25         throws IOException {
26         handler.sendSystemMsg("Closing connection : " +
27             handler.getSocket().getInetAddress());
28     }
29
30     public void handleCommand(ClientHandler handler, String command)
31         throws SocketTimeoutException, IOException {
32         if(command.equals("Quit")) {
33             handler.sendClientMsg("Bye ;-");
34             handler.closeConnection();
35         } if(command.equalsIgnoreCase("hello")) {
36             EchoServerData data = (EchoServerData) handler.getClientData();
37             data.setHelloCount(data.getHelloCount()+1);
38             if(data.getHelloCount()==1) {
39                 handler.sendClientMsg("Hello "+data.getUsername());
40             } else {
41                 handler.sendClientMsg("You told Hello "+data.getHelloCount()+
42                     " times. ");
43             }
44         } else {
45             handler.sendClientMsg("Echo : "+command);
46         }
47     }
48 }
```

Listing 5 - 4

5. Now lets compile our changed program.
6. Run it as described the previous chapters.
7. When you try to connect this time it will ask you for username and password. If you supply a user name equal to the password it will log you in.
8. If you enter 'Hello' it will greet you with your username and you send hello again it will tell how many times you sent hello to it. Bellow is a sample output.

```

R: ++++++
R: | Welcome to EchoServer v 1.0 |
R: | Note: Password = Username |
R: | Send 'Quit' to exit |
R: ++++++
R: User Name :
S: Akshath
R: Password :
S: Akshath
R: Auth OK
S: Hello World
R: Echo : Hello World
S: Hello
R: Hello Akshath
S: Hello
R: You told Hello 2 times.
S: quit
R: Echo : quit
S: Quit
R: Bye ;- )
R: Echo : Quit

```

## ***Making ClientData Poolable***

The code that we have written till now that uses ClientData works fine, but the problem is that for every client connected QuickServer will create a new object of ClientData. This can be performance bottleneck, in high performance production servers.

This problem can be slowed by telling QuickServer that it has to, create a pool of objects of ClientData and use objects from this pool whenever a client connects to it. We can do this implementing the interface

```
org.quickserver.util.pool.PoolableObject
```

If we look into QuickServer API documentation we find that PoolableObject has only two methods that must be implemented

```

getPoolableObjectFactory() which returns an object of
org.apache.commons.pool.PoolableObjectFactory which
belongs to Commons Pool library.
isPoolable() which returns if Object is poolable or not (boolean)

```



## Basics of PoolableObjectFactory

Interface `org.apache.commons.pool.PoolableObjectFactory` contains following method that need to be implemented

- `void activateObject(Object obj)`  
Reinitialise an instance to be returned by the pool.
- `void destroyObject(Object obj)`  
Destroys an instance no longer needed by the pool.
- `Object makeObject()`  
Creates an instance that can be returned by the pool.
- `void passivateObject(Object obj)`  
Uninitialise an instance to be returned to the pool.
- `boolean validateObject(Object obj)`  
Ensures that the instance is safe to be returned by the pool.

There is also a basic no-operation implementation that we can extend to make our object poolable this class is

`org.apache.commons.pool.BasePoolableObjectFactory` this class only has `makeObject()` method has abstract and the `validateObject()` function will always return true.

So let write the code so that our `ClientData` class is also poolable let it be `EchoServerPoolableData`, bellow is the code.

```
01 //---- EchoServerPoolableData.java ----
02 package echoserver;
03
04 import org.quickserver.net.server.*;
05 import java.io.*;
06
07 public class EchoServerPoolableData
08     extends EchoServerData
09     implements org.apache.commons.pool.PoolableObjectFactory {
10
11     public void activateObject(Object obj) {
12     }
13     public void destroyObject(Object obj) {
14         if(obj==null) return;
15         passivateObject(obj);
16         obj = null;
17     }
18     public Object makeObject() {
19         return new EchoServerPoolableData();
20     }
21     public void passivateObject(Object obj) {
22         EchoServerPoolableData pd = (EchoServerPoolableData)obj;
23         pd.setHelloCount(0);
24         pd.setUsername(null);
25     }
26     public boolean validateObject(Object obj) {
```

```

27     if(obj==null)
28         return false;
29     else
30         return true;
31     }
32 }
33 //--- end of code ---

```

The above code just extends our old ClientData and then we implement `org.apache.commons.pool.BasePoolableObjectFactory`. The implementation is simple so no explanation is needed.

Now we need to tell QuickServer to use this class instead of the old data class. Below is the code.

