Web Server Core
Cpt S 422 Homework Assignment
by Evan Olds

Assignment Instructions:

Read all instructions *carefully* before you write any code.

In this assignment you will extend your web server to have proper concurrency and the ability to support web services/apps on top of the core.

Implement the ConcatStream Class

Implement the ConcatStream class in the CS422 namespace. Remember to match the naming exactly, including casing. This is the class we discussed in lecture, that inherits from Stream and concatenates two streams together into one. The overall functionality of the ConcatStream is limited by the lesser-functional of the two streams being concatenated. So the ConcatStream can only seek if both streams can seek, can only read if both streams can read, and so on. Implement two constructors in this class:

```
ConcatStream(Stream first, Stream second)
ConcatStream(Stream first, Stream second, long fixedLength)
```

The first constructor takes two Stream objects, and doesn't need to support the "Length" property unless both streams support it. However, it's only the second stream that is permitted to not have a length. If the first of the two does not have a length, then it's not possible to know where it ends and the next stream begins, so in this case throw an exception. Given that a stream that doesn't have a length generally throws an exception when you access its Length property, you can just let that exception bubble up to outside of the constructor. The ConcatStream cannot be implemented without knowing the length of the first stream.

The second stream, on the other hand, should be allowed to be a stream without a length or the capability to seek, such as a NetworkStream. If the second stream doesn't support seeking, have the ConcatStream provide forward-only reading functionality with no seeking.

Much of the above applies to the second constructor as well, only this time even if the second stream does not have a length, you will support the Length property. Have the second constructor store the fixed length value and return it in the getter for the Length property. You'll also need some way of storing information about which of the two constructors was used to instantiate the class, since the Length property behaves differently based on how the ConcatStream is constructed.

Implement the WebRequest class

Write a public class named WebRequest in the CS422 namespace. Note that there is a class named WebRequest in the System.Net namespace, so don't confuse the two. Recall from the notes and lecture that this class represents data parsed from a request and utility functions to respond. It must contain the HTTP method, URI (request target), HTTP version, dictionary of headers, and a single Stream object for the body of a request. It also must contain a private reference to the NetworkStream that can be used to write a response. The body will often be a ConcatStream object, since the request object is created after you've seen received the double line break after the headers and likely after you've read some of the body data from the network stream. The first byte of the body stream must be the first byte of the actual request body.

If the "Content-Length" header is present in the request, then the body stream will support querying the length. Otherwise querying the length will throw an exception. Make sure that the ConcatStream class supports "mocking" the length if this header is present.

Add 2 response-sending functions to the request class. Web services/handlers, which are described in the next section, will use these functions to write responses to the requests. The request class object should store a reference to the network stream, so it has the capability to write a response to that stream. You'll need to review the format of an HTTP response in order to properly implement these functions. Their signatures follow.

```
public void WriteNotFoundResponse(string pageHTML)
public bool WriteHTMLResponse(string htmlString)
```

The first writes a response with a 404 status code and the specified HTML string as the body of the response. The second writes a response with a 200 status code and the specified HTML string as the body of the response. This allows the handlers/services, or the web server core in the case of no appropriate service for the URI, to easily write responses to the clients.

Both functions create and write the response line, response headers, and double break before writing the HTML string to the network stream. In other words, the strings for HTML content are to be the bodies of the responses. For the response headers, include at a minimum:

```
Content-Type: text/html
```

and

```
Content-Length: ___
```

where "___" is replaced with the actual content (body) length, in bytes.

Implement the WebService Class

Implement an abstract base class for a web "service". This is essentially a web app that runs on top of the server core. When the server gets a new connection, it processes it on a thread pool thread, builds a request object, finds a service/handler for this request, and then sends the request to the handler to process. The signature for the class is below. The “Handler” function should be self-explanatory and the comment on the `ServiceURI` property should suffice to explain its purpose.

```
internal abstract class WebService
{
    public abstract void Handler(WebRequest req);

    /// <summary>
    /// Gets the service URI. This is a string of the form:
    /// /MyServiceName.whatever
    /// If a request hits the server and the request target starts with this
    /// string then it will be routed to this service to handle.
    /// </summary>
    public abstract string ServiceURI
    {
        get;
    }
}
```

Extend the WebServer class

From a previous assignment, you should already have a class named “WebServer” in the CS422 namespace. Since you had this from a previous assignment, you should be able to copy and paste some of the code and alter/extend it. In that code there was a “Start” function.

