Chapter 7

Sockets for Servers

Contents

- Using ServerSockets
  - Serving Binary data
  - Multithreaded Servers
  - Closing Server Sockets
- Logging: What/How to Log
- Constructing Server Sockets: Constructing Without Binding
- Getting Information about Server Socket
- Socket Options
  - SO_OPTION
  - SO_REUSEADDR
  - SO_RCVBUF
  - Class of Service
- HTTP Servers
  - A single file server
  - A re-director
  - A Fully-Fledged HTTP Server

***ref*:**

- *Java Network Programming by Elliotte Rusty Harold*
- *Prefer to class labs for better insights of the programs*

Using ServerSocket

- The `ServerSocket` class contains everything needed to write servers in Java.
- It has constructors that create new ServerSocket objects, methods that listen for connections on a specified port, methods that configure the various server socket options, and the usual miscellaneous methods such as toString()
- In Java, the **basic life cycle of a server program** is this:
  - A new ServerSocket is created on a particular port using a ServerSocket() constructor.
  - The ServerSocket listens for incoming connection attempts on that port using its accept() method.
    - accept() blocks until a client attempts to make a connection, at which point accept() returns a Socket object connecting the client and the server.
  - Depending on the type of server, either the Socket's getInputStream() method,
  - getOutputStream() method, or both are called to get input and output streams that communicate with the client.
  - The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
  - The server, the client, or both close the connection.
  - The server returns to step 2 and waits for the next connection.

Example: Simple Daytime Server

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class DaytimeServer {
public final static int PORT = 13;
public static void main(String[] args) {
try (ServerSocket server = new ServerSocket(PORT)) {
while (true) {
try (Socket connection = server.accept()) {
Writer out = new OutputStreamWriter(connection.getOutputStream());
Date now = new Date();
out.write(now.toString() +"\r\n");
out.flush();
connection.close();
} catch (IOException ex) {}
}
} catch (IOException ex) {
System.err.println(ex);
}
}
}
```

Multithreaded Server

- If The server sends a few dozen bytes at most and then closes the connection. It’s plausible here to process each connection fully before moving on to the next one.
- The solution is to give each connection its own thread, separate from the thread that accepts incoming connections into the queue

Example: Multi-Threaded Daytime server

```
import java.net.*;
import java.io.*;
import java.util.Date;
public class MultithreadedDaytimeServer {
public final static int PORT = 13;
public static void main(String[] args) {
try (ServerSocket server = new ServerSocket(PORT)) {
while (true) {
try {
Socket connection = server.accept();
Thread task = new DaytimeThread(connection);
task.start();
} catch (IOException ex) {}
}
} catch (IOException ex) {
System.err.println("Couldn't start server");
}
}
private static class DaytimeThread extends Thread {
private Socket connection;
//Constructor
DaytimeThread(Socket connection) {
this.connection = connection;
}
@Override
public void run() {
try {
Writer out = new OutputStreamWriter(connection.getOutputStream());
Date now = new Date();
out.write(now.toString() +"\r\n");
out.flush();
} catch (IOException ex) {
System.err.println(ex);
} finally {
try {
connection.close();
} catch (IOException e) {
// ignore;
}
}
}
}
}
```

- Here, a new thread for each connection, numerous roughly simultaneous incoming connections can cause it to spawn an indefinite number of threads.
- Eventually, the Java virtual machine will run out of memory and crash.
- A better approach is to use a fixed thread pool to limit the potential resource usage

Example: A multi-threaded daytime server with thread pool

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.*;
public class PooledDaytimeServer {

public final static int PORT = 13;
public static void main(String[] args) {
ExecutorService pool = Executors.newFixedThreadPool(50);
try (ServerSocket server = new ServerSocket(PORT)) {
while (true) {
try {
Socket connection = server.accept();
Callable<Void> task = new DaytimeTask(connection);
pool.submit(task);
} catch (IOException ex) {}
}
} catch (IOException ex) {
System.err.println("Couldn't start server");
}
}

private static class DaytimeTask implements Callable<Void> {
private Socket connection;
DaytimeTask(Socket connection) {
this.connection = connection;
}
@Override
public Void call() {
try {
Writer out = new OutputStreamWriter(connection.getOutputStream());
Date now = new Date();
out.write(now.toString() +"\r\n");
out.flush();
} catch (IOException ex) {
System.err.println(ex);
} finally {
try {
connection.close();
} catch (IOException e) {
// ignore;
}
}
return null;
}
}
}
```