```

01 package echoserver;
02
03 import org.quickserver.net.*;
04 import org.quickserver.net.server.*;
05
06 import java.io.*;
07
08 public class EchoServer {
09     public static void main(String s[]) {
10
11         String cmd = "echoserver.EchoCommandHandler";
12         String auth = "echoserver.EchoServerQuickAuthenticator";
13         //String data = "echoserver.EchoServerData"; //Non-Poolable
14         String data = "echoserver.EchoServerPoolableData"; //Poolable
15
16         QuickServer myServer = new QuickServer(cmd);
17         myServer.setAuthenticator(auth);
18         myServer.setClientData(data);
19
20         myServer.setPort(4123);
21         myServer.setName("Echo Server v 1.0");
22         try {
23             myServer.startServer();
24         } catch (AppException e){
25             System.out.println("Error in server : "+e);
26         }
27     }
28 }
29

```

Now lets compile our changed program. This will, most likely will return an error saying package `org.apache.commons.pool` does not exist

This is because Compiler does not know this class, you will have to add the `commons-pool.jar` in QuickServer library installation directory (Say `c:\QuickServer`) this can be done by the following command on windows

```
Set CLASSPATH=%CLASSPATH%;c:\QuickServer\dist\commons-pool.jar
```

Run it as described the previous chapters. When you try to connect this time, the output of the server will be the same but ClientData class is not recreated for each client connected instead it will be reused.

## 6. Using Remote Admin Support

Our server EchoServer, works fine.. say we need to modify few server configuration like Timeout message or Max Auth try message or max times authorization try can be done per session (maxAuthTry) with out changing code, this is supported by QuickServer.

To do the above-mentioned task has well as much other control function without changing server code or even shutdown the server when it is running we need an admin server that will control our server. This server is implemented by QuickServer library has another QuickServer implementation called `org.quickserver.net.qsadmin.QSAdminServer`

To use this feature all we have to do is call `startQSAdminServer()` method of QuickServer object. The default port on which QSAdminServer will run on is 9876, this can be changed by using any of the two methods

```
setQSAdminServerPort(4124);  
(or)  
getQSAdminServer().getServer().setPort(4124);
```

Bellow is the new code that has enabled QSAdminServer for EchoServer on port 4124

```
01 package echoserver;  
02  
03 import org.quickserver.net.*;  
04 import org.quickserver.net.server.*;  
05  
06 import java.io.*;  
07  
08 public class EchoServer {  
09     public static void main(String s[]) {  
10  
11         String cmd = "echoserver.EchoCommandHandler";  
12         String auth = "echoserver.EchoServerQuickAuthenticator";  
13         String data = "echoserver.EchoServerPoolableData"; //Poolable  
14  
15         QuickServer myServer = new QuickServer(cmd);  
16         myServer.setAuthenticator(auth);  
17         myServer.setClientData(data);  
18  
19         myServer.setPort(4124);
```

```

20     myServer.setName("Echo Server v 1.0");
21
22     //config QAdminServer
23     myServer.setQAdminServerPort(4124);
24     myServer.getQAdminServer().getServer().setName("EchoAdmin v 1.0");
25     try {
26         myServer.startQAdminServer();
27         myServer.startServer();
28     } catch(AppException e){
29         System.out.println("Error in server : "+e);
30     }
31 }
32 }
33

```

So we have defined our QAdminServer to run on 4124 port.. but what will the Authentication used? If not explicit Authenticator QAdminServer will use `org.quickserver.net.qsadmin.Authenticator` has its Authenticator. If we check the Api docs we find that is a very simple, is username and password is hard coded has

```

Username : Admin
Password : QsAdmin

```

So, now lets compile the new program and run it, it should show the output has bellow indicating admin server is running on 4124 port.

```

Loading QuickServer v1.3 .. Done
-----
Name : Echo Server v 1.0
Port : 4123
-----

