GeneralHttpCoreIntroduction

General HttpCore Introduction

This guide and those derived from it will discuss the basic usage of HttpCore as an HTTP client. A class will be built that connects to the specified host and retrieves the selected URL. An InputStream and a facility to read the response headers are supplied to read the result of the connection.

From here I will describe some ways to expand on this class, by improving weaknesses, implementing a smarter thread safe connection manager, facilities to send custom headers, do POST requests, SSL, NIO, and so on.

It's not recommended to use this class for production, as is. This is mainly because some bad practises are used. It is sufficient to retrieve one URL here or there, but for high concurrency and mission critical HTTP connections, much more needs to be done to make it stronger.

At the end a completely production ready class will be shown.

The basic structure of the class

The basic structure of this class is in the format of a "request". The idea is to create an HTTP request which can be executed, and there after the response read from. It doesn't work exactly as HttpClient where all these things are split into separate components/classes, but this isn't the ideal I'm chasing after.

I'm doing all this purely for demonstrating the basic usage of HttpCore.

We will be calling the basic class org.apache.http.example.SimpleHttpRequest. It will contain a main method to easily run and experiment with it.

The complete source code for the classes built in on this page are found at the end of this page. See the table of contents above.

I also recommend reading GuidedTourOfHttpCore for more information on the concepts, ideas and classes for HttpCore.

Static Member Setup

For all requests there are some objects which are shared. These do not need new instances, as they are for things like options, classes required by HttpCore to process/execute the requests/responses.

These we will setup once the class is loaded for the first time, and save them as static variables.

The first of these are the BasicHttpParams class. It is used to specify HTTP protocol parameters used by HttpCore to configure certain behaviours/headers.

Parameters work in a "heritage". Those you don't specify will be read from any parent parameters objects. So you have quite a bit of control over your HttpCore/HttpClient configuration. The exact details of this is beyond this scope guide.

We will be setting the protocol version, user agent, character set and the "expect continue" option with this.

The next static member is BasicHttpProcessor. This is basically an ordered list of interceptors. You get 2 types of interceptors, request and response. Each handles it's respective step in the request/response of the connection, and will be called before during these phases of the connection. They are used to create/process request/response headers, often just one header per interceptor. To more easily explain their purpose, I'll give an example of a request interceptor, the RequestTargetHost request interceptor. This is called during the request phase, and sets the "Host" header for HTTP requests.

Interceptors are called in the order they were added to the processor, in each respective phase of the connection (before/after you execute the request). To create your own interceptors you simple need to override either HttpRequestInterceptor or HttpResponseInterceptor and add an instance of your class to your processor.

There are 2 required request interceptors, and 3 required response interceptors. You need to add these to have HttpCore work correctly. There are no limits to the amount you can add after this.

Later I will demonstrate implementing a custom interceptor by creating a response interceptor that modifies all HTML documents retrieved by adding a special header to the response. It's purpose is useless, but demonstrates it's purposes.

Next up is the HttpRequestExecutor, a very simple object, and the connection manager. Basically the HttpRequestExecutor is just a utility class for executing the separate steps of the request, specifically the interceptors and the request itself. The connection manager is a class we will be making later to manage the persistent or keep-alive connections and their sockets. HttpCore doesn't have any built in connection management. It does however have classes that aid in connection manager, like for connection reuse strategies, which we'll use later on to improve on our basic connection manager.

