**Multithreaded Servers in Java**

This trail is about implementing multithreaded servers in Java. Because of Java's built-in multithreading capabilities multithreaded servers are reasonably easy to implement. But not all server designs are equal. This trail will take a look at different server designs and discuss their pros and cons.

If you are new to Java multithreading there is an extensive tutorial here on   
[Java Concurrency / Multithreading](http://tutorials.jenkov.com/java-concurrency/index.html).

This trail on multithreaded servers in Java is still work in progress.

# Singlethreaded Server in Java

This text will show how to implement a singlethreaded server in Java. A singlethreaded server is not the most optimal design for a server, but the code illustrates the life cycle of a server very well. The following texts on multithreaded servers will built upon this code template.

Here is a simple singlethreaded server:

package servers;

import java.net.ServerSocket;

import java.net.Socket;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

public class SingleThreadedServer implements Runnable{

protected int serverPort = 8080;

protected ServerSocket serverSocket = null;

protected boolean isStopped = false;

protected Thread runningThread= null;

public SingleThreadedServer(int port){

this.serverPort = port;

}

public void run(){

synchronized(this){

this.runningThread = Thread.currentThread();

}

openServerSocket();

**while(! isStopped()){**

**Socket clientSocket = null;**

**try {**

**clientSocket = this.serverSocket.accept();**

**} catch (IOException e) {**

**if(isStopped()) {**

**System.out.println("Server Stopped.") ;**

**return;**

**}**

**throw new RuntimeException(**

**"Error accepting client connection", e);**

**}**

**try {**

**processClientRequest(clientSocket);**

**} catch (IOException e) {**

**//log exception and go on to next request.**

**}**

**}**

System.out.println("Server Stopped.");

}

private void processClientRequest(Socket clientSocket)

throws IOException {

InputStream input = clientSocket.getInputStream();

OutputStream output = clientSocket.getOutputStream();

long time = System.currentTimeMillis();

output.write(("HTTP/1.1 200 OK\n\n<html><body>" +

"Singlethreaded Server: " +

time +

"</body></html>").getBytes());

output.close();

input.close();

System.out.println("Request processed: " + time);

}

private synchronized boolean isStopped() {

return this.isStopped;

}

public synchronized void stop(){

this.isStopped = true;

try {

this.serverSocket.close();

} catch (IOException e) {

throw new RuntimeException("Error closing server", e);

}

}

private void openServerSocket() {

try {

this.serverSocket = new ServerSocket(this.serverPort);

} catch (IOException e) {

throw new RuntimeException("Cannot open port 8080", e);

}

}

}

And here is the code to run it:

SingleThreadedServer server = new SingleThreadedServer(9000);

new Thread(server).start();

try {

Thread.sleep(10 \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Stopping Server");

server.stop();

When the server is running you can access it using an ordinary web browser. Use the address <http://localhost:9000/>

## The Server Loop

The most interesting part of the singlethreaded server is its main loop marked in bold in the code above. The loop is repeated here:

while(! isStopped()){

Socket clientSocket = null;

try {

clientSocket = this.serverSocket.accept();

} catch (IOException e) {

if(isStopped()) {

System.out.println("Server Stopped.") ;

return;

}

throw new RuntimeException("Error accepting client connection", e);

}

try {

processClientRequest(clientSocket);

} catch (IOException e) {

//log exception and go on to next request.

}

}

In short what the server does is this:

1. Wait for a client request
2. Process client request
3. Repeat from 1.

This loop is pretty much the same for most servers implemented in Java. What separates the single threaded server from a multithreaded server is that the single threaded server processes the incoming requests in the same thread that accepts the client connection. A multithreaded server passes the connection on to a worker thread that processes the request.

Processing the incoming requests in the same thread that accepts the client connections is not a good idea. Clients can only connect to the server while the server is inside the serverSocket.accept() method call. The longer time the listening thread spends outside the serverSocket.accept() call, the higher the probability that the client will be denied access to the server. This is the reason that multithreaded servers pass the incoming connections on to worker threads, who will process the request. That way the listening thread spends as little time as possible outside the serverSocket.accept() call.

# Multithreaded Server in Java

This text describes a simple multithreaded server implemented in Java. The code is based on the singlethreaded server desbribed in the text on [Singlethreaded Servers](http://tutorials.jenkov.com/java-multithreaded-servers/singlethreaded-server.html). The main difference is the server loop. Rather than processing the incoming requests in the same thread that accepts the client connection, the connection is handed off to a worker thread that will process the request.