-----
Name : EchoAdmin v 1.0
Port : 4124
-----

```

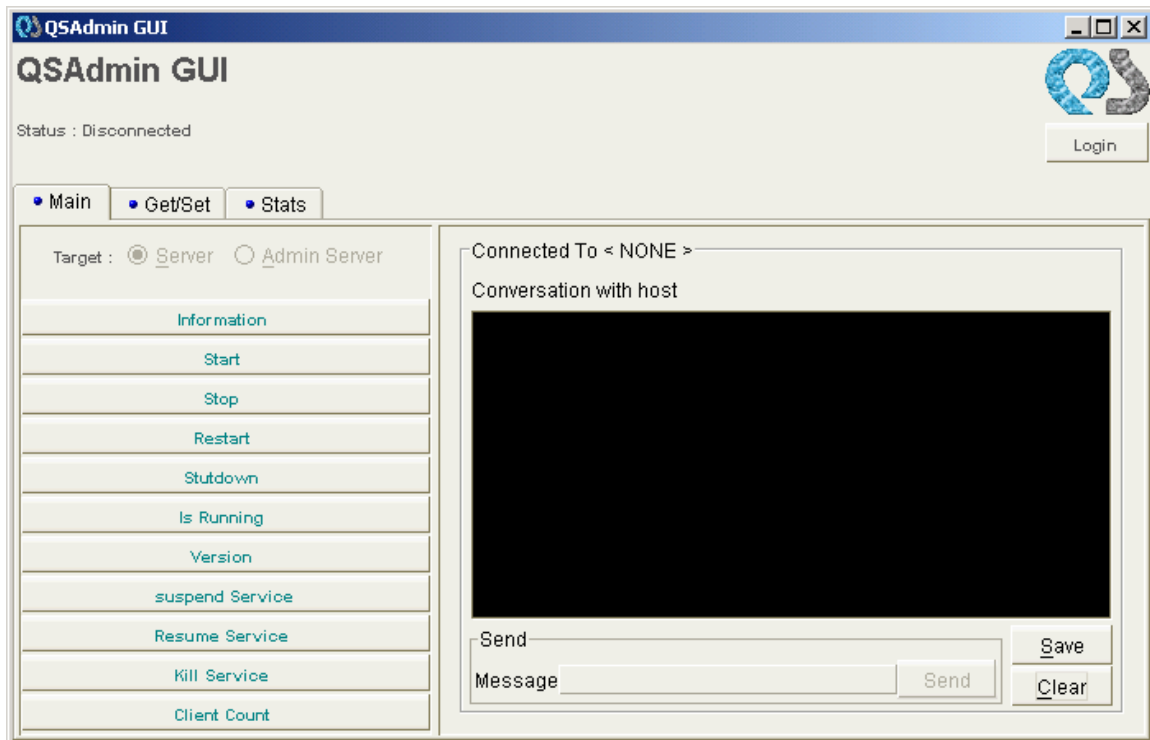
Start the QAdmin GUI using following command

```
java org.quickserver.net.server.QuickServer -admin
```

On Windows you can also do the following:-

Clicking on batch file “QsAdminGUI.bat” located under bin directory of QuickServer installation.

The GUI should be some thing like this



You can now connect to the Admin server of the EchoServer by clicking on “Login” button on the top right hand side. This will popup an login dialog box where you can input Ip Address and port were our server is running like 127.0.0.1 and 4124

It will also ask you the username and password, since we are using the default authenticator the default will work for us. Then click on login button.

Ones logged in the Status will show Authorised, now you can send it command by selecting the target server and clicking on the command buttons on the main tab.



So what’s target? Target is basically the server to which the command has to be sent, since both EchoServer(Server) and our EchoAdmin (AdminServer) are basically quickserver. The command buttons have nice descriptive tool tips that tell what the command will do.

On the right hand side there is a textbox that can be used to send any commands directly to the server, refer to the QsAdmin command handler:

`org.quickserver.net.qsadmin.CommandHandler` api docs.

To change server configuration you can use “Get/Set” tab. Just select the target and click on “Reload Properties For The Target” to load the configuration for the target server. Change the parameter you like and click on “Save” button to save it. Some parameter will need the server to be restarted for it to take effect (though this will not kill any connected client).

Just try a few options and explore the GUI.

## ***Adding your own command***

Say we want to change some property of the server when its running with out changing code or restarting server, we can do this by adding your own command to admin server.

To demonstrate this feature lets take hypothetically example. Say, we will display current interest rate offered when user send "*What's interest?*" This can be stored in a db and changed if needed with out restarting server, but lets say it was stored has a variable and we need to change it when the server was running.

Now lets implement the command handler to display the interest, bellow is the code.

Lets first change our echoserver and command handler and display interest.

```
01 package echoserver;
02
03 import org.quickserver.net.*;
04 import org.quickserver.net.server.*;
05
06 import java.io.*;
07
08 public class EchoServer {
09     public static void main(String s[]) {
10
11         String cmd = "echoserver.EchoCommandHandler";
12         String auth = "echoserver.EchoServerQuickAuthenticator";
13         String data = "echoserver.EchoServerPoolableData"; //Poolable
14
15         QuickServer myServer = new QuickServer(cmd);
16         myServer.setAuthenticator(auth);
17         myServer.setClientData(data);
18
19         myServer.setPort(4123);
20         myServer.setName("Echo Server v 1.0");
21
22         //store data needed to be changed by QsAdminServer
23         Object[] store = new Object[]{"12.00"};
24         myServer.setStoreObjects(store);
```

```

25
26 //config QSAdminServer
27 myServer.setQSAdminServerPort(4124);
28 myServer.getQSAdminServer().getServer().setName("EchoAdmin v 1.0");
29 try {
30     myServer.startQSAdminServer();
31     myServer.startServer();
32 } catch(AppException e){
33     System.out.println("Error in server : "+e);
34 }
35 }
36 }
37

```

