# Table of Contents

[Table of Contents 1](#_Toc416246221)

[Introduction 5](#_Toc416246222)

[Source Code 7](#_Toc416246223)

[Why Build a Web Server? 8](#_Toc416246224)

[Your First Web Server 10](#_Toc416246225)

[Writing a Web Server is Simple! 10](#_Toc416246226)

[Issues with localhost? 13](#_Toc416246227)

[Writing a Web Server is Complicated! 13](#_Toc416246228)

[We Need an Architecture 14](#_Toc416246229)

[Dynamic vs. Static Content and the Single-Page Paradigm 15](#_Toc416246230)

[But Do We Need All This Overhead? 15](#_Toc416246231)

[Threads, Tasks and async/await 17](#_Toc416246232)

[Multiple Listeners 17](#_Toc416246233)

[Test Results 19](#_Toc416246234)

[Why async/await Is Not the Right Solution 21](#_Toc416246235)

[Allocating Our Own Threads 21](#_Toc416246236)

[What About ThreadPool? 23](#_Toc416246237)

[Conclusion 24](#_Toc416246238)

[Single Thread Listener 24](#_Toc416246239)

[Conclusion 27](#_Toc416246240)

[Thread Spanning Workflows 29](#_Toc416246241)

[Workflow Continuation State 30](#_Toc416246242)

[Workflow Continuation 31](#_Toc416246243)

[Workflow Item 31](#_Toc416246244)

[Workflow Class 32](#_Toc416246245)

[Putting it All Together 33](#_Toc416246246)

[Exception Handling 38](#_Toc416246247)

[Context Extension Methods 40](#_Toc416246248)

[Routing 42](#_Toc416246249)

[A Routing Entry 43](#_Toc416246250)

[A Route Key 43](#_Toc416246251)

[A Route Table 44](#_Toc416246252)

[The Route Handler 46](#_Toc416246253)

[Try it Out 47](#_Toc416246254)

[Conclusion 47](#_Toc416246255)

[Sessions 49](#_Toc416246256)

[Session 49](#_Toc416246257)

[Session Manager 51](#_Toc416246258)

[CSRF Token 53](#_Toc416246259)

[Try it Out 53](#_Toc416246260)

[Automatically Cleaning Up Expired Sessions 56](#_Toc416246261)

[Conclusion 57](#_Toc416246262)

[HTTPS 58](#_Toc416246263)

[Domain Validation 58](#_Toc416246264)

[Organization Validation 58](#_Toc416246265)

[Extended Validation 58](#_Toc416246266)

[How to Make a Domain Level Certificate 58](#_Toc416246267)

[Make the Certificate Trusted 59](#_Toc416246268)

[Add the “Certificates” Snap In 60](#_Toc416246269)

[Verify Certificate Creation 61](#_Toc416246270)

[Get the Certificate Thumbprint 61](#_Toc416246271)

[Copy the Certificate to the Trusted Root Certification Authorities Folder 62](#_Toc416246272)

[Verify the Certificate is now Trusted 64](#_Toc416246273)

[Bind the Certificate to all IP addresses and Port on the Machine 64](#_Toc416246274)

[That’s All! 64](#_Toc416246275)

[Enabling Your Web Server to Receive Port 443 Requests 65](#_Toc416246276)

[Error Handling 66](#_Toc416246277)

[Parameterized Routes 67](#_Toc416246278)

[Agreeing on a Syntax 68](#_Toc416246279)

[Handling ID’s 68](#_Toc416246280)

[Test It Out! 70](#_Toc416246281)

[Conclusion 71](#_Toc416246282)

[Form Parameters 72](#_Toc416246283)

[AJAX 73](#_Toc416246284)

[View Engines 74](#_Toc416246285)

[CSRF 75](#_Toc416246286)

[Other Things I Need to Talk About 76](#_Toc416246287)

[A Simple Website Example 78](#_Toc416246288)

# Introduction

In 1990, Tim Berners-Lee wrote the first web server, known as CERN httpd, and the first browser, which he called WorldWideWeb. Imagine for a moment a world without the World Wide Web. Then on Christmas Day, 1990[[1]](#footnote-1), the first ever web server goes live. Imagine that the only way to view this web, such as it was with only one server was with the browser program WorldWideWeb[[2]](#footnote-2).

The motivation for the World Wide Web (not to be confused with the first browser, the WorldWideWeb – no spaces) dates back even further, to 1980, when Berners-Lee was working at CERN[[3]](#footnote-3). Here, where approximately 10,000 people were working, the exchange of information between numerous and disparate system was nearly impossible[[4]](#footnote-4). To address this, Berners-Lee wrote a software project called ENQUIRE which was a simple hypertext program that was similar to Apple’s HyperCard but with advantage that it was portable and ran on different systems. ENQUIRE was sort of like a modern-day wiki.