Writing to Servers with Sockets

- In this section we will see how client can send message to server and server can respond to it, making it an echo server

  Example:

```
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
import java.util.*;
import java.io.IOException

public class EchoServer {
public static int DEFAULT_PORT = 7;
public static void main(String[] args) {
int port;
try {
port = Integer.parseInt(args[0]);
} catch (RuntimeException ex) {
port = DEFAULT_PORT;
}
System.out.println("Listening for connections on port " +
port);
ServerSocketChannel serverChannel;
Selector selector;
try {
serverChannel = ServerSocketChannel.open();
ServerSocket ss = serverChannel.socket();
InetSocketAddress address = new InetSocketAddress(port);
ss.bind(address);
serverChannel.configureBlocking(false);
selector = Selector.open();
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
} catch (IOException ex) {
ex.printStackTrace();
return;
}

while (true) {
try {
selector.select();
} catch (IOException ex) {
ex.printStackTrace();
break;
}
Set<SelectionKey> readyKeys = selector.selectedKeys();
Iterator<SelectionKey> iterator = readyKeys.iterator();
while (iterator.hasNext()) {
SelectionKey key = iterator.next();
iterator.remove();
try {
if (key.isAcceptable()) {
ServerSocketChannel server = (ServerSocketChannel)
key.channel();
SocketChannel client = server.accept();
System.out.println("Accepted connection from " + client);
client.configureBlocking(false);
SelectionKey clientKey = client.register(
selector, SelectionKey.OP_WRITE | SelectionKey.OP_READ);
ByteBuffer buffer = ByteBuffer.allocate(100);
clientKey.attach(buffer);

}
if (key.isReadable()) {
SocketChannel client = (SocketChannel) key.channel();
ByteBuffer output = (ByteBuffer) key.attachment();
client.read(output);
}
if (key.isWritable()) {
SocketChannel client = (SocketChannel) key.channel();
ByteBuffer output = (ByteBuffer) key.attachment();
output.flip();
client.write(output);
output.compact();
}
} catch (IOException ex) {
key.cancel();
try {
key.channel().close();
} catch (IOException cex) {}
}
}
}
}
}
```

Closing Server Sockets

- If you're finished with a server socket, you should close it,
- especially if the program is going to continue to run for some time. This frees up the port for other programs that may wish to use it.
- Closing a ServerSocket frees a port on the local host, allowing another server to bind to the port; it also breaks all currently open sockets that the ServerSocket has accepted.

```
ServerSocket server = new ServerSocket();

try {

SocketAddress address = new InetSocketAddress(port);

server.bind(address);

// ... work with the server socket

} finally {

try {
```

```
server.close();
} catch (IOException ex) {
// ignore
}
}
```

- The isClosed() method returns true if the ServerSocket has been closed, false if it hasn't.
- isBound() returns true if the ServerSocket has ever been bound to a port, even if it's currently closed.
- If you need to test whether a ServerSocket is open, you must check both that isBound() returns true and that isClosed() returns false

```
public static boolean isOpen(ServerSocket ss) {
return ss.isBound() && !ss.isClosed();
}
```

Logging

- Servers run unattended for long periods of time. It's often important to debug what happened when in a server long after the fact. For this reason, it's advisable to store server logs for at least some period of time.