```

01 // EchoCommandHandler.java
02 package echoserver;
03
04 import java.net.*;
05 import java.io.*;
06 import org.quickserver.net.server.ClientCommandHandler;
07 import org.quickserver.net.server.ClientHandler;
08
09 public class EchoCommandHandler implements ClientCommandHandler {
10
11     public void gotConnected(ClientHandler handler)
12         throws SocketTimeoutException, IOException {
13         handler.sendClientMsg("+++++++");
14         handler.sendClientMsg("| Welcome to EchoServer v 1.0 |");
15         handler.sendClientMsg("| Note: Password = Username |");
16         handler.sendClientMsg("| Send 'Quit' to exit |");
17         handler.sendClientMsg("+++++++");
18     }
19     public void lostConnection(ClientHandler handler)
20         throws IOException {
21         handler.sendSystemMsg("Connection lost : " +
22             handler.getSocket().getInetAddress());
23     }
24     public void closingConnection(ClientHandler handler)
25         throws IOException {
26         handler.sendSystemMsg("Closing connection : " +
27             handler.getSocket().getInetAddress());
28     }
29
30     public void handleCommand(ClientHandler handler, String command)
31         throws SocketTimeoutException, IOException {
32         if(command.equals("Quit")) {
33             handler.sendClientMsg("Bye ;-");
34             handler.closeConnection();
35             return;
36         }
37         if(command.equals("What's interest?")) {
38             handler.sendClientMsg("Interest is : "+
39                 (String)handler.getServer().getStoreObjects()[0]+
40                 "%").

```



```

41     } else if(command.equalsIgnoreCase("hello")) {
42         EchoServerData data = (EchoServerData) handler.getClientData();
43         data.setHelloCount(data.getHelloCount()+1);
44         if(data.getHelloCount()==1) {
45             handler.sendClientMsg("Hello "+data.getUsername());
46         } else {
47             handler.sendClientMsg("You told Hello "+data.getHelloCount()+
48                 " times. ");
49         }
50     } else {
51         handler.sendClientMsg("Echo : "+command);
52     }
53 }
54 }

```

Now lets define our CommandPlugin for QSAdminServer.

```

01 package echoserver;
02
03
04 import java.io.*;
05 import java.net.SocketTimeoutException;
06 import org.quickserver.net.server.*;
07 import org.quickserver.net.qsadmin.*;
08
09 public class QSAdminCommandPlugin implements CommandPlugin {
10     /**
11      * Echoserver commands
12      * -----
13      * set interest value
14      * get interest
15      */
16     public boolean handleCommand(ClientHandler handler, String command)
17         throws SocketTimeoutException, IOException {
18
19         QuickServer echoserver = (QuickServer)
20             handler.getServer().getStoreObjects()[0];
21         Object obj[] = echoserver.getStoreObjects();
22
23         if(command.toLowerCase().startsWith("set interest ")) {
24             String temp = "";
25             temp = command.substring("set interest ".length());
26             obj[0] = temp;
27             echoserver.setStoreObjects(obj);
28             handler.sendClientMsg("+OK interest changed");
29             return true;
30         } else if(command.toLowerCase().equals("get interest")) {
31             handler.sendClientMsg("+OK " + (String)obj[0]);
32             return true;
33         }
34         //ask QSAdminServer to process the command
35         return false;
36     }

```

In the above code we get the reference to the QuickServer of EchoServer from the store objects of QAdminServer [Ref. QuickServer API Docs for QAdminServer and CommandPlugin], using the code

```
QuickServer echoserver = (QuickServer)
    handler.getServer().getStoreObjects()[0];
```

Then we return true if processed the command got, else we return false indicating the QAdminServer's default command handler should be used to handle the command. The same technique can be used to override the default command of QAdminServer's command handler.

Now lets tell QuickServer to use this class has its QAdminServer's command plugin. Bellow is the code.