Here is the code to setup the static members of the class.

```
numbers=off
 private static final HttpParams params;

 private static final BasicHttpProcessor httpProcessor;

 private static final HttpRequestExecutor httpExecutor;

 private static final ConnectionManager connectionManager;

 static
 {
   // parameters
   params = new BasicHttpParams();
   HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
   HttpProtocolParams.setContentCharset(params, "UTF-8");
   HttpProtocolParams.setUserAgent(params, "HttpComponents/1.1");
   HttpProtocolParams.setUseExpectContinue(params, true);

   // processors
   httpProcessor = new BasicHttpProcessor();
   // Required protocol interceptors
   httpProcessor.addInterceptor(new RequestContent());
   httpProcessor.addInterceptor(new RequestTargetHost());
   // Recommended protocol interceptors
   httpProcessor.addInterceptor(new RequestConnControl());
   httpProcessor.addInterceptor(new RequestUserAgent());
   httpProcessor.addInterceptor(new RequestExpectContinue());

   // execturor
   httpExecutor = new HttpRequestExecutor();

   // connection manager
   connectionManager = new ConnectionManager(params);
 }
```

Context

The context is used to tie together the different parts of the complete request/response. It's also used to store some internal data that needs to be shared between these classes.

The context is configured by setting attributes with the void setAttribute(String id, Object obj) method. Ids for context attributes are defined in the Execution Context class, and each attribute takes a class as a value. Attribute names take the form of property names, and the ExecutionContext constants are simply String objects creating an abstraction over these property names.

```
 numbers=off
private final HttpContext context = new BasicHttpContext(null);
```

Construction

Now we can start setting up the particulars for the request itself.

This is the constructor we are going to use:

```
numbers=off
 private final HttpRequest request;

 private final DefaultHttpClientConnection connection;

 public SimpleHttpRequest(String url) throws IOException
 {
   this(new URL(url));
 }

 public SimpleHttpRequest(URL url) throws IOException
 {
   // create the request
   request = new BasicHttpRequest("GET", url.toString(), HttpVersion.HTTP_1_1);
   context.setAttribute(ExecutionContext.HTTP_REQUEST, request);
   request.setParams(params);

   // set the host to send as a host header
   HttpHost requestHost = new HttpHost(url.getHost(), url.getPort());
   context.setAttribute(ExecutionContext.HTTP_TARGET_HOST, requestHost);

   // fetch a connection from the connection manager
   connection = connectionManager.getConnection(url);
   context.setAttribute(ExecutionContext.HTTP_CONNECTION, connection);
 }
```

The connection manager will be discussed at a later point. For now it will be a simple class that creates and keeps open some connections new connection upon request. No thread safety will be built into it for now. Later when we make it more advanced enough attention will be given to make it production ready.

```
numbers=off
   // create the request
   request = new BasicHttpRequest("GET", url.toString(), HttpVersion.HTTP_1_1);
   context.setAttribute(ExecutionContext.HTTP_REQUEST, request);
   request.setParams(params);
```

This basically constructs a simple HttpRequest object for the given URL, to use the "GET" method and the HTTP/1.1 protocol. We also need to tell the context about it and set it's configuration with the HttpRequest.setParams() method.

As seen the ExecutionContext.HTTP_REQUEST attribute (http.request) takes a HttpRequest object as it's value.

```
numbers=off
   // set the host to send as a host header
   HttpHost requestHost = new HttpHost(url.getHost(), url.getPort());
   context.setAttribute(ExecutionContext.HTTP_TARGET_HOST, requestHost);
```

Next we need to set the value which will be sent for the Host header. The HttpHost object is used by HttpComponents whenever a hostname/port pair is required. Some examples are for proxies, routes, virtual hosts, host headers and target hosts. Since it represents a host for an HTTP connection, it can take one extra parameter, which is the scheme. If you would be using SSL/TLS, you would set the scheme to "https". The default scheme is "http", and sufficient for our current class. Also, not all places where HttpHost is accepted use the scheme. The parameter is present, but used only where the implementation requires this information. More on this later.

Again we set the ExecutionContext.HTTP_TARGET_HOST context attribute (http.target_host), which takes an HttpHost object as value.

```
numbers=off
   // fetch a connection from the connection manager
   connection = connectionManager.getConnection(url);
   context.setAttribute(ExecutionContext.HTTP_CONNECTION, connection);
```

This simply fetches a connection from the connection manager, and configures the current instance's context to use it, by setting the "http.connection" context attribute.