What to Log

There are two primary things you want to store in your logs:

1. Requests
2. Server errors

- The error log is exclusively for unexpected exceptions
- The error log contains mostly unexpected exceptions that occurred while the server was running.
- For instance, any NullPointerException that happens should be logged here because it indicates a bug in the server you'll need to fix.
- The error log does not contain client errors, such as a client that unexpectedly disconnects or sends a malformed request. These go into the request log

How to Log

- The easies way is to just create one per class like

  ```
  private final static Logger auditLogger =
  Logger.getLogger("requests")
  ```

- This example outputs to a log named “requests.”
- Multiple Logger objects can output to the same log, but each logger always logs to exactly one log. What and where the log is, depends on external configuration.
- Most commonly it’s a file, which may or may not be named “requests”
- Once you have a logger, you can write to it using any of several methods. The most basic is log().
- For example, this catch block logs an unexpected runtime exception at the highest level:

  ```
  catch (RuntimeException ex) {

  logger.log(Level.SEVERE, "unexpected error " +
  ex.getMessage(), ex);

  }
  ```

- There are seven levels defined as named constants in java.util.logging.Level in
- descending order of seriousness:
  - Level.SEVERE (highest value)
  - Level.WARNING
  - Level.INFO
  - Level.CONFIG
  - Level.FINE
  - Level.FINER
  - Level.FINEST (lowest value)
- Generally, each record should contain a timestamp, the client address, and any information specific to the request that was being processed.

Q. A daytime server with logger

```
import java.io.*;

import java.net.*;

import java.util.Date;

import java.util.concurrent.*;
```

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

public class LoggingDaytimeServer {
public final static int PORT = 13;
private final static Logger auditLogger =
Logger.getLogger("requests");
private final static Logger errorLogger =
Logger.getLogger("errors");
public static void main(String[] args) {
ExecutorService pool = Executors.newFixedThreadPool(50);
try (ServerSocket server = new ServerSocket(PORT)) {
while (true) {
try {
Socket connection = server.accept();
Callable<Void> task = new DaytimeTask(connection);
pool.submit(task);
} catch (IOException ex) {
errorLogger.log(Level.SEVERE, "accept error", ex);
} catch (RuntimeException ex) {
errorLogger.log(Level.SEVERE, "unexpected error " +
ex.getMessage(), ex);
}
}
} catch (IOException ex) {
errorLogger.log(Level.SEVERE, "Couldn't start server", ex);
} catch (RuntimeException ex) {
errorLogger.log(Level.SEVERE, "Couldn't start server: " +
ex.getMessage(), ex);
}
}
private static class DaytimeTask implements Callable<Void> {
private Socket connection;
DaytimeTask(Socket connection) {
this.connection = connection;
}
@Override
public Void call() {
try {
Date now = new Date();
// write the log entry first in case the client disconnects
auditLogger.info(now + " " + connection.getRemoteSocketAddress());
Writer out = new OutputStreamWriter(connection.getOutputStream());
out.write(now.toString() +"\r\n");
out.flush();
} catch (IOException ex) {
// client disconnected; ignore;
} finally {
try {
connection.close();
} catch (IOException ex) {
// ignore;
}
}
return null;
}
}
}
```

Properties of logger file

By default, the logs are just output to the console. For example, here's the output from the preceding server when I connected to it a few times in quick succession:

```
Apr 13, 2013 8:54:50 AM LoggingDaytimeServer$DaytimeTask call
INFO: Sat Apr 13 08:54:50 EDT 2013 /0:0:0:0:0:0:0:1:56665
Apr 13, 2013 8:55:08 AM LoggingDaytimeServer$DaytimeTask call
INFO: Sat Apr 13 08:55:08 EDT 2013 /0:0:0:0:0:0:0:1:56666
Apr 13, 2013 8:55:16 AM LoggingDaytimeServer$DaytimeTask call
INFO: Sat Apr 13 08:55:16 EDT 2013 /0:0:0:0:0:0:0:1:56667
```

- You'll want to configure the runtime environment such that logs go to a more permanent destination
- The java.util.logging.config.file system property points to a file in the normal properties format that controls the logging

S**ample logging properties file that specifies:**

- Logs should be written to a file.
- The requests log should be in /var/logs/daytime/requests.log at level Info.
- The errors log should be in /var/logs/daytime/requests.log at level Severe.
- Limit the log size to about 10 megabytes, then rotate.
- Keep two logs: the current one and the previous one.
- Use the basic text formatter (not XML).
- Each line of the logfile should be in the form level message timestamp.
- Example: A logging properties file