```
01 package echoserver;
02
03 import org.quickserver.net.*;
04 import org.quickserver.net.server.*;
05
06 import java.io.*;
07
08 public class EchoServer {
09     public static void main(String s[]) {
10
11         String cmd = "echoserver.EchoCommandHandler";
12         String auth = "echoserver.EchoServerQuickAuthenticator";
13         String data = "echoserver.EchoServerPoolableData"; //Poolable
14
15         QuickServer myServer = new QuickServer(cmd);
16         myServer.setAuthenticator(auth);
17         myServer.setClientData(data);
18
19         myServer.setPort(4123);
20         myServer.setName("Echo Server v 1.0");
21
22         //store data needed to be changed by QAdminServer
23         Object[] store = new Object[]{"12.00"};
24         myServer.setStoreObjects(store);
25
26         //config QAdminServer
27         myServer.setQAdminServerPort(4124);
28         myServer.getQAdminServer().getServer().setName("EchoAdmin v 1.0");
29         try {
30             //add command plugin
31             myServer.getQAdminServer().setCommandPlugin(
32                 "echoserver.QAdminCommandPlugin");
33             myServer.startQAdminServer();
```

```
34     myServer.startServer();
35     } catch(AppException e){
36         System.out.println("Error in server : "+e);
37     } catch(Exception e){
38         System.out.println("Error : "+e);
39     }
40 }
41 }
42
```

So, now lets compile the new program and run it. Now start a client like SocketTest and send the command "*What's interest?*" it should display "**Interest is : 12.00%**". Now lets start QAdminGUI has told earlier. And type the following command from Send message box and hit send.

```
get interest
```

This should now show the following output

```
+OK 12.00
```

Now lets change it, send the following command

```
set interest 15.00
```

Now again start a client like SocketTest and send the command "*What's interest?*" it should display "**Interest is : 15.00%**".

So try a few tricks and experiment and learn. That's it for QAdmin has of now, the next version of QuickServer will support the creation of GUI with in the QAdminGUI by passing it a XML file that define all the custom command implemented by your server.

## 7. Using Logging and customisation

Logging is an important tool for any project. Logging help us to understand what is happening inside our project, it also provides the auditing and debugging information. To learn more about how to use logging in general refer to the Sun web site <http://java.sun.com/j2se/1.4.0/docs/guide/util/logging/overview.html>

QuickServer currently only support Java Logging API (`java.util.logging`), logging was added to java in 1.4 version, so if you are using older version you can install an alternative implementation provided by Lumberjack library [<http://javalogging.sourceforge.net/>].

Lets go about making our sample project ‘EchoServer’ to do some logging. QuickServer by default has logging enabled. But it by default has no handler attached to it except `ConsoleHandler` and is set to `INFO` level. So whenever we close connection from an connected client to our EchoServer we can see some thing similar to this on console.

```
Feb 16, 2000 10:11:25 PM ClientHandler sendSystemMsg  
INFO: Closing connection : /127.0.0.1
```

This indicates that client which was connected from 127.0.0.1 closed the connection, this message was displayed using the `ClientHalder` class method `sendSystemMsg()` method.

Lets us see how we can control logging in our example project, to use java logging we must first import `java.util.logging` package into the class.

From the above logging, we just find that some client closed the connection or lost the connection, but how did this even come, we had not written any logging command. This happened because QuickServer uses logging internally for `sendSystemMsg()` method of `ClientHandler` class.

If you check the QuickServer documentation you find that `sendSystemMsg()` uses `INFO` by default if no level was specified. You can use `sendSystemMsg()` in `ClientCommandHandler` or `Authenticator` or any other class to log an event.

## Simple Logging

Now lets log every client's ip address when a client connects to our EchoServer, open and edit EchoCommandHandler.java and add the following line to gotConnected(ClientHandler handler) function.

```
handler.sendMessage("New Client : "+
    handler.getSocket().getInetAddress().getHostAddress(),
    Level.INFO);
```

You will have to import the following package to enable you to compile.

```
import java.util.logging.*;
```

Now compile and run your modified program, now try to connect you will now see that it will log your ip address when a client connects.

Now lets make QuickServer to send logs to a file (in XML format), bellow is the modified file.

```
01 package echoserver;
02
03 import org.quickserver.net.*;
04 import org.quickserver.net.server.*;
05
06 import java.io.*;
07 import java.util.logging.*;
08
09 public class EchoServer {
10     public static void main(String s[]) {
11
12         String cmd = "echoserver.EchoCommandHandler";
13         String auth = "echoserver.EchoServerQuickAuthenticator";
14         String data = "echoserver.EchoServerPoolableData"; //Poolable
15
16         QuickServer myServer = new QuickServer();
17
18         //setup logger to log to file
19         Logger logger = null;
20         FileHandler xmlLog = null;
21         File log = new File("./log/");
22         if(!log.canRead())
23             log.mkdir();
24         try {
25             logger = Logger.getLogger(""); //get root logger
26             logger.setLevel(Level.INFO);
27             xmlLog = new FileHandler("log/EchoServer.xml");
28             logger.addHandler(xmlLog);
29         } catch(IOException e){
30             System.err.println("Could not create xmlLog FileHandler : "+e);
31         }
32         //set logging level to fine
33         myServer.setConsoleLoggingLevel(Level.INFO);
```