However, management of the content of ENQUIRE was restricted to its user – it was the user’s responsibility to keep the information up to date, which became a time consuming process for the user. In 1984, Berners-Lee realized that a different system, one that was accessible to everybody and that allowed people to create content independently of others was necessary. Furthermore, a person could link to content created by other people without having to update the linked content. This linkage could be thought of as a “web.”

In 1989, Berners-Lee proposed an Internet-based hypertext system known as HTML[[5]](#footnote-5), consisting of 18 elements which were strongly influenced by the Standard Generalized Markup Language (SGML) documentation format at CERN. It is interesting to note that eleven of those elements still exist in HTML 4.

Twenty five or so years later, the World Wide Web has become ubiquitous, living on our computers, phones, entertainment boxes, and vending machines. In April 2014, web servers were responsible for serving the content of almost *one billion* websites[[6]](#footnote-6). In a list of uses for hypertext in 1990, Berners-Lee put an encyclopedia as the first entry in that list. Today, the World Wide Web has become much more dynamic (and some would say invasive) with constant notifications of social media, email, and calendar events, and much more an entertainment (as opposed to a research) tool with the advent of chat rooms, YouTube, Netflix, and so forth.

All of this media richness has grown from that initial vision by Tim Berners-Lee, and is dependent upon that thing we call a “web server.” Today’s web server is much more sophisticated than the essentially static file system content server of the early 1990’s. Today’s web servers support a variety of capabilities such as user authentication, secure and encrypted data transport, server-side scripting to generate dynamic content, virtual hosting for serving many web sites from one IP address, and bandwidth throttling in order to be able to serve more clients[[7]](#footnote-7). Furthermore, web servers exist on many devices, including routers, printers and even cameras, and may be localized to just an intranet, having no exposure to the rest of the web.

Furthermore, we no longer simply develop static content that others can reference with hyperlinks in their own static content “pages.” Today, there are whole technology “stacks” that are necessary to know in order to develop web *applications­­* -- websites (or “web-apps”) that accomplish what would in the past have been implemented as a desktop application. We now can write (and share in real time) documents, make appointments in a calendar, balance our checkbook in a spreadsheet, and even put together our marketing presentation, all using “web-apps.”

As a result, the concept of a “web server” has become fuzzy because the server is now entwined with the dynamic requirements of the web application. Handling a request is no longer the simple process of “send back the content of this file” but instead involves routing the request to the web application, which, among other things, determines where the content comes from (a database, a file, a stock ticker service, etc.). Furthermore, while the web application can determine whether the user is authorized to view the content, this requirement can be separated into the imperative code (living on the web server) that does the authorization check, and the declarative code (living in the web application) that defines what routes require authorization. In a more complex authorization model involving roles and authorization levels, the web server might query the web application for authorization as a completely independent function from the handling of the route’s content. Issues of authentication and session management are also fuzzy – what functionality does the web server provide vs. the web application, and how do the two interact?

Because this line between the web server and the web application is fuzzy, and because often enough the answer is “it depends on the requirements”, we arrive at an initial answer to the question “why build a web server?” This question does however require even further inquiry.

# Source Code

The source code for this book can be found here:

<https://github.com/cliftonm/WebServersSuccinctly>

Clone the repo from the command line with:

|  |
| --- |
| git clone <https://github.com/cliftonm/WebServersSuccinctly.git> |

The code examples in this book are not necessarily found in the repo, as they are either intermediate steps in building the server or represent test code.

## About the Code in This Book

The writing style that I often use for my articles is one of:

1. Requirements
2. Design
3. Research / Implement
4. Implement / Analyze
5. Refactor

Those five steps are more or less always repeated. As a result, the code examples in this book don’t precisely match the final code in the GitHub because this book essentially walks you through the development process, the questions that I had and researched, and the refactoring that was necessary as I went along and added additional requirements. You’ll note, for example, a significant refactoring that took place at the start of the chapter on view engines. I hope that this process is as interesting to you as it is to me, as I feel it gives the reader the entire gestalt of the software development process, not just the (supposedly) finished, polished version.

## Where are the Unit Tests?

You will also notice a glaring omission of unit tests, which is ironic because I wrote an entire e-book for SyncFusion on unit testing. The reason they are missing from the book is simply one of space constraint. The reason they are missing from the code repository is because there are actually very few pieces of code in this implementation that are complex enough to warrant unit testing. In fact, in my opinion, there are none. As it odd as it may sound, there are actually less than 400 lines of code in the core web server assembly and almost all classes have a cyclometric complexity under the threshold of 25, which according the Microsoft would represent a “violation”[[8]](#footnote-8), and certainly no function exceeds 8:

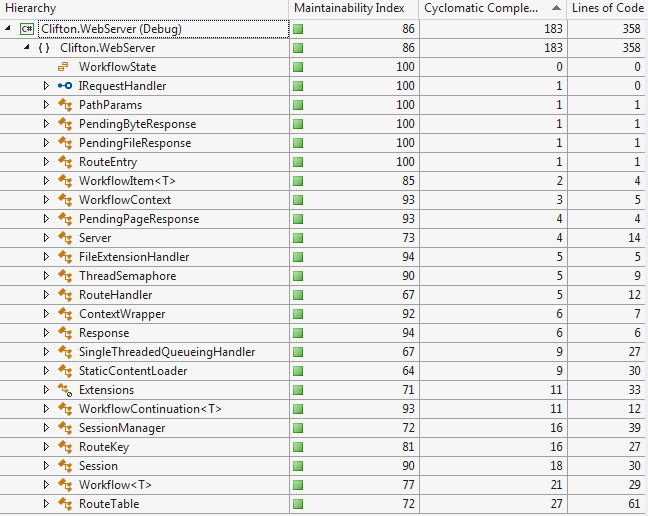


Figure 1: Core Assembly Analysis

So as ironic as it may seem, there really isn’t anything here worth the time and trouble to unit test, which is one of the qualifiers that I wrote about in by book Unit Testing Succinctly.

# Why Build a Web Server?

Modern web application development frameworks such as ASP.NET and its three flavors (Web Forms, MVC, Web Pages) and Ruby on Rails, sit firmly between the web server (typically IIS, Apache, or nginx, to name three) and you, the web application builder.

Rails says this about itself:

“Rails is a web application development framework…designed to make programming web applications easier by making assumptions about what every developer needs to get started. It allows you to write less code while accomplishing more than many other languages and frameworks…Rails is opinionated software. It makes the assumption that there is the "best" way to do things, and it's designed to encourage that way - and in some cases to discourage alternatives.”[[9]](#footnote-9)

Microsoft says this about ASP.NET:

“ASP.NET is a unified Web development model that includes the services necessary for you to build enterprise-class Web applications with a minimum of coding.”[[10]](#footnote-10)

And, with regards to ASP.NET MVC:

“ASP.NET MVC targets developers who are interested in patterns and principles like test-driven development, separation of concerns, inversion of control (IoC), and dependency injection (DI). This framework encourages separating the business logic layer of a web application from its presentation layer.

By dividing the application into the model (M), views (V), and controllers (C), ASP.NET MVC can make it easier to manage complexity in larger applications. With ASP.NET MVC, you can have multiple teams working on a web site because the code for the business logic is separate from the code and markup for the presentation layer — developers can work on the business logic while designers work on the markup and JavaScript that is sent to the browser.”

To complicate matters, one also can choose from a variety of view engines: a server-side markup language and processing tool for creating dynamic web pages. For example, Razor[[11]](#footnote-11), introduced in 2010[[12]](#footnote-12), is perhaps the in vogue view engine that can be used in conjunction with ASP.NET MVC (at least at the time of this writing.) Rails comes with its own view engine as well and supports other view engines, such as Slim[[13]](#footnote-13), one of over 24 different template engine offerings[[14]](#footnote-14).

Furthermore, these frameworks come with their own ideas of how you should interface with a database, in other words, there is a strong push towards using an Object Relational Mapper. In ASP.NET MVC, the preferred ORM is Entity Framework. With Rails, the ORM is implemented with Active Record.

So what does all of this have to do with a web server? Because, to write a web application, you typically have to buy 100% into the, as Rails puts it, “opinionated” framework’s idea of how you should do things: MVC, ORM, view engine. If you don’t want to buy into this approach (and there are often good reasons not to), you have several alternatives. One such alternative is to use PHP[[15]](#footnote-15), a server-side scripting language designed for web development, along with a web server like IIS or Apache. PHP is simply a scripting language, it doesn’t enforce a particular architecture (like MVC) or have an opinion about the database connectivity framework.

# Your First Web Server

## Writing a Web Server is Simple!

Writing a web server is essentially rather simple.  If all we wanted to do is serve up some HTML pages, we could be done with this implementation:

Namespaces we need to use:

|  |
| --- |
| using System; using System.IO; using System.Net; using System.Text; using System.Threading; using System.Threading.Tasks; |

A couple helpful extension methods:

|  |
| --- |
| /// <summary>  /// Some useful string extensions.  /// </summary>  public static class ExtensionMethods  {  /// <summary>  /// Return everything to the left of the first occurrence of the specified string,  /// or the entire source string.  /// </summary>  public static string LeftOf(this String src, string s)  {  string ret = src;  int idx = src.IndexOf(s);  if (idx != -1)  {  ret = src.Substring(0, idx);  }  return ret;  }  /// <summary>  /// Return everything to the right of the first occurrence of the specified string,  /// or an empty string.  /// </summary>  public static string RightOf(this String src, string s)  {  string ret = String.Empty;  int idx = src.IndexOf(s);  if (idx != -1)  {  ret = src.Substring(idx + s.Length);  }  return ret;  } |