As you probably noticed I also added a convenience constructor to which you pass a String url, instead of a URL object. This simple initiates the real constructor by create a URL object from the String representation.

Executing the request and receiving the response

```
numbers=off
 private HttpResponse response;

 private InputStream inputStream;

 public void execute() throws HttpException, IOException
 {
   httpExecutor.preProcess(request, httpProcessor, context);
   response = httpExecutor.execute(request, connection, context);
   httpExecutor.postProcess(response, httpProcessor, context);

   inputStream = response.getEntity().getContent();
 }
```

This is the icing on the cake. Here we execute the connection, and save a reference to the input stream.

As you can see here, we run the interceptors before and after we execute the request. This is done with the httpExecutor.preProcess(...) and httpExecutor.postProcess(...) methods. It also becomes obvious here why I referred to the HttpExecutor class as a utility class, and why you only need one to share between all your request instances. It doesn't store any request specified information, but merely uses the objects supplied to each of it's methods to execute the relevant step of the HTTP request. The HttpContext class is used by HttpExecute to store instance type information.

After we finished the complete execution procedure, we can store the input stream so as to later retrieve the content. To explain how content is retrieved, let me first explain the entity.

HttpEntity

A body of content in HttpCore and HttpClient is represented by the HttpEntity class. Whether it's a body received from a response, or a body sent by a request (POST/PUT), the HttpEntity is your friend.

HttpEntity has a few methods for retrieving the length of the content (when available), the encoding, and some information like whether it's chunked /repeatable and so on.

For this guide we will only be discussing response entities. To retrieve the content from these entities you retrieve the InputStream object with the HttpEntity .getContent() method.

The EntityUtils class has 4 static utility methods for dealing with entity objects. Three of these are used to consume the whole stream and store it in either a byte array or a String. This is an example of their use:

```
numbers=off
   String charset = EntityUtils.getContentCharSet(response.getEntity());
   byte[] data = EntityUtils.toByteArray(response.getEntity());
   String data = EntityUtils.toString(response.getEntity());
   String data = EntityUtils.toString(response.getEntity(), "UTF-8");
```

To learn more about the HttpEntity and it's different implementations, see HttpEntity.

Closing/releasing the connection/stream

A very important part of the request is to release the connection to the connection pool. If you don't have a connection pool, it's just as important to close it, since having idle connections float around unnecessarily consumes valuable memory. They will eventually time out, but it's still bad practise to not close them.

The requirements when releasing a connection is to ensure that all remaining data from the response are read, so a next response can send it's request, then to also close the input stream. After this we can release the connection back to the pool.

We catch and ignore all exceptions thrown here, to ensure the next step at least completes. For instance if a null pointer exception occurs because you didn't execute the request, at least the connection will be close/released.

```
numbers=off
 public void close()
 {
   try
   {
     HttpEntity entity = response.getEntity();
     if (entity != null)
     {
       entity.consumeContent();
     }
   }
   catch (Exception e) {}

   try
   {
     if (inputStream != null)
     {
       inputStream.close();
     }
   }
   catch (Exception e) {}

   try
   {
     connectionManager.releaseConnection(connection);
   }
   catch (IOException e) {}
 }
```

Finishing off the SimpleHttpRequest class

We have completed all the necessary steps to setup and execute a basic HttpCore request, and retrieve the response stream.

All we need to do now is add some utility methods for reading the headers and retrieving the stream so itself can be read from.

Here they are, they are pretty self explanatory, though we will be discussing the Header class.

```
numbers=off
 public Header[] getHeader(String name) throws HttpException
 {
   return response.getHeaders(name);
 }

 public Header[] getAllHeaders() throws HttpException
 {
   return response.getAllHeaders();
 }

 public long getContentLength()
 {
   return response.getEntity().getContentLength();
 }

 public String getContentType()
 {
   try
   {
     return response.getEntity().getContentType().getValue();
   }
   catch (Exception e)
   {
     return null;
   }
 }

 public InputStream getContentStream()
 {
   return inputStream;
 }
```

The Header class

A header class represents a request/response header.