```

34
35
36     myServer.setClientCommandHandler(cmd);
37     myServer.setAuthenticator(auth);
38     myServer.setClientData(data);
39
40     myServer.setPort(4123);
41     myServer.setName("Echo Server v 1.0");
42
43     //store data needed to be changed by QAdminServer
44     Object[] store = new Object[]{"12.00"};
45     myServer.setStoreObjects(store);
46
47     //config QAdminServer
48     myServer.setQAdminServerPort(4124);
49     myServer.getQAdminServer().getServer().setName("EchoAdmin v 1.0");
50     try {
51         //add command plugin
52         myServer.getQAdminServer().setCommandPlugin(
53             "echoserver.QAdminCommandPlugin");
54         myServer.startQAdminServer();
55         myServer.startServer();
56     } catch(AppException e){
57         System.out.println("Error in server : "+e);
58     } catch(Exception e){
59         System.out.println("Error : "+e);
60     }
61 }
62 }
63

```

In the above code, in line 21 and 22 we check if log directory is present in the location from where it was run from, if the directory can't be read, it will create it.

In line 25, we try to get the root logger and then in line 28, we add a new FileHandler, here we have not specified the formatter since default formatter is XMLFormatter.

In line 33, we set the console logging level to INFO.

Bellow is the modified code for EchoCommandHandler.java

```

01 // EchoCommandHandler.java
02 package echoserver;
03
04 import java.net.*;
05 import java.io.*;
06 import org.quickserver.net.server.ClientCommandHandler;
07 import org.quickserver.net.server.ClientHandler;
08 import java.util.logging.*;
09
10 public class EchoCommandHandler implements ClientCommandHandler {
11
12     public void gotConnected(ClientHandler handler)

```

```

13     throws SocketTimeoutException, IOException {
14     handler.sendSystemMsg("New Client : "+
15         handler.getSocket().getInetAddress().getHostAddress(),
16         Level.INFO);
17     handler.sendClientMsg("+++++++");
18     handler.sendClientMsg("| Welcome to EchoServer v 1.0 |");
19     handler.sendClientMsg("| Note: Password = Username |");
20     handler.sendClientMsg("| Send 'Quit' to exit |");
21     handler.sendClientMsg("+++++++");
22 }
23 public void lostConnection(ClientHandler handler)
24     throws IOException {
25     handler.sendSystemMsg("Connection lost : " +
26         handler.getSocket().getInetAddress());
27 }
28 public void closingConnection(ClientHandler handler)
29     throws IOException {
30     handler.sendSystemMsg("Closing connection : " +
31         handler.getSocket().getInetAddress());
32 }
33
34 public void handleCommand(ClientHandler handler, String command)
35     throws SocketTimeoutException, IOException {
36     if(command.equals("Quit")) {
37         handler.sendClientMsg("Bye ;-");
38         handler.closeConnection();
39         return;
40     }
41     if(command.equals("What's interest?")) {
42         handler.sendClientMsg("Interest is : "+
43             (String)handler.getServer().getStoreObjects()[0]+
44             "%");
45     } else if(command.equalsIgnoreCase("hello")) {
46         EchoServerData data = (EchoServerData) handler.getClientData();
47         data.setHelloCount(data.getHelloCount()+1);
48         if(data.getHelloCount()==1) {
49             handler.sendClientMsg("Hello "+data.getUsername());
50         } else {
51             handler.sendClientMsg("You told Hello "+data.getHelloCount()+
52                 " times. ");
53         }
54     } else {
55         handler.sendClientMsg("Echo : "+command);
56     }
57 }
58 }

```

Now compile and run your modified program, now try to connect you will now see that it will log your ip address on console and to the xml file.

Lets set the level of the logging to FINEST, using the QSAdminGUI and then observe the change in logging content, this will increase the text being logged. One can also change

logging level by modifying the code and letting level to the handler or you can also use QuickServer's method `setLoggingLevel()` to set logging level of all handler.

## Advanced Logging

When you set logging level to FINEST it loges a lot of information, one might want to separate logs of QuickServer and application. Bellow is a code that will allow you to do so.