And the program itself:

|  |
| --- |
| class Program  {  static Semaphore sem;  static void Main(string[] args)  {  // Supports 20 simultaneous connections.  sem = new Semaphore(20, 20);  HttpListener listener = new HttpListener();  string url = "http://localhost/";  listener.Prefixes.Add(url);  // listener.Prefixes.Add("http://172.16.16.115/");  listener.Start();  Task.Run(() =>  {  while (true)  {  sem.WaitOne();  StartConnectionListener(listener);  }  });  Console.WriteLine("Press a key to exit the server.");  Console.ReadLine();  }  /// <summary>  /// Await connections.  /// </summary>  static async void StartConnectionListener(HttpListener listener)  {  // Wait for a connection. Return to caller while we wait.  HttpListenerContext context = await listener.GetContextAsync();  // Release the semaphore so that another listener can be immediately started up.  sem.Release();  // Get the request.  HttpListenerRequest request = context.Request;  HttpListenerResponse response = context.Response;  // Get the path, everything up to the first ? and excluding the leading "/"  string path = request.RawUrl.LeftOf("?").RightOf("/");  Console.WriteLine(path); // Nice to see some feedback.    // Load the file and respond with a UTF8 encoded version of it.  string text = File.ReadAllText(path);  byte[] data = Encoding.UTF8.GetBytes(text);  response.ContentType = "text/html";  response.ContentLength64 = data.Length;  response.OutputStream.Write(data, 0, data.Length);  response.ContentEncoding = Encoding.UTF8;  response.StatusCode = 200; // OK  response.OutputStream.Close();  }  } |

That's it for the C# code. There’s two more things we need to do:

First, create a Hello World HTML file:

|  |
| --- |
| <p>Hello World</p> |

Put it into the bin\Debug folder of a console app:

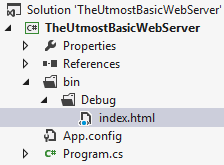


Figure 2: Solution Tree

Second, put an icon file name “favicon.ico” into the bin\Debug folder as well, otherwise, if the browser requests it, the web server will throw a File Not Found exception.

Now, when you run the console app, it will wait for a connection.  Fire up your browser and for the URL, enter:

|  |
| --- |
| <http://localhost/index.html> |

In the console window, you'll see the path emitted, and in the browser, you'll see the page rendered:

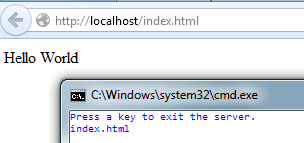


Figure 3: Serving Static Content

### Issues with localhost?

If your browser is having problems connecting to localhost, edit your C:\Windows\System32\drivers\etc\hosts file and make sure there is an entry that looks like this:

127.0.0.1 localhost

If it's missing, add it, save the file, and reboot the computer.

## Writing a Web Server is Complicated!

What we wrote above is a static HTML page server, and there's a lot wrong with it:

* Only HTML MIME type is supported (your browser is rather forgiving, if you get the content type wrong, most of the time it will accommodate the error).  Other MIME types[[16]](#footnote-16) include CSS, JavaScript, and of course media, such as images.
* It doesn't handle the common HTTP methods[[17]](#footnote-17), namely GET, POST, PUT and DELETE.
* We have no exception handling.
* There's no support for Cross-Site Request Forgery[[18]](#footnote-18) (CSRF) tokens.
* The server has no concept of session.
* The server doesn't support https.  SSL/TLS support is critically important in today's world.
* Some sort of HTML processing engine would be very useful to resolve connection-specific content on the server before the page is sent to the browser.
* There is no support for routing requests to, say, a Model-View-Controller[[19]](#footnote-19) (MVC) or Model-View-ViewModel[[20]](#footnote-20) (MVVM) architecture.
* Our server implementation above is entangled with the application-specific HTML pages.  We need to decouple it -- basically, make it an assembly that our application-specific stuff references.
* What about master pages?
* What about authorization, authentication, session expiration?
* What about model support?
* What about integration testing?

Request routing combined with some sort of a controller implementation is really useful when implement a REST[[21]](#footnote-21) API, which our web server should be able to do as well.  And REST is at the center of AJAX[[22]](#footnote-22) / AJAJ[[23]](#footnote-23) requests (SOAP[[24]](#footnote-24) being another common protocol, but REST is much more in vogue nowadays), allowing us to write single-page applications.  Here we are implicitly entering into the realm of serving dynamic content.  If you're rendering mostly static content, then you could also look at Apache (especially in conjunction with PHP) or Nginx, both of which are primarily static content web servers but with support for dynamic content[[25]](#footnote-25).

## We Need an Architecture

If you look at a few popular web servers, such as ASP.NET[[26]](#footnote-26), Ruby on Rails[[27]](#footnote-27) or NancyFx[[28]](#footnote-28), you'll immediately get a sense that there is a sophisticated architecture supporting the web server along with some sophisticated built-in functionality that really doesn't have anything to do with handling requests but that tends to make the job easier because there's a typical set of common tasks people need to perform when creating a professional website.  So you will almost immediately notice one or more of the following:

1. Either enforces or at least defaults to creating a project with an MVC architecture
2. Has some sort of a view engine, such as ASP.NET's Razor[[29]](#footnote-29) or ASPX view engines, or NancyFx's SuperSimpleViewEngine[[30]](#footnote-30).  Rails supports a wide range of view (aka "template) engines[[31]](#footnote-31).
3. Possibly includes some sort of Object-Relational Mapper (ORM).  In the ASP.NET world, this is usually Entity Framework[[32]](#footnote-32), in Rails we find ActiveRecord[[33]](#footnote-33).