```
handlers=java.util.logging.FileHandler

java.util.logging.FileHandler.pattern =
/var/logs/daytime/requests.log

java.util.logging.FileHandler.limit = 10000000

java.util.logging.FileHandler.count = 2

java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter

java.util.logging.FileHandler.append = true

java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n


requests.level = INFO

audit.level = SEVERE
```

- Here's some typical log output

```
SEVERE: Couldn't start server [Sat Apr 13 10:07:01 EDT 2013]

INFO: Sat Apr 13 10:08:05 EDT 2013 /0:0:0:0:0:0:0:1:57275

[Sat Apr 13 10:08:05 EDT 2013]

INFO: Sat Apr 13 10:08:06 EDT 2013 /0:0:0:0:0:0:0:1:57276

[Sat Apr 13 10:08:06 EDT 2013]
```

Constructing Server Sockets

There are four public ServerSocket constructors:

1. public ServerSocket(int port) throws **BindException, IOException**
2. public ServerSocket(int port, int queueLength) throws **BindException, IOException**
   - **queueLength indicates max no of unexpected connections**
3. public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws **IOException**

4. public ServerSocket() throws **IOException**

Constructing Without Binding

The noargs constructor creates a ServerSocket object but does not actually bind it to a port, so it cannot initially accept any connections. It can be bound later using the bind() methods

1. public void bind(SocketAddress endpoint) throws IOException
2. public void bind(SocketAddress endpoint, int queueLength) throws IOException

Example:

```
ServerSocket ss = new ServerSocket();
// set socket options...
SocketAddress http = new InetSocketAddress(80);
ss.bind(http);
```

Getting Information About a Server Socket

The ServerSocket class provides two getter methods that tell you the local address and port occupied by the server socket. These are useful if you've opened a server socket on an anonymous port and/or an unspecified network interface

1. public InetAddress getInetAddress()
2. public int getLocalPort()

Q. Bind server to a random port

```
public class RandomPort {
 public static void main(String[] args) {
 try {
// when 0 is passed a random port is picked by java
 ServerSocket server = new ServerSocket(0);
 System.out.println("This server runs on port "
 + server.getLocalPort());
 } catch (IOException ex) {
 System.err.println(ex);
 }
```

```
  }
}
```

If the ServerSocket has not yet bound to a port, getLocalPort() returns -1.

Socket Options

Socket options specify how the native sockets on which the ServerSocket class relies

send and receive data. For server sockets, Java supports three options:

• SO_TIMEOUT

• SO_REUSEADDR

• SO_RCVBUF

1. SO_TIMEOUT

- SO_TIMEOUT is the amount of time, in milliseconds, that accept() waits for an incoming connection before throwing a **java.io.InterruptedIOException**
- If SO_TIMEOUT is 0, accept() will never time out.
- The default is to never time out
- Java provides, getter and setter for SO_TIMEOUT
  - ⇨ **public void setSoTimeout(int timeout) throws SocketException**
  - ⇨ **public int getSoTimeout() throws IOException**
- **Example:**

  SET TIMEOUT:

```
try (ServerSocket server = new ServerSocket(port)) {
server.setSoTimeout(30000); // block for no more than 30
seconds
try {
Socket s = server.accept();
// handle the connection
// ...
} catch (SocketTimeoutException ex) {
System.err.println("No connection within 30 seconds");
}
} catch (IOException ex) {
 System.err.println("Unexpected IOException: " + e);
}
```

- **Example: GET TIMEOUT**