For those who aren't familiar with the internal details of the HTTP protocol, I seriously recommend reading the RFC (See ReferenceMaterials). An HTTP header takes the name/value pair form of "Name: value1; value2; value3;" and so on. It's very similar to the MIME standard, with one difference of not having a maximum length limit, due to very long post/cookie headers.

Headers are used to specify options/information to be sent to the server during a response, or for reading details about the response received from the server. Some of the most common uses for these are cookies/post variables, content type, content length and all these you've probably used before. Content length and content type are so common, that the HttpEntity class has methods named after them for easily retrieving their values.

When reading a header, you can either request an array of all headers, or select only a specific header by specifying it's name.

There can be more than one header in a request/response by the same name, which is why the getHeader() method returns an array. A single header line can also have more than one value separated by a semi-colon. These are called elements.

To retrieve the value of a header, call the Header.getValue() method. Similarly to retrieve the name you would call the Header.getName() method.

For more details on working with headers refer to the API documentation.

Simple Connection Manager

Before we can run our HttpCore class, we need to complete the last step, which is to make the connection manager.

HttpCore has no connection management, which makes it very flexible, as the connection management is purely left to you, and no limits are placed as to how you have to do this. You need to create the connection yourself and specify the socket this connection uses. The flexibility comes in where you can use any network connection methods, as long as you supply a java.net.Socket object to HttpCore's connection wrapper.

To learn more about the connection management design in HttpComponents, specifically HttpClient, see ClientConnectionManagementDesign.

```
 numbers=off
public class ConnectionManager
{
  // max requests per connection
  private static final int MAX_PERSISTENT_REQUEST = 100;
```

```
// connection configuration parameters
private HttpParams params;

// list of open connections
private List<HttpClientConnectionWrapper> connections;

// wrapper class to track whether a connection is in use
private class HttpClientConnectionWrapper extends DefaultHttpClientConnection
{
  boolean inUse = false;
  int useCount;
}

public ConnectionManager(HttpParams params)
{
  this.params = params;
  connections = new LinkedList<HttpClientConnectionWrapper>();
}

public DefaultHttpClientConnection getConnection(URL url)
    throws IOException
{
  // check to see if we have an available connection
  synchronized (connections)
  {
    Iterator<HttpClientConnectionWrapper> iter = connections.iterator();
    while (iter.hasNext())
    {
      HttpClientConnectionWrapper connection = iter.next();

      // if the connection is closed remove it
      if (!connection.isOpen())
      {
        iter.remove();
        continue;
      }

      // it's open, is it in use?
      if (!connection.inUse)
      {
        connection.inUse = true;
        connection.useCount++;
        return connection;
      }
    }
  }

  // we haven't found one, make a new one
  return makeNewConnection(url);
}

private DefaultHttpClientConnection makeNewConnection(URL url)
    throws IOException
{
  Socket socket = new Socket(url.getHost(), url.getPort());

  HttpClientConnectionWrapper connection = new HttpClientConnectionWrapper();
  connection.bind(socket, params);
  connection.inUse = true;
  synchronized (connections)
  {
    connections.add(connection);
  }
  return connection;
}

public void releaseConnection(DefaultHttpClientConnection connection)
    throws IOException
{
  if (!(connection instanceof HttpClientConnectionWrapper))
```

```
    {
      throw new IllegalArgumentException("*slap* Invalid connection.");
    }
    HttpClientConnectionWrapper con = ((HttpClientConnectionWrapper)connection);

    // reached max keep-alive requests, close it
    if (con.useCount >= MAX_PERSISTENT_REQUEST)
    {
      con.close();
      connections.remove(con);
      return;
    }

    con.inUse = false;
  }
}
```

This is the simple connection manager class. The wrapper class HttpClientConnectionWrapper simply adds to variables to the standard connection class, inUse and useCount. The former marks the connection is unavailable for getConnection() calls, and the latter counts how many times it has been used. The reason for this is to lighten the weight of individual connections, as well as avoid exceeding the request limit per connection.