Underlying these three common features of popular web servers are three very important premises:

1. You will almost always be rendering dynamic content
2. The dynamic content will be determined in large part from data in a database
3. The Model-View-Controller paradigm is the architectural glue that you are going to use for interactions between the user interface and the database.

I mentioned this earlier as well in conjunction with REST / AJAX, and the distinction between dynamic and static content web servers should not be so quickly dismissed.

## Dynamic vs. Static Content and the Single-Page Paradigm

TODO: Footnotes on things I mention here

The trend (especially as "push servers", see SignalR) is to move toward single-page applications (SPAs) -- the content of the page updates without requiring a full page refresh. A full page refresh requires a callback to the server to load all the content, whereas an SPA requests only the content that it needs.  This makes developing a web application more complicated because you're not just rendering the page on the server, you're coding in JavaScript on the client-side to implement the dynamic behavior, and probably using additional JavaScript packages such as jQuery, Knockout, Backbone, Angular, or any number of additional options "out there" (you'll note I'm completely ignoring node.js.)  Furthermore, you’re not just writing “render this page” server/client side code. Instead, a significant portion of what you write on the server will look more like an API to support AJAX/REST callbacks to return the content the client is requesting. In fact, it probably is helpful to think more in terms of writing an API than in terms of writing a website!

## But Do We Need All This Overhead?

The simple answer is, no.  The whole reason this author has even bothered to write yet another web server from scratch is because those features, which are often integrated with the basic process of a web server and initialized in a new project template, are, while not altogether unnecessary, sometimes better served by a lightweight version of the feature.  The question often comes up whether to roll your own or buy into an existing architecture, and the deeper question, why are we always rewriting prior work?  The answer to both, and the premise of why you're reading this book (other than to learn about the internals of how web servers work) is that, based on the experiences of working with other technologies, you have discovered that your needs are not being met by the existing solutions.  The typical answer "because the existing technology can be improved upon" is actually a weak argument, especially when one considers that any new technology will have deficiencies in areas other than the technology that it replaces.  So, this author's motivations are to write a web server that not only meets his needs but also employs an architecture that does not hinder you from meeting *your* needs.  And the premise of such architecture is that the function of a web server should be completely decoupled from paradigms such as MVC as well as view engine and ORM implementations.  These should be in the purview of, if not the application, then at least some middle-tier that you can take or leave depending on your needs.

# Threads, Tasks and async/await

In order to begin looking at our architecture, we really need to take a deep dive into the issues of threading. Along the way, we’ll discover some surprising things.

There are two basic options for how to handle incoming requests:

* Multiple listeners: We create multiple listeners and process the request on the thread allocated to the continuation of the awaited GetContextAsync call. Because there is not a Windows Form, the continuation is free to allocate its own thread as opposed to the Windows application behavior, which marshals onto the main application thread.
* Single listener: A single thread listens for incoming connections and immediately queues that request so that it can go back to listening for the next connection request. A separate thread (or threads) processes the requests.

## Multiple Listeners

Let's look at instrumenting the StartConnectionListener function so that we can get a sense of the processing times and threads.  First, a couple basic instrumentation functions:

|  |
| --- |
| protected static DateTime timestampStart;  static public void TimeStampStart()  {  timestampStart = DateTime.Now;  }  static public void TimeStamp(string msg)  {  long elapsed = (long)(DateTime.Now - timestampStart).TotalMilliseconds;  Console.WriteLine("{0} : {1}", elapsed, msg);  } |

Next, we add the instrumentation to the StartConnectionListener:

|  |
| --- |
| /// <summary>  /// Await connections.  /// </summary>  static async void StartConnectionListener(HttpListener listener)  {  TimeStamp("StartConnectionListener Thread ID: " + Thread.CurrentThread.ManagedThreadId);  // Wait for a connection. Return to caller while we wait.  HttpListenerContext context = await listener.GetContextAsync();  // Release the semaphore so that another listener can be immediately started up.  sem.Release();  handler.Process(context);  } |

Recall that these listeners are all initialized on a separate thread, but as noted above, we let the .NET framework allocate a thread on the continuation.

|  |
| --- |
| Task.Run(() =>  {  while (true)  {  sem.WaitOne();  StartConnectionListener(listener);  }  }); |

For this test, I've created a ListenerThreadHandler class:

|  |
| --- |
| public class ListenerThreadHandler : CommonHandler, IRequestHandler  {  public void Process(HttpListenerContext context)  {  Program.TimeStamp("Process Thread ID: " + Thread.CurrentThread.ManagedThreadId);  CommonResponse(context);  }  } |

