Tiny Web Server Take 2

honey the codewitch

9 Sep 2019 CPOL

A Tiny Dynamic Home Webserver in .NET that runs on Core and DNF both (Take 2)

Download source - 26.1 KB

Introduction

So my first crack at this was not up to par. I dramatically misunderstood where my app was blocking despite using all asynchronous I/O and was only serving synchronously, which is a real problem in a webserver, even one intended for home networks like this.

Speaking of which, don't make this internet facing. It's not scalable. It's intended for small apps and a limited number of connections, on something where you don't want the full ASP.NET stack to haul around, like on a standalone system where .NET Core runs but you don't have Apache on it. Or maybe you just want to make an HTML based application in your own winforms app so you want to serve to yourself and use the web browser control (though I don't recommend this these days). Maybe you want a little gadget to talk to your IoT gadgets and need to mess with headers, or whatever. Maybe you're just interested in stuff like this.

Background

HTTP is ugly, but it works. Most of what this project does is hack its way through headers and mangle responses to do chunking. The rest is just serving async, which is pretty easy in .NET, unless you muck it up like I did last time.

What we do is create a listening port and then accept on a pooled thread, where we handle the request. Right now, it's about half asynchronous on the I/O end and the threading takes care of any blocking that still happens, although there's a lot more async support in `SocketUtility` than is being used right now. I didn't want to complicate the source more than it is.

The code does minimal validation, and I didn't spend a lot of time making it robust. It's more of an example than anything.

Using the Code

`SocketUtility` is the foundation of all the HTTP and socket I/O, usually by way of extension methods on `Socket`.

Basically, much of the class is just HTTP protocol stuff, and asynchronous socket I/O, like the socket awaitable adapter that sort of munges the weird async socket API into something more async/await based (courtesy of MSDN, I didn't write that little adapter class - I use what's good! - link in the source).

It processes HTTP primarily using the ServeHttp() method, takes a listener socket that's already bound and *blocks* on it, so call it from another thread - preferably a thread pool. I've found that even using awaitable asynchronous methods, you still need the threads or it will block. I understand why, but only now. Basically, they're implemented by waiting the thread, and then waking up on a callback, or at least that's how it appears to run underneath the abstraction, but that's not exactly what I needed. So I just converted it to block, and set it on the `ThreadPool`. I suspect this is proper in any case, even if we were using some sort of `ServeHttpAsync()` method - which I have yet to write.

```
var listener = new Socket(SocketType.Stream, ProtocolType.Tcp);
var endPoint = new IPEndPoint(IPAddress.Any,8080));
listener.Bind(endPoint);
listener.Listen(10);
ThreadPool.QueueUserWorkItem((l) => {
       listener.ServeHttp((request, response) => {
               response.WriteLine("Hello World!");
           });
   }, listener);
   // execute wait here as the above doesn't block
```

This is basically what it looks like to set up a server. However, the `WebServer` component leverages `SocketUtility` to handle this for you.

All you have to do is set the properties and go. You'll note the surrounding code is more expansive than the actual webserving part. It's super simple. On WinForms, it's a component, so it can be present in the designer on a WinForm, and you can just set the two properties on it, wire up the event and go. See the TinyWebDemo.

```
static void Main(string[] args)
{
    var w3s = new WebServer();
    if (0 < args.Length)
    {
        w3s.EndPoint = _ToEndPoint(args[0]);
    }
    else
    {
        Console.Error.WriteLine("Usage: w3serv <ip>:<port>");
        Console.Error.WriteLine("\t<ip> can be \"*\"");
        return;
    }
    w3s.IsStarted = true;
    w3s.ProcessRequest += W3s_ProcessRequest;
    Console.Error.WriteLine("Press any key to stop serving...");
    Console.ReadKey();
    w3s.Dispose(); // shut down - in production you'd use the "using" directive or try/finally
}

private static void W3s_ProcessRequest(object sender, ProcessRequestEventArgs args)
{
    // default is text/plain
    args.Response.ContentType = "text/html";
    args.Response.WriteLine("<html><body><h1>Hello World</h1></body>");
}
```

It has no mechanism for serving files, only dynamic content, and it's your responsibility to handle the request path and serve the appropriate content. You also need to set the Content-Type. Currently, it doesn't buffer but it can be easily updated to be able to buffer the output like ASP.NET can. I just didn't bother to do it, and I like to stream anyway.

History

- 8th September, 2019 - Initial submission

License

About the Author

honey the codewitch
United States

Just a shiny lil monster. Casts spells in C++. Mostly harmless.

Comments and Discussions

35 messages have been posted for this article Visit **https://www.codeproject.com/Articles/5205722/Tiny-Web-Server-Take-2** to post and view comments on this article, or click **here** to get a print view with messages.