```
public void printSoTimeout(ServerSocket server) {
int timeout = server.getSoTimeOut();
 if (timeout > 0) {
 System.out.println(server + " will time out after "
 + timeout + "milliseconds.");
 } else if (timeout == 0) {
 System.out.println(server + " will never time out.");
 } else {
 System.out.println("Impossible condition occurred in " + server);
 System.out.println("Timeout cannot be less than zero." );
 }
}
```

2. SO_REUSEADDR

- The SO_REUSEADDR option for server sockets is very similar to the same option for client sockets
- It determines whether a new socket will be allowed to bind to a previously used port while there might still be data traversing the network addressed to the old socket
- there are two methods to get and set this option:
  - **public boolean getReuseAddress() throws SocketException**
  - **public void setReuseAddress(boolean on) throws SocketException**
- **Example:**

```
ServerSocket ss = new ServerSocket(10240);
System.out.println("Reusable: " + ss.getReuseAddress());
```

- The default value is platform dependent.

3. SO_RCVBUF

- The SO_RCVBUF option sets the default receive buffer size for client sockets accepted by the server socket.
- It's read and written by these two methods:
  - **public int getReceiveBufferSize() throws SocketException**
  - **public void setReceiveBufferSize(int size) throws SocketException**
- Setting SO_RCVBUF on a server socket is like calling setReceiveBufferSize() on each individual socket returned by accept()

Class of Service

As you learned in the previous chapter, different types of Internet services have different performance needs. For instance, live streaming video of sports needs relatively high bandwidth. On the other hand, a movie might still need high bandwidth but be able to tolerate more delay and latency. Email can be passed over low-bandwidth connections and even held up for several hours without major harm

Four general traffic classes are defined for TCP:

• Low cost

• High reliability

• Maximum throughput

• Minimum delay

The setPerformancePreferences() method expresses the relative preferences given to connection time, latency, and bandwidth for sockets accepted on this server:

```
    public void setPerformancePreferences(int connectionTime, int
latency, int bandwidth)
```

For instance, by setting connectionTime to 2, latency to 1, and bandwidth to 3, you indicate that maximum bandwidth is the most important characteristic, minimum latency is the least important, and connection time is in the middle:

ss.setPerformancePreferences(2, 1, 3);

HTTP Servers

A Single-File Server

- A Single-File HTTP Server is a type of server that always sends out the samefile, no matter what the request is.
- Example: An HTTP server that serves a single file