There are better ways to implement this limit, but for now this is sufficient. When the limit has been reached on the server, a NoHttpResponseException will be thrown when you execute the request. If you receive such an exception, try reducing the MAX_PERSISTENT_REQUESTS constant.

This isn't ideal, but sufficient for focusing on demonstrating the basic usage of HttpCore. Later we will build around these weaknesses, making it a stronger, more reliable manage.

The manager is constructed by supplying the HttpParams object used by the request. To create a new connection we supply the URL we want to fetch. This is used to retrieve the hostname/port we wish to connect to. From here we check to see if our list has one that isn't in use and return it. If there is no such connection we create the Socket and the HttpClientConnectionWrapper, then bind the Socket to the connection object, add it to our list and return it. Simple.

Pedal to the metal

All we need to do now is test it. The usage of the class is very simple. You construct it with a URL, execute it, read from it and close it.

We accept a URL from the command line. If it's not present an error will be shown and execution will be ceased with a non-zero exit code. Then we build /execute the request and print the response to standard output. If an exception occurs (connection failure, invalid URL, etc.) during the creation/execution of the request, we again catch and display it.

```
numbers=off
 public static void main(String[] args)
 {
   if (args.length < 1)
   {
     System.err.println("Please supply a URL to fetch as a first argument.");
     Runtime.getRuntime().exit(1);
   }

   SimpleHttpRequest request = null;
   try
   {
     final int BUFFER_SIZE = 128;

     request = new SimpleHttpRequest(args[0]);
     request.execute();
     BufferedInputStream inputStream =
       new BufferedInputStream(request.getContentStream());

     byte[] buf = new byte[BUFFER_SIZE];
     int read = 0;
     while ((read = inputStream.read(buf, 0, BUFFER_SIZE)) > -1)
     {
       System.out.print(new String(buf, 0, read));
     }
   }
   catch (MalformedURLException e)
   {
     System.err.println("Invalid URL.");
     e.printStackTrace();
   }
   catch (IOException e)
   {
     System.err.println("IOException caught.");
     e.printStackTrace();
   }
   catch (HttpException e)
   {
     System.err.println("HttpException caught.");
     e.printStackTrace();
   }
   finally
   {
     if (request != null)
     {
       request.close();
     }
   }
 }
```

Just a note on the buffer size I use to read the input stream. It's important to not pick a too large buffer size, as InputStreams are blocked IO. To put this more clearly, if you pick a buffer size of 10kb, but it's a 1kb/s connection, then you will block for 10 seconds on every read. Making a small buffer size allows you to write to the output stream/handle the data as these little bits of data become available, and this can cause a minor performance increase for locally handled data, or smoother flowing output streams, especially when the output stream is another network connection.

Conclusion

And... That's it, simply put.

Some weaknesses that needs to be addressed are server side request limits. If the server only allows 10 requests per connection, our class will fail with an exception after it has done 10 connections. In general much better error handling needs to be implemented.

The connection is created upon construction of the request object. Later we will modify the class and the connection manager to create a connection only the moment it is required. We will also implement a maximum concurrent connections, a total and on a per host basis.

We will implement support for POST requests. Also handlers that detect connection failure, and tries to recover from this without making a request fail.

We will also make a stream reader thread, that pre-buffers the response content, to allow a connection to be released earlier. This will be configurable, so if you do the request, but only much later read the response, the connection won't be hogged for too long.

Authentication, SSL, thread safety, built in cache and much more. The idea behind these guides is to run through miscellaneous ideas surrounding HttpCore , to basically demonstrate the different parts of it while also giving useable examples.

Many others, the developers and myself are available on the HttpComponents users mailing list for any questions or guidance you might have.

Hope this guide was helpful.