```
01 package echoserver;
02
03 import org.quickserver.net.*;
04 import org.quickserver.net.server.*;
05
06 import java.io.*;
07 import java.util.logging.*;
08
09 public class EchoServer {
10     public static void main(String s[]) {
11
12         String cmd = "echoserver.EchoCommandHandler";
13         String auth = "echoserver.EchoServerQuickAuthenticator";
14         String data = "echoserver.EchoServerPoolableData"; //Poolable
15
16         QuickServer myServer = new QuickServer();
17
18         //setup logger to log to file
19         Logger logger = null;
20         FileHandler xmlLog = null;
21         FileHandler txtLog = null;
22         File log = new File("./log/");
23         if(!log.canRead())
24             log.mkdir();
25         try {
26             logger = Logger.getLogger("org.quickserver.net"); //get QS logger
27             logger.setLevel(Level.FINEST);
28             xmlLog = new FileHandler("log/EchoServer.xml");
29             logger.addHandler(xmlLog);
30
31             logger = Logger.getLogger("echoserver"); //get App logger
32             logger.setLevel(Level.FINEST);
33             txtLog = new FileHandler("log/EchoServer.txt");
34             txtLog.setFormatter(new SimpleFormatter());
35             logger.addHandler(txtLog);
36             myServer.setAppLogger(logger); //img : Sets logger to be used for app.
37         } catch (IOException e){
38             System.err.println("Could not create xmlLog FileHandler : "+e);
39         }
40         //set logging level to fine
41         myServer.setConsoleLoggingLevel(Level.INFO);
42
43     }
```



```

44     myServer.setClientCommandHandler(cmd);
45     myServer.setAuthenticator(auth);
46     myServer.setClientData(data);
47
48     myServer.setPort(4123);
49     myServer.setName("Echo Server v 1.0");
50
51     //store data needed to be changed by QAdminServer
52     Object[] store = new Object[]{"12.00"};
53     myServer.setStoreObjects(store);
54
55     //config QAdminServer
56     myServer.setQAdminServerPort(4124);
57     myServer.getQAdminServer().getServer().setName("EchoAdmin v 1.0");
58     try {
59         //add command plugin
60         myServer.getQAdminServer().setCommandPlugin(
61             "echoserver.QAdminCommandPlugin");
62         myServer.startQAdminServer();
63         myServer.startServer();
64     } catch (AppException e) {
65         System.out.println("Error in server : "+e);
66     } catch (Exception e) {
67         System.out.println("Error : "+e);
68     }
69 }
70 }
71

```

The comments in the code should be self explanatory, Now compile and run your modified program, now try to connect you will now see that it will log QuickServer internal logging to the xml file and application level logging to txt file has we wanted.

To open the xml file you will need the logging dtd, a copy of which can be found under log directory of **examples** folder.

### **Important:**

Always try to minimise the amount of logging on the console and for detailed logging use file handler and setting it level to a low value. This will improve the performance of your application. Avoid using System.out.println() and try to use logging instead. In ClientHandler there is a very useful method sendSystemMsg() that will logs and string passed to it .

## 8. XML Configuration

In the previous chapters as we have developed the EchoServer, we have been configuring the QuickServer aspects in the actual class. While this is an acceptable practice in some situations, many applications prefer to allow users to dynamically configure the application at start-up. To do this for the QuickServer portion, you can tell the server instance to read a XML file.

To configure QuickServer all you have to do is to first write the configuration file and then tell QuickServer object to load its configuration from it. Below is a very simple configuration file.

```
<quickserver>
  <name>EchoServer v 1.0</name>
  <client-command-handler>
    echoserver.EchoCommandHandler
  </client-command-handler>
</quickserver>
```

Now there are two ways to start your server,

- Using QuickServer startup parameter – load  
In this all you have to do is put QuickServer.jar and the all your classes to the classpath and start QuickServer by specifying the path of the xml configuration file using “-load” switch. Eg:  

```
java -jar QuickServer.jar -load myxmlconfig.xml
      or
java org.quickserver.net.server.QuickServer -load myxmlconfig.xml
      or
quickserver.bat -load myxmlconfig.xml
```
- Using QuickServer initService() method  
Some time you may need to start your application from your own jar file or class file or may want to add few code before server starts then this option is the best suited. Below is the code to initialise QuickServer from a xml configuration file.

```

QuickServer myServer = new QuickServer();
//pick the xml file form config folder
String configFile = "conf" + File.separator + "MyServer.xml";
Object config[] = new Object[] {configFile};
if(myServer.initService(config) != true) {
    System.err.println("Could't init server !!");
}

```

So now lets write the configuration file for our EchoServer that we wrote in previous chapter. Bellow is the XML file.

The XML configuration that is shipped with the examples is below



```

01 <quickserver>
02   <name>EchoServer v 1.0</name>
03   <port>4123</port>
04   <bind-address>127.0.0.1</bind-address>
05
06   <client-command-handler>
07     echoserver.EchoCommandHandler
08   </client-command-handler>
09   <authenticator>
10     echoserver.EchoServerQuickAuthenticator
11   </authenticator>
12   <client-data>
13     echoserver.EchoServerPoolableData
14   </client-data>
15
16   <console-logging-level>INFO</console-logging-level>
17
18   <!-- some extra config. added just to show -->
19   <timeout>4</timeout>
20   <timeout-msg>-ERR Timeout</timeout-msg>
21   <max-auth-try>5</max-auth-try>
22   <max-auth-try-msg>-ERR Max Auth Try Reached</max-auth-try-msg>
23   <max-connection>-1</max-connection>
24   <max-connection-msg>
25     Server Busy\nMax Connection Reached
26   </max-connection-msg>
27   <object-pool>
28     <max-active>-1</max-active>
29     <max-idle>15</max-idle>
30   </object-pool>
31   <!-- some extra config. added just to show -->
32
33   <qsadmin-server>
34     <name>EchoAdmin v 1.0</name>
35     <port>4124</port>
36     <bind-address>127.0.0.1</bind-address>
37     <command-plugin>

```