```
import java.io.*;
import java.net.*;
import java.nio.charset.Charset;
import java.nio.file.*;
import java.util.concurrent.*;
import java.util.logging.*;
public class SingleFileHTTPServer {
 private static final Logger logger =
Logger.getLogger("SingleFileHTTPServer");
 private final byte[] content;
 private final byte[] header;
 private final int port;
 private final String encoding;
 public SingleFileHTTPServer(String data, String encoding,
 String mimeType, int port) throws
UnsupportedEncodingException {
 this(data.getBytes(encoding), encoding, mimeType, port);
 }
 public SingleFileHTTPServer(
 byte[] data, String encoding, String mimeType, int port) {
 this.content = data;
 this.port = port;
 this.encoding = encoding;
 String header = "HTTP/1.0 200 OK\r\n"
 + "Server: OneFile 2.0\r\n"
 + "Content-length: " + this.content.length + "\r\n"
 + "Content-type: " + mimeType + "; charset=" + encoding +
"r\n\r\n";
 this.header = header.getBytes(Charset.forName("US-ASCII"));
 }
 public void start() {
 ExecutorService pool = Executors.newFixedThreadPool(100);
 try (ServerSocket server = new ServerSocket(this.port)) {
 logger.info("Accepting connections on port " +
server.getLocalPort());
 logger.info("Data to be sent:");
 logger.info(new String(this.content, encoding));
 while (true) {
 try {
 Socket connection = server.accept();
 pool.submit(new HTTPHandler(connection));
 } catch (IOException ex) {
 logger.log(Level.WARNING, "Exception accepting
connection",ex);
 } catch (RuntimeException ex) {
 logger.log(Level.SEVERE, "Unexpected error", ex);
 }
 }
 } catch (IOException ex) {
logger.log(Level.SEVERE, "Could not start server", ex);
 }
 }
 private class HTTPHandler implements Callable<Void> {
 private final Socket connection;
 HTTPHandler(Socket connection) {
 this.connection = connection;
 }
 @Override
 public Void call() throws IOException {
 try {
 OutputStream out = new BufferedOutputStream(
 connection.getOutputStream()
 );
 InputStream in = new BufferedInputStream(
 connection.getInputStream()
 );
 // read the first line only; that's all we need
 StringBuilder request = new StringBuilder(80);
 while (true) {
 int c = in.read();
 if (c == '\r' || c == '\n' || c == -1) break;
 request.append((char) c);
 }
 // If this is HTTP/1.0 or later send a MIME header
 if (request.toString().indexOf("HTTP/") != -1) {
 out.write(header);
 }
 out.write(content);
 out.flush();
 } catch (IOException ex) {
 logger.log(Level.WARNING, "Error writing to client", ex);
 } finally {
 connection.close();
 }
 return null;
 }
 }
public static void main(String[] args) {
 // set the port to listen on
 int port;
 try {
 port = Integer.parseInt(args[1]);
 if (port < 1 || port > 65535) port = 80;
 } catch (RuntimeException ex) {
 port = 80;
 }
String encoding = "UTF-8";
 if (args.length > 2) encoding = args[2];
 try {
 Path path = Paths.get(args[0]);;
 byte[] data = Files.readAllBytes(path);
 String contentType =
URLConnection.getFileNameMap().getContentTypeFor(args[0]);
 SingleFileHTTPServer server = new SingleFileHTTPServer(data,
encoding,
 contentType, port);
 server.start();
 } catch (ArrayIndexOutOfBoundsException ex) {
System.out.println(
 "Usage: java SingleFileHTTPServer filename port encoding");
 } catch (IOException ex) {
 logger.severe(ex.getMessage());
 }
 }
}
```

A Redirector

Redirection is another simple but useful application for a special-purpose HTTP server.