Where CommonResponse artificially injects a 1 second delay to simulate some complex process before issuing the response:

|  |
| --- |
| public void CommonResponse(HttpListenerContext context)  {  // Artificial delay.  Thread.Sleep(1000);  // Get the request.  HttpListenerRequest request = context.Request;  HttpListenerResponse response = context.Response;  // Get the path, everything up to the first ? and excluding the leading "/"  string path = request.RawUrl.LeftOf("?").RightOf("/");  // Load the file and respond with a UTF8 encoded version of it.  string text = File.ReadAllText(path);  byte[] data = Encoding.UTF8.GetBytes(text);  response.ContentType = "text/html";  response.ContentLength64 = data.Length;  response.OutputStream.Write(data, 0, data.Length);  response.ContentEncoding = Encoding.UTF8;  response.StatusCode = 200; // OK  response.OutputStream.Close();  } |

Now, we'll test how the server responds to 10 effectively simultaneous requests:

|  |
| --- |
| for (int i = 0; i < 10; i++)  {  Console.WriteLine("Request #" + i);  MakeRequest(i);  } |

and:

|  |
| --- |
| /// <summary>  /// Issue GET request to localhost/index.html  /// </summary>  static async void MakeRequest(int i)  {  TimeStamp("MakeRequest " + i + " start, Thread ID: " + Thread.CurrentThread.ManagedThreadId);  string ret = await RequestIssuer.HttpGet("http://localhost/index.html");  TimeStamp("MakeRequest " + i + " end, Thread ID: " + Thread.CurrentThread.ManagedThreadId);  } |

In the above code, once an asynchronous function blocks, the await will return to the caller and the next MakeRequest is issued.  When the asynchronous function completes, MakeRequest continues.

### Test Results

What we're interested in is:

* When was the request issued?
* How long did it take to complete?
* Was the continuation on the same thread as the request call, or a different thread?

In the trace log, we first see all the MakeRequest function calls, all on the same thread, which is expected as they're all being issued by the same Task:

|  |
| --- |
| Request #0  3 : MakeRequest 0 start, Thread ID: 1  Request #1  55 : MakeRequest 1 start, Thread ID: 1  Request #2  57 : MakeRequest 2 start, Thread ID: 1  Request #3  58 : MakeRequest 3 start, Thread ID: 1  Request #4  59 : MakeRequest 4 start, Thread ID: 1  Request #5  61 : MakeRequest 5 start, Thread ID: 1  Request #6  62 : MakeRequest 6 start, Thread ID: 1  Request #7  63 : MakeRequest 7 start, Thread ID: 1  Request #8  63 : MakeRequest 8 start, Thread ID: 1  Request #9  63 : MakeRequest 9 start, Thread ID: 1 |

Next, we see the Process messages coming in as well as the MakeRequest "end" calls (I'm omitting the StartConnectionListener and MakeRequest messages for clarity):

|  |
| --- |
| 78 : Process Thread ID: 11 79 : Process Thread ID: 5 80 : Process Thread ID: 9 81 : Process Thread ID: 10   783 : Process Thread ID: 12   1080 : Process Thread ID: 11 1084 : Process Thread ID: 5 1091 : Process Thread ID: 9 1106 : Process Thread ID: 10   1315 : Process Thread ID: 13   1789 : MakeRequest 7 end, Thread ID: 12 |

What's revealing here is that:

* The requests appear to be processed in batches of 4 (the computer I'm testing on has 4 cores).
* Threads are being re-used.
* The continuation is not happening on the same thread (we expect that because this is a console application and we haven't defined a continuation context.)
* Because only "roughly" 4 threads are active at once, the whole process takes about 2.3 seconds to complete (odd how 10 requests / 4 threads is 2.5)

Conversely, observe what happens on an 8 core system:

|  |
| --- |
| 38 : Process Thread ID: 15  38 : Process Thread ID: 13  38 : Process Thread ID: 5  38 : Process Thread ID: 16  39 : Process Thread ID: 17  39 : Process Thread ID: 14  40 : Process Thread ID: 19  41 : Process Thread ID: 18  782 : Process Thread ID: 20  1039 : Process Thread ID: 15 |

Now we see 8 requests being processed simultaneously, and the last two occurring later.  What's going on?

### Why async/await Is Not the Right Solution

From the above trace, we can surmise that the thread being allocated for the continuation is allocated based on the number of CPU cores. This is really not the behavior we want. Many requests will involve file I/O, interacting with the database, contacting social media, and so forth, all of which are processes where the thread will be blocked waiting for a response. We certainly don’t want to delay the processing of other incoming requests simply because the mechanism for allocating the continuation thread thinks it should be based on available cores. Unfortunately, this mechanism seems to be in the bowels of how continuations are handled. It is not controllable through TaskCreationOptions because we’re dealing with how the continuation of the awaited call is being handled. All we can declare here is that this is not the implementation we want.