Source Code for Classes

SimpleHttpRequest.java

```
 numbers=off
package org.apache.http.example;

import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.MalformedURLException;
import java.net.URL;

import org.apache.http.Header;
import org.apache.http.HttpEntity;
import org.apache.http.HttpException;
import org.apache.http.HttpHost;
import org.apache.http.HttpRequest;
import org.apache.http.HttpResponse;
import org.apache.http.HttpVersion;
import org.apache.http.impl.DefaultHttpClientConnection;
import org.apache.http.message.BasicHttpRequest;
import org.apache.http.params.BasicHttpParams;
import org.apache.http.params.HttpParams;
import org.apache.http.params.HttpProtocolParams;
import org.apache.http.protocol.BasicHttpContext;
import org.apache.http.protocol.BasicHttpProcessor;
import org.apache.http.protocol.ExecutionContext;
import org.apache.http.protocol.HttpContext;
import org.apache.http.protocol.HttpRequestExecutor;
import org.apache.http.protocol.RequestConnControl;
import org.apache.http.protocol.RequestContent;
import org.apache.http.protocol.RequestExpectContinue;
import org.apache.http.protocol.RequestTargetHost;
import org.apache.http.protocol.RequestUserAgent;


public class SimpleHttpRequest
{
  private static final HttpParams params;

  private static final BasicHttpProcessor httpProcessor;

  private static final HttpRequestExecutor httpExecutor;

  private static final ConnectionManager connectionManager;

  private final HttpContext context = new BasicHttpContext(null);

  private final HttpRequest request;

  private final DefaultHttpClientConnection connection;

  private HttpResponse response;

  private InputStream inputStream;

  static
  {
    // parameters
    params = new BasicHttpParams();
```

```
  HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
  HttpProtocolParams.setContentCharset(params, "UTF-8");
  HttpProtocolParams.setUserAgent(params, "HttpComponents/1.1");
  HttpProtocolParams.setUseExpectContinue(params, true);

  // processors
  httpProcessor = new BasicHttpProcessor();
  // Required protocol interceptors
  httpProcessor.addInterceptor(new RequestContent());
  httpProcessor.addInterceptor(new RequestTargetHost());
  // Recommended protocol interceptors
  httpProcessor.addInterceptor(new RequestConnControl());
  httpProcessor.addInterceptor(new RequestUserAgent());
  httpProcessor.addInterceptor(new RequestExpectContinue());

  // executuror
  httpExecutor = new HttpRequestExecutor();

  // connection manager
  connectionManager = new ConnectionManager(params);
}

public SimpleHttpRequest(String url) throws MalformedURLException, IOException
{
  this(new URL(url));
}

public SimpleHttpRequest(URL url) throws IOException
{
  // create the request
  request = new BasicHttpRequest("GET", url.toString(), HttpVersion.HTTP_1_1);
  context.setAttribute(ExecutionContext.HTTP_REQUEST, request);
  request.setParams(params);

  // set the host to send as a host header
  HttpHost requestHost = new HttpHost(url.getHost(), url.getPort());
  context.setAttribute(ExecutionContext.HTTP_TARGET_HOST, requestHost);

  // fetch a connection from the connection manager
  connection = connectionManager.getConnection(url);
  context.setAttribute(ExecutionContext.HTTP_CONNECTION, connection);
}

public void execute() throws HttpException, IOException
{
  httpExecutor.preProcess(request, httpProcessor, context);
  response = httpExecutor.execute(request, connection, context);
  httpExecutor.postProcess(response, httpProcessor, context);

  inputStream = response.getEntity().getContent();
}

public void close()
{
  try
  {
    HttpEntity entity = response.getEntity();
    if (entity != null)
    {
      entity.consumeContent();
    }
  }
  catch (Exception e) {}

  try
  {
    if (inputStream != null)
    {
      inputStream.close();
    }
  }
```

```
  catch (Exception e) {}

  try
  {
    connectionManager.releaseConnection(connection);
  }
  catch (IOException e) {}
}

public Header[] getHeader(String name) throws HttpException
{
  return response.getHeaders(name);
}

public Header[] getAllHeaders() throws HttpException
{
  return response.getAllHeaders();
}

public long getContentLength()
{
  return response.getEntity().getContentLength();
}

public String getContentType()
{
  try
  {
    return response.getEntity().getContentType().getValue();
  }
  catch (Exception e)
  {
    return null;
  }
}

public InputStream getContentStream()
{
  return inputStream;
}

public static void main(String[] args)
{
  if (args.length < 1)
  {
    System.err.println("Please supply a URL to fetch as a first argument.");
    Runtime.getRuntime().exit(1);
  }

  SimpleHttpRequest request = null;
  try
  {
    final int BUFFER_SIZE = 128;

    request = new SimpleHttpRequest(args[0]);
    request.execute();
    BufferedInputStream inputStream =
      new BufferedInputStream(request.getContentStream());

    byte[] buf = new byte[BUFFER_SIZE];
    int read = 0;
    while ((read = inputStream.read(buf, 0, BUFFER_SIZE)) > -1)
    {
      System.out.print(new String(buf, 0, read));
    }
  }
  catch (MalformedURLException e)
  {
    System.err.println("Invalid URL.");
    e.printStackTrace();
  }
```

```
    catch (IOException e)
    {
      System.err.println("IOException caught.");
      e.printStackTrace();
    }
    catch (HttpException e)
    {
      System.err.println("HttpException caught.");
      e.printStackTrace();
    }
    finally
    {
      if (request != null)
      {
        request.close();
      }
    }
  }
}
```