Note: This code uses a "thread per connection" design which most of us originally thought less efficient than a [thread pooled server](http://tutorials.jenkov.com/java-multithreaded-servers/thread-pooled-server.html). But read this blog post and think again:   
[Writing Java Multithreaded Servers - whats old is new](http://paultyma.blogspot.com/2008/03/writing-java-multithreaded-servers.html)

Here is how the server loop looks in the multithreaded edition:

while(! isStopped()){

Socket clientSocket = null;

try {

clientSocket = this.serverSocket.accept();

} catch (IOException e) {

if(isStopped()) {

System.out.println("Server Stopped.") ;

return;

}

throw new RuntimeException(

"Error accepting client connection", e);

}

**new Thread(**

**new WorkerRunnable(**

**clientSocket, "Multithreaded Server")**

**).start();**

}

The only change in the loop from the singlethreaded server to here is the code in bold:

new Thread(

new WorkerRunnable(

clientSocket, "Multithreaded Server")

).start();

Rather than processing the incoming requests in the same thread that accepts the client connection, the connection is handed off to a worker thread that processes the request. That way the thread listening for incoming requests spends as much time as possible in the serverSocket.accept() call. That way the risk is minimized for clients being denied access to the server because the listening thread is not inside the accept() call.

Here is the code for the WorkerRunnable class, which is passed to the worker thread constructor:

package servers;

import java.io.InputStream;

import java.io.OutputStream;

import java.io.IOException;

import java.net.Socket;

/\*\*

\*/

public class WorkerRunnable implements Runnable{

protected Socket clientSocket = null;

protected String serverText = null;

public WorkerRunnable(Socket clientSocket, String serverText) {

this.clientSocket = clientSocket;

this.serverText = serverText;

}

public void run() {

try {

InputStream input = clientSocket.getInputStream();

OutputStream output = clientSocket.getOutputStream();

long time = System.currentTimeMillis();

output.write(("HTTP/1.1 200 OK\n\nWorkerRunnable: " +

this.serverText + " - " +

time +

"").getBytes());

output.close();

input.close();

System.out.println("Request processed: " + time);

} catch (IOException e) {

//report exception somewhere.

e.printStackTrace();

}

}

}

## Multithreaded Server Advantages

The advantages of a multithreaded server compared to a singlethreaded server are summed up below:

1. Less time is spent outside the accept() call.
2. Long running client requests do not block the whole server

As mentioned earlier the more time the thread calling serverSocket.accept() spends inside this method call, the more responsive the server will be. Only when the listening thread is inside the accept() call can clients connect to the server. Otherwise the clients just get an error.

In a singlethreaded server long running requests may make the server unresponsive for a long period. This is not true for a multithreaded server, unless the long-running request takes up all CPU time time and/or network bandwidth.

## Multithreaded Server Code

Here is the full code for the MultiThreadedServer:

package servers;

import java.net.ServerSocket;

import java.net.Socket;

import java.io.IOException;

public class MultiThreadedServer implements Runnable{

protected int serverPort = 8080;

protected ServerSocket serverSocket = null;

protected boolean isStopped = false;

protected Thread runningThread= null;

public MultiThreadedServer(int port){

this.serverPort = port;

}

public void run(){

synchronized(this){

this.runningThread = Thread.currentThread();

}

openServerSocket();

while(! isStopped()){

Socket clientSocket = null;

try {

clientSocket = this.serverSocket.accept();

} catch (IOException e) {

if(isStopped()) {

System.out.println("Server Stopped.") ;

return;

}

throw new RuntimeException(

"Error accepting client connection", e);

}

new Thread(

new WorkerRunnable(

clientSocket, "Multithreaded Server")

).start();

}

System.out.println("Server Stopped.") ;

}

private synchronized boolean isStopped() {

return this.isStopped;

}

public synchronized void stop(){

this.isStopped = true;

try {

this.serverSocket.close();

} catch (IOException e) {

throw new RuntimeException("Error closing server", e);

}

}

private void openServerSocket() {

try {

this.serverSocket = new ServerSocket(this.serverPort);

} catch (IOException e) {

throw new RuntimeException("Cannot open port 8080", e);

}

}

}

And here is the code to run it:

MultiThreadedServer server = new MultiThreadedServer(9000);

new Thread(server).start();

try {

Thread.sleep(20 \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Stopping Server");

server.stop();

When the server is running you can access it using an ordinary web browser. Use the address <http://localhost:9000/>

# Thread Pooled Server

This text describes a simple thread pooled server implemented in Java. The code is based on the multithreaded server desbribed in the text on [Multithreaded Servers](http://tutorials.jenkov.com/java-multithreaded-servers/multithreaded-server.html). The main difference is the server loop. Rather than starting a new thread per incoming connection, the connection is wrapped in a Runnable and handed off to a thread poool with a fixed number of threads. The Runnable's are kept in a queue in the thread pool. When a thread in the thread pool is idle it will take a Runnable from the queue and execute it.