## Allocating Our Own Threads

What happens when we allocate the threads ourselves?  Let's give that a try.  First, we change the way the context listener threads are initialized:

|  |
| --- |
| for (int i = 0; i < 20; i++)  {  Thread thread = new Thread(new ParameterizedThreadStart(WaitForConnection));  thread.IsBackground = true;  thread.Start(listener);  } |

Then, instead of using async/await and semaphores, each thread blocks until a connection is received:

|  |
| --- |
| /// <summary>  /// Block until a connection is received.  /// </summary>  static void WaitForConnection(object objListener)  {  HttpListener listener = (HttpListener)objListener;  while (true)  {  TimeStamp("StartConnectionListener Thread ID: " + Thread.CurrentThread.ManagedThreadId);  HttpListenerContext context = listener.GetContext();  handler.Process(context);  }  } |

Now, when our requests are issued, we see immediately that they are processed by 10 unique threads:

|  |
| --- |
| 75 : Process Thread ID: 3  75 : Process Thread ID: 9  75 : Process Thread ID: 4  75 : Process Thread ID: 5  76 : Process Thread ID: 8  75 : Process Thread ID: 10  76 : Process Thread ID: 7  76 : Process Thread ID: 6  76 : Process Thread ID: 11  76 : Process Thread ID: 12 |

And we also see that the responses are all in the same "one second later" block of time:

|  |
| --- |
| 1083 : MakeRequest 4 end, Thread ID: 31  1090 : MakeRequest 2 end, Thread ID: 31  1098 : MakeRequest 3 end, Thread ID: 31  1097 : MakeRequest 1 end, Thread ID: 28  1104 : MakeRequest 0 end, Thread ID: 32  1091 : MakeRequest 8 end, Thread ID: 29  1113 : MakeRequest 6 end, Thread ID: 29  1088 : MakeRequest 5 end, Thread ID: 30  1119 : MakeRequest 7 end, Thread ID: 32  1121 : MakeRequest 9 end, Thread ID: 29 |

This unequivocally shows us that using async/await is not the right implementation choice!

## What About ThreadPool?

But is the problem with async/await or the system ThreadPool? This is not an ideal situation because at the moment we’re implementing long running threads, but we’ll try it regardless:

|  |
| --- |
| For (int i = 0; i < 20; i++)  {  ThreadPool.QueueUserWorkItem(WaitForConnection, listener);  } |

Look at what happens to the initialization process:

|  |
| --- |
| 781 : StartConnectionListener Thread ID: 7  1313 : StartConnectionListener Thread ID: 8  1845 : StartConnectionListener Thread ID: 9  2377 : StartConnectionListener Thread ID: 10  2909 : StartConnectionListener Thread ID: 11  3441 : StartConnectionListener Thread ID: 12  3973 : StartConnectionListener Thread ID: 13  4505 : StartConnectionListener Thread ID: 14  5037 : StartConnectionListener Thread ID: 15  5569 : StartConnectionListener Thread ID: 16  6100 : StartConnectionListener Thread ID: 17 |

We certainly experience what the MSDN documentation says regarding ThreadPool[[34]](#footnote-34): “As part of its thread management strategy, the thread pool delays before creating threads. Therefore, when a number of tasks are queued in a short period of time, there can be a significant delay before all the tasks are started.”

Fortunately though, once the threads have been initialized, we see that tha the processing happens simultaneously:

|  |
| --- |
| 12121 : Process Thread ID: 4  12123 : Process Thread ID: 5  12125 : Process Thread ID: 6  12125 : Process Thread ID: 3  12127 : Process Thread ID: 7  12127 : Process Thread ID: 10  12127 : Process Thread ID: 11  12128 : Process Thread ID: 9  12128 : Process Thread ID: 12  12128 : Process Thread ID: 8 |

So, while it works, ThreadPool’s are also not the correct solution. And as the documentation indicates, a thread pool is not the right solution here because we’re creating a number of threads in a very short time, and these threads will run perpetually for the life of the server, furthermore, the threads will potentially block for long periods of timing waiting for connection requests.

### Conclusion

It is now very clear that we should not use async/await to implement asynchronous connection requests.  Async/await limits you to running processes based on the number of cores, preventing you (and the CPU) from distributing request processing across more threads than you have cores. This is a very reasonable thing to do when you consider that many processes, such as querying a database or third party social media API, will be for the most part waiting for a response.

## Single Thread Listener

Besides having determined that we need to use threads rather than the Task async/await mechanism, we also should consider whether we want multiple threads listening for requests or a single thread. With a single thread, one and only one thread is listening for incoming requests.  As soon as a request is received, the request is placed into a queue and the thread immediately waits for the next request.  In a separate thread, requests are dequeued and enqueued into a worker thread.  We can implement different algorithms for determining which worker thread to enqueue the request, but in the implementation below we use a simple round-robin algorthm.