ConnectionManager.java

```
 numbers=off
package org.apache.http.example;

import java.io.IOException;
import java.net.Socket;
import java.net.URL;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import org.apache.http.impl.DefaultHttpClientConnection;
import org.apache.http.params.HttpParams;

public class ConnectionManager
{
  // max requests per connection
  private static final int MAX_PERSISTENT_REQUEST = 100;

  // connection configuration parameters
  private HttpParams params;

  // list of open connections
  private List<HttpClientConnectionWrapper> connections;

  // wrapper class to track whether a connection is in use
  private class HttpClientConnectionWrapper extends DefaultHttpClientConnection
  {
    boolean inUse = false;
    int useCount;
  }

  public ConnectionManager(HttpParams params)
  {
    this.params = params;
    connections = new LinkedList<HttpClientConnectionWrapper>();
  }

  public DefaultHttpClientConnection getConnection(URL url)
      throws IOException
  {
    // check to see if we have an available connection
    synchronized (connections)
    {
      Iterator<HttpClientConnectionWrapper> iter = connections.iterator();
      while (iter.hasNext())
      {
```

```
        HttpClientConnectionWrapper connection = iter.next();

        // if the connection is closed remove it
        if (!connection.isOpen())
        {
          iter.remove();
          continue;
        }

        // it's open, is it in use?
        if (!connection.inUse)
        {
          connection.inUse = true;
          connection.useCount++;
          return connection;
        }
      }
    }

    // we haven't found one, make a new one
    return makeNewConnection(url);
  }

  private DefaultHttpClientConnection makeNewConnection(URL url)
      throws IOException
  {
    Socket socket = new Socket(url.getHost(), url.getPort());

    HttpClientConnectionWrapper connection = new HttpClientConnectionWrapper();
    connection.bind(socket, params);
    connection.inUse = true;
    synchronized (connections)
    {
      connections.add(connection);
    }
    return connection;
  }

  public void releaseConnection(DefaultHttpClientConnection connection)
      throws IOException
  {
    if (!(connection instanceof HttpClientConnectionWrapper))
    {
      throw new IllegalArgumentException("*slap* Invalid connection.");
    }
    HttpClientConnectionWrapper con = ((HttpClientConnectionWrapper)connection);

    // reached max keep-alive requests, close it
    if (con.useCount >= MAX_PERSISTENT_REQUEST)
    {
      con.close();
      connections.remove(con);
      return;
    }

    con.inUse = false;
  }
}
```