In this section, you develop a server that redirects users from one website to another—

for example, from cnet.com to www.cnet.com.

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.logging.*;
public class Redirector {
 private static final Logger logger = Logger.getLogger("Redirector");
 private final int port;
 private final String newSite;
 public Redirector(String newSite, int port) {
 this.port = port;
 this.newSite = newSite;
 }
 public void start() {
 try (ServerSocket server = new ServerSocket(port)) {
 logger.info("Redirecting connections on port "
 + server.getLocalPort() + " to " + newSite);
 while (true) {
 try {
 Socket s = server.accept();
 Thread t = new RedirectThread(s);
 t.start();
 } catch (IOException ex) {
 logger.warning("Exception accepting connection");
 } catch (RuntimeException ex) {
 logger.log(Level.SEVERE, "Unexpected error", ex);
 }
 }
 } catch (BindException ex) {
 logger.log(Level.SEVERE, "Could not start server.", ex);
 } catch (IOException ex) {
 logger.log(Level.SEVERE, "Error opening server socket", ex);
 }
 }
private class RedirectThread extends Thread {
 private final Socket connection;
 RedirectThread(Socket s) {
 this.connection = s;
 }
 public void run() {
 try {
 Writer out = new BufferedWriter(
 new OutputStreamWriter(
 connection.getOutputStream(), "US-ASCII"
 )
 );
 Reader in = new InputStreamReader(
 new BufferedInputStream(
 connection.getInputStream()
)
 );
 // read the first line only; that's all we need
 StringBuilder request = new StringBuilder(80);
 while (true) {
 int c = in.read();
 if (c == '\r' || c == '\n' || c == -1) break;
 request.append((char) c);
 }
 String get = request.toString();
 String[] pieces = get.split("\\w*");
 String theFile = pieces[1];
 // If this is HTTP/1.0 or later send a MIME header
 if (get.indexOf("HTTP") != -1) {
 out.write("HTTP/1.0 302 FOUND\r\n");
 Date now = new Date();
 out.write("Date: " + now + "\r\n");
 out.write("Server: Redirector 1.1\r\n");
 out.write("Location: " + newSite + theFile + "\r\n");
 out.write("Content-type: text/html\r\n\r\n");
 out.flush();
 }
// Not all browsers support redirection so we need to
 // produce HTML that says where the document has moved to.
 out.write("<HTML><HEAD><TITLE>Document moved</TITLE></HEAD>\r\n");
 out.write("<BODY><H1>Document moved</H1>\r\n");
 out.write("The document " + theFile
 + " has moved to\r\n<A HREF=\"" + newSite + theFile + "\">"
 + newSite + theFile
 + "</A>.\r\n Please update your bookmarks<P>");
 out.write("</BODY></HTML>\r\n");
 out.flush();
 logger.log(Level.INFO,
 "Redirected " + connection.getRemoteSocketAddress());
 } catch(IOException ex) {
 logger.log(Level.WARNING,
 "Error talking to " + connection.getRemoteSocketAddress(), ex);
 } finally {
 try {
 connection.close();
 } catch (IOException ex) {}
 }
 }
 }
 public static void main(String[] args) {
 int thePort;
 String theSite;
try {
 theSite = args[0];
 // trim trailing slash
 if (theSite.endsWith("/")) {
 theSite = theSite.substring(0, theSite.length() - 1);
 }
 } catch (RuntimeException ex) {
 System.out.println(
 "Usage: java Redirector http://www.newsite.com/ port");
 return;
 }
 try {
 thePort = Integer.parseInt(args[1]);
 } catch (RuntimeException ex) {
 thePort = 80;
 }
 Redirector redirector = new Redirector(theSite, thePort);
 redirector.start();
 }
}
```