Change the Start function:

In this assignment the “Start” function in the WebServer class will still be static but now take 2 integer parameters:

- The first is an integer (Int32) for the port to listen on. The server will continue listening for connections on this port until the “Stop” function is called (discussed later).
- The second is an integer (Int32) for the number of threads in the thread pool. If this value is non-positive, us 64 as a default instead.

The static “Start” function is a non-blocking function that starts the server listening on the specified port. Since it is non-blocking, this implies that you should be creating a new “listen thread”, where you do the following:

- Accept new TCP socket connection
- Get a thread from the thread pool and pass it the TCP socket
- Repeat

Implement the BuildRequest function:

Have a member function in the WebServer class that's designed to build a request object from a TCP client. This function takes a TcpClient object, reads a request from it, builds a request object, and returns it. Should the request from the client be invalid, it must return null. All the error checking from the previous web server homework assignment should be present to ensure that it only returns a non-null request object if the TCP client did indeed send a valid HTTP request. Below are some additional requirements for handling requests and implementing the BuildRequest function:

1. The signature must be:
   ```
   private WebRequest BuildRequest(TcpClient client)
   ```
2. You still only need to support the "GET" method at this time. In future assignments you will implement other HTTP methods.
3. As mentioned previously, the function returns null if the request is invalid. In this case the client must be disposed and null returned. The requirements for an invalid request are the same as they were in the previous web server coding assignment.
4. If a non-null request object is returned, implying a valid HTTP request was sent, then the TcpClient object must NOT be disposed, because you will still need access to the network stream to send the response.

Implement the ThreadWork function:

Implement the ThreadWork function to serve as the request-handling function for a thread from the thread pool. Like the thread pool implemented in a previous homework assignment, there should be threads sitting idle until a TCP client is put in a shared queue, then one of the idle threads will wake up and process that client.

Processing a client involves calling the BuildRequest function first. Should the return value be null, the client connection is closed immediately and the thread goes back into a waiting/sleeping state. Should it be non-null, then an appropriate handler is found in the list of handlers/services that the web server stores, and the request is sent to that handler by calling its "Handle" function. The service "Handle" function is expected to write a response using the request object. Should no such handler exist, write a *404 – Not Found* response. Close the client connection and dispose the NetworkStream and TcpClient after either of these scenarios (i.e. either handling with an appropriate handler or writing a 404).

Implement the AddService function:

Add a public function that allows a service to be added to the list of services that the web server holds. Make sure it is thread-safe and use the signature below.

```
public void AddService(WebService service)
```

We will use a simple definition of how to know whether or not a WebService object can handle a request: if the request-target/URI starts with the string specified by the WebService object's ServiceURI parameter, then it can process that request.

Implement the blocking Stop function:

Add a public, static function named "Stop" that has no parameters, to the WebServer class. Implement this as a blocking function that lets all threads in the thread pool finish the current task they're on, and then terminates all threads in the pool. If Stop is called when no threads are processing clients, then the call should return almost immediately, because shutting down all the idle threads will not take much time. If one or more threads is processing a client, it should finish that client's transaction in an orderly fashion, then terminate. Make sure that the "Stop" function waits for each thread in the thread pool to complete entirely before returning. It must also stop the listening thread.

Implement the DemoService Class

Write a class named DemoService in the CS422 namespace that inherits from the WebService class. All this service will do is write a response HTML string for every request. It will be very similar to the responses from the previous web server assignment, but now it's happening through a service and not hard-coded in to the server core code. In this and future assignments, implement all service classes in different code files than the web server core code. Remember that these services are things that change from app to app, as we write different web apps with different purposes.

Use the exact string below as a response template, and fill in the pieces with a string.Format call, using data from the request object. Each of the 4 pieces should be self-explanatory when you look at the template string.

```
private const string c_template =
        "<html>This is the response to the request:<br>" +
        "Method: {0}<br>Request-Target/URI: {1}<br>" +
        "Request body size, in bytes: {2}<br><br>" +
        "Student ID: {3}</html>";
```

Implement the demo service to make the response from the template, and then send it using the WebRequest.WriteHTMLResponse function. It will handle all requests sent to the server, so provide the service URI using the following code:

```
public override string ServiceURI {
    get
    {
        return "/";
    }
}
```