Note: Thread pools are discussed in more detail in the text [Thread Pools](http://tutorials.jenkov.com/java-concurrency/thread-pools.html).

Here is how the server loop looks in the thread pooled edition (the full code is shown at the bottom of this text):

while(! isStopped()){

Socket clientSocket = null;

try {

clientSocket = this.serverSocket.accept();

} catch (IOException e) {

if(isStopped()) {

System.out.println("Server Stopped.") ;

return;

}

throw new RuntimeException(

"Error accepting client connection", e);

}

**this.threadPool.execute(**

**new WorkerRunnable(clientSocket, "Thread Pooled Server"));**

}

The only change in the loop from the multithreaded server to here is the code in bold:

this.threadPool.execute(

new WorkerRunnable(clientSocket, "Thread Pooled Server"));

Rather than starting a new thread per incoming connection, the WorkerRunnable is passed to the thread pool for execution when a thread in the pool becomes idle.

Here is the code for the WorkerRunnable class, which is passed to the worker thread constructor:

package servers;

import java.io.InputStream;

import java.io.OutputStream;

import java.io.IOException;

import java.net.Socket;

public class WorkerRunnable implements Runnable{

protected Socket clientSocket = null;

protected String serverText = null;

public WorkerRunnable(Socket clientSocket, String serverText) {

this.clientSocket = clientSocket;

this.serverText = serverText;

}

public void run() {

try {

InputStream input = clientSocket.getInputStream();

OutputStream output = clientSocket.getOutputStream();

long time = System.currentTimeMillis();

output.write(("HTTP/1.1 200 OK\n\nWorkerRunnable: " +

this.serverText + " - " +

time +

"").getBytes());

output.close();

input.close();

System.out.println("Request processed: " + time);

} catch (IOException e) {

//report exception somewhere.

e.printStackTrace();

}

}

}

## Thread Pooled Server Advantages

The advantages of a thread pooled server compared to a multithreaded server is that you can control the maximum number of threads running at the same time. This has certain advantages.

First of all if the requests require a lot of CPU time, RAM or network bandwidth, this may slow down the server if many requests are processed at the same time. For instance, if memory consumption causes the server to swap memory in and out of disk, this will result in a serious performance penalty. By controlling the maximum number of threads you can minimize the risk of resource depletion, both due to limiting the memory taken by the processing of the requests, but also due to the limitation and reuse of the threads. Each thread take up a certain amount of memory too, just to represent the thread itself.

Additionally, executing many requests concurrently will slow down all requests processed. For instance, if you process 1.000 requests concurrently and each request takes 1 second, then all requests will take 1.000 seconds to complete. If you instead queue the requests up and process them say 10 at a time, the first 10 requests will complete after 10 seconds, the next 10 will complete after 20 seconds etc. Only the last 10 requests will complete after 1.000 seconds. This gives a better service to the clients.

## Thread Pooled Server Code

Here is the full code for the ThreadPooledServer:

package servers;

import java.net.ServerSocket;

import java.net.Socket;

import java.io.IOException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ThreadPooledServer implements Runnable{

protected int serverPort = 8080;

protected ServerSocket serverSocket = null;

protected boolean isStopped = false;

protected Thread runningThread= null;

protected ExecutorService threadPool =

Executors.newFixedThreadPool(10);

public ThreadPooledServer(int port){

this.serverPort = port;

}

public void run(){

synchronized(this){

this.runningThread = Thread.currentThread();

}

openServerSocket();

while(! isStopped()){

Socket clientSocket = null;

try {

clientSocket = this.serverSocket.accept();

} catch (IOException e) {

if(isStopped()) {

System.out.println("Server Stopped.") ;

return;

}

throw new RuntimeException(

"Error accepting client connection", e);

}

this.threadPool.execute(

new WorkerRunnable(clientSocket,

"Thread Pooled Server"));

}

this.threadPool.shutdown();

System.out.println("Server Stopped.") ;

}

private synchronized boolean isStopped() {

return this.isStopped;

}

public synchronized void stop(){

this.isStopped = true;

try {

this.serverSocket.close();

} catch (IOException e) {

throw new RuntimeException("Error closing server", e);

}

}

private void openServerSocket() {

try {

this.serverSocket = new ServerSocket(this.serverPort);

} catch (IOException e) {

throw new RuntimeException("Cannot open port 8080", e);

}

}

}

And here is the code to run it:

ThreadPooledServer server = new ThreadPooledServer(9000);

new Thread(server).start();

try {

Thread.sleep(20 \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Stopping Server");

server.stop();

When the server is running you can access it using an ordinary web browser. Use the address <http://localhost:9000/>