A Full-Fledged HTTP Server

- a full-blown HTTPserver, called JHTTP, that can serve an entire document tree, including images, applets, HTML files, text files, and more
- It will be very similar to the SingleFileHTTPServer, except that it pays attention to the GET requests
- Rather than processing each request as it arrives in the main thread of execution,
- you'll place incoming connections in a pool. Separate instances of a RequestProcessor class will remove the connections from the pool and process them

```
            import java.io.*;
import java.net.*;
import java.util.concurrent.*;
import java.util.logging.*;
public class JHTTP {
 private static final Logger logger = Logger.getLogger(
 JHTTP.class.getCanonicalName());
private static final int NUM_THREADS = 50;
 private static final String INDEX_FILE = "index.html";
 private final File rootDirectory;
 private final int port;
 public JHTTP(File rootDirectory, int port) throws IOException {
 if (!rootDirectory.isDirectory()) {
 throw new IOException(rootDirectory
 + " does not exist as a directory");
 }
 this.rootDirectory = rootDirectory;
 this.port = port;
 }
 public void start() throws IOException {
 ExecutorService pool = Executors.newFixedThreadPool(NUM_THREADS);
 try (ServerSocket server = new ServerSocket(port)) {
 logger.info("Accepting connections on port " + server.getLocalPort());
 logger.info("Document Root: " + rootDirectory);
 while (true) {
 try {
 Socket request = server.accept();
 Runnable r = new RequestProcessor(
 rootDirectory, INDEX_FILE, request);
 pool.submit(r);
 } catch (IOException ex) {
 logger.log(Level.WARNING, "Error accepting connection", ex);
 }
 }
 }
 }
public static void main(String[] args) {
 // get the Document root
 File docroot;
 try {
 docroot = new File(args[0]);
 } catch (ArrayIndexOutOfBoundsException ex) {
 System.out.println("Usage: java JHTTP docroot port");
 return;
 }
 // set the port to listen on
 int port;
 try {
 port = Integer.parseInt(args[1]);
 if (port < 0 || port > 65535) port = 80;
 } catch (RuntimeException ex) {
```

```
public class RequestProcessor implements Runnable {
 private final static Logger logger = Logger.getLogger(
 RequestProcessor.class.getCanonicalName());
 private File rootDirectory;
 private String indexFileName = "index.html";
 private Socket connection;
 public RequestProcessor(File rootDirectory,
 String indexFileName, Socket connection) {
 if (rootDirectory.isFile()) {
 throw new IllegalArgumentException(
 "rootDirectory must be a directory, not a file");
 }
 try {
 rootDirectory = rootDirectory.getCanonicalFile();
 } catch (IOException ex) {
 }
 this.rootDirectory = rootDirectory;
 if (indexFileName != null) this.indexFileName = indexFileName;
 this.connection = connection;
 }
 @Override
 public void run() {
 // for security checks
 String root = rootDirectory.getPath();
 try {
 OutputStream raw = new BufferedOutputStream(
 connection.getOutputStream()
 );
 Writer out = new OutputStreamWriter(raw);
 Reader in = new InputStreamReader(
 new BufferedInputStream(
 connection.getInputStream()
 ),"US-ASCII"
 );
 StringBuilder requestLine = new StringBuilder();
 while (true) {
 int c = in.read();
 if (c == '\r' || c == '\n') break;
 requestLine.append((char) c);
 }
 String get = requestLine.toString();
 logger.info(connection.getRemoteSocketAddress() + " " + get);
String[] tokens = get.split("\\s+");
 String method = tokens[0];
 String version = "";
 if (method.equals("GET")) {
 String fileName = tokens[1];
 if (fileName.endsWith("/")) fileName += indexFileName;
 String contentType =
 URLConnection.getFileNameMap().getContentTypeFor(fileName);
 if (tokens.length > 2) {
 version = tokens[2];
 }
 File theFile = new File(rootDirectory,
 fileName.substring(1, fileName.length()));
 if (theFile.canRead()
 // Don't let clients outside the document root
 && theFile.getCanonicalPath().startsWith(root)) {
 byte[] theData = Files.readAllBytes(theFile.toPath());
 if (version.startsWith("HTTP/")) { // send a MIME header
 sendHeader(out, "HTTP/1.0 200 OK", contentType, theData.length);
 }
 // send the file; it may be an image or other binary data
 // so use the underlying output stream
 // instead of the writer
 raw.write(theData);
 raw.flush();
 } else { // can't find the file
 String body = new StringBuilder("<HTML>\r\n")
 .append("<HEAD><TITLE>File Not Found</TITLE>\r\n")
 .append("</HEAD>\r\n")
 .append("<BODY>")
 .append("<H1>HTTP Error 404: File Not Found</H1>\r\n")
 .append("</BODY></HTML>\r\n").toString();
 if (version.startsWith("HTTP/")) { // send a MIME header
 sendHeader(out, "HTTP/1.0 404 File Not Found",
 "text/html; charset=utf-8", body.length());
 }
 out.write(body);
 out.flush();
 }
 } else { // method does not equal "GET"
 String body = new StringBuilder("<HTML>\r\n")
 .append("<HEAD><TITLE>Not Implemented</TITLE>\r\n")
 .append("</HEAD>\r\n")
 .append("<BODY>")
 .append("<H1>HTTP Error 501: Not Implemented</H1>\r\n")
 .append("</BODY></HTML>\r\n").toString();
 if (version.startsWith("HTTP/")) { // send a MIME header
 sendHeader(out, "HTTP/1.0 501 Not Implemented",
"text/html; charset=utf-8", body.length());
 }
 out.write(body);
 out.flush();
 }
 } catch (IOException ex) {
```

```
 logger.log(Level.WARNING,
 "Error talking to " + connection.getRemoteSocketAddress(), ex);
 } finally {
 try {
 connection.close();
 }
 catch (IOException ex) {}
 }
 }
 private void sendHeader(Writer out, String responseCode,
 String contentType, int length)
 throws IOException {
 out.write(responseCode + "\r\n");
 Date now = new Date();
 out.write("Date: " + now + "\r\n");
 out.write("Server: JHTTP 2.0\r\n");
 out.write("Content-length: " + length + "\r\n");
 out.write("Content-type: " + contentType + "\r\n\r\n");
 out.flush();
 }
}
```

This server is functional but still rather austere. Here are a few features that could be added:

- A server administration interface
- Support for the Java Servlet API
- Support for other request methods, such as POST, HEAD, and PUT
- Support for multiple document roots so individual users can have their own sites