```

38     echoserver.QSAdminCommandPlugin
39     </command-plugin>
40 </qsadmin-server>
41
42 </quickserver>

```

### Listing 8 - 1

Bellow is the modified EchoServer.java file; it is now changed to load configuration from the xml file.

```

01 package echoserver;
02
03 import org.quickserver.net.*;
04 import org.quickserver.net.server.*;
05
06 import java.io.*;
07 import java.util.logging.*;
08
09 public class EchoServer {
10     public static void main(String s[]) {
11
12         QuickServer myServer = new QuickServer();
13
14         //setup logger to log to file
15         Logger logger = null;
16         FileHandler xmlLog = null;
17         FileHandler txtLog = null;
18         File log = new File("./log/");
19         if(!log.canRead())
20             log.mkdir();
21         try {
22             logger = Logger.getLogger("org.quickserver.net"); //get qs logger
23             logger.setLevel(Level.FINEST);
24             xmlLog = new FileHandler("log/EchoServer.xml");
25             logger.addHandler(xmlLog);
26
27             logger = Logger.getLogger("echoserver"); //get app logger
28             logger.setLevel(Level.FINEST);
29             txtLog = new FileHandler("log/EchoServer.txt");
30             txtLog.setFormatter(new SimpleFormatter());
31             logger.addHandler(txtLog);
32             //img : Sets logger to be used for app.
33             myServer.setAppLogger(logger);
34         } catch(IOException e){
35             System.err.println("Could not create xmlLog FileHandler : "+e);
36         }
37
38         //store data needed to be changed by QSAdminServer
39         Object[] store = new Object[]{"12.00"};
40         myServer.setStoreObjects(store);
41
42         //load QuickServer from xml

```

```
43     String confFile = "config"+File.separator+"EchoServer.xml";
44     Object config[] = new Object[] {confFile};
45     if(myServer.initService(config) == true) {
46         try {
47             myServer.startQSAdminServer();
48             myServer.startServer();
49         } catch(AppException e){
50             System.out.println("Error in server : "+e);
51         } catch(Exception e){
52             System.out.println("Error : "+e);
53         }
54     }
55 }
56 }
57 }
```

### Listing 8 - 2

To learn more about how to use XML configuration refer to QuickServer Java Documentation (main page has a sample xml) and to the example that come with QuickServer.

## 9. Data Modes and Data Types

Until now we have been communicating only using string that end with `<CR><LF>`. Well this will work most Internet standard protocols. But some time we may have to be able to receive stream of character bytes or receive java objects.

This is, were the `DataMode` and `DataType` classes comes in handy. Using this you can tell `ClientHandler` which mode of communication to be used.

`DataMode` class is used to define format of data exchange between `QuickServer` and client socket. There are currently three supported modes

- **DataMode.STRING** – This is the default mode of exchange. In this mode you can receive data as `String` (characters terminated by `<CR><LF>`).
- **DataMode.OBJECT** – In this mode you can receive java objects that are `Serializable`. This mode is only useful if the client is also written in java.
- **DataMode.BYTE** – In this mode you can receive all the bytes sent by the client including `<CR>` or `<LF>` or any other control characters. This mode is very useful when dealing with hardware based client or non-standard protocol like XML over http or when defining your own protocol.

`DataType` class is used to define the type (direction) of data exchanging between `QuickServer` and client socket. There are currently two types

- **DataType.IN** – This is used to define incoming data into `QuickServer`.
- **DataType.OUT** – This is used to define outgoing data from `QuickServer`.

The data mode of any data type can be set to a `ClientHandler` object using the `setDataMode()` function. The format is given bellow

```
setDataMode(DataMode dataMode, DataType dataType)
```

☛ **Note:** When mode is `DataMode.OBJECT` and type is `DataType.IN` this call will block until the client `ObjectOutputStream` has written and flushes the header.

In next chapters lets try to explore the other 2 modes of data exchange.

## 10. Communication with OBJECT Mode

TODO.

## 11. Communication with BYTE Mode

TODO.



## 12. XML Based JDBC Mapping

TODO.