We’ll begin with a helper class that allows us to create a queue for each thread and a semaphore for signaling the thread:

|  |
| --- |
| /// <summary>  /// Track the semaphore and context queue associated with a worker thread.  /// </summary>  public class ThreadSemaphore  {  public int QueueCount { get { return requests.Count; } }  protected Semaphore sem;  protected ConcurrentQueue<HttpListenerContext> requests;  public ThreadSemaphore()  {  sem = new Semaphore(0, Int32.MaxValue);  requests = new ConcurrentQueue<HttpListenerContext>();  }  /// <summary>  /// Enqueue a request context and release the semaphore that  /// a thread is waiting on.  /// </summary>  public void Enqueue(HttpListenerContext context)  {  requests.Enqueue(context);  sem.Release();  }  /// <summary>  /// Wait for the semaphore to be released.  /// </summary>  public void WaitOne()  {  sem.WaitOne();  }  /// <summary>  /// Dequeue a request.  /// </summary>  public bool TryDequeue(out HttpListenerContext context)  {  return requests.TryDequeue(out context);  }  } |

Our handler, instead of processing the request immediately, queues the request and returns. A separate thread dequeues the request and assigns it, round-robin, to a worker thread:

|  |
| --- |
| public class SingleThreadedQueueingHandler : CommonHandler, IRequestHandler  {  protected ConcurrentQueue<HttpListenerContext> requests;  protected Semaphore semQueue;  protected List<ThreadSemaphore> threadPool;  protected const int MAX\_WORKER\_THREADS = 20;  public SingleThreadedQueueingHandler()  {  threadPool = new List<ThreadSemaphore>();  requests = new ConcurrentQueue<HttpListenerContext>();  semQueue = new Semaphore(0, Int32.MaxValue);  StartThreads();  MonitorQueue();  }  protected void MonitorQueue()  {  Task.Run(() =>  {  int threadIdx = 0;  // Forever...  while (true)  {  // Wait until we have received a context.  semQueue.WaitOne();  HttpListenerContext context;  if (requests.TryDequeue(out context))  {  // In a round-robin manner, queue up the request on the current  // thread index then increment the index.  threadPool[threadIdx].Enqueue(context);  threadIdx = (threadIdx + 1) % MAX\_WORKER\_THREADS;  }  }  });  }  /// <summary>  /// Enqueue the received context rather than processing it.  /// </summary>  public void Process(HttpListenerContext context)  {  requests.Enqueue(context);  semQueue.Release();  }  /// <summary>  /// Start our worker threads.  /// </summary>  protected void StartThreads()  {  for (int i = 0; i < MAX\_WORKER\_THREADS; i++)  {  Thread thread = new Thread(new ParameterizedThreadStart(ProcessThread));  thread.IsBackground = true;  ThreadSemaphore ts = new ThreadSemaphore();  threadPool.Add(ts);  thread.Start(ts);  }  }  /// <summary>  /// As a thread, we wait until there's something to do.  /// </summary>  protected void ProcessThread(object state)  {  ThreadSemaphore ts = (ThreadSemaphore)state;  while (true)  {  ts.WaitOne();  HttpListenerContext context;  if (ts.TryDequeue(out context))  {  Program.TimeStamp("Processing on thread " + Thread.CurrentThread.ManagedThreadId);  CommonResponse(context);  }  }  }  } |

The result is what we should expect, namely that our 10 requests begin processing simultaneously and complete processing simultaneously.

|  |
| --- |
| 76 : Processing on thread 4  76 : Processing on thread 3  76 : Processing on thread 5  77 : Processing on thread 6  78 : Processing on thread 7  78 : Processing on thread 8  79 : Processing on thread 10  79 : Processing on thread 11  79 : Processing on thread 9  81 : Processing on thread 12    1086 : MakeRequest 0 end, Thread ID: 31  1086 : MakeRequest 8 end, Thread ID: 29  1093 : MakeRequest 1 end, Thread ID: 29  1094 : MakeRequest 2 end, Thread ID: 29  1102 : MakeRequest 7 end, Thread ID: 29  1102 : MakeRequest 9 end, Thread ID: 31  1109 : MakeRequest 3 end, Thread ID: 31  1110 : MakeRequest 4 end, Thread ID: 29  1111 : MakeRequest 6 end, Thread ID: 31  1113 : MakeRequest 5 end, Thread ID: 31 |

### Conclusion

The advantage of the single-threaded connection queueing approach is that it can consume thousands of requests very quickly, and those requests can then be queued onto a finite number of worker threads.  The multi-listener approach will stop accepting requests when the all worker threads become busy.  While in either implementation the client ends up waiting for its request to be serviced, the major advantage of the second approach is that you are not creating potentially thousands of threads to handle high volume periods.  In fact, the single thread listener approach could even be implemented to dynamically start allocating more threads as volume increases or even to spool up additional servers.  This approach is a much more flexible solution.

# Thread Spanning Workflows

Processing client requests almost always involves a series of steps, which may include one or more of the following (and undoubtedly other things not in the list):

* Whitelist Validation
* Blacklist Exclusion
* Logging
* Work Distribution
* Authorization
* Session Expiration
* Routing
* View Engine

It behooves us therefore to look at requests as sequential workflows and to implement this abstraction such that the tasks can span different threads. For example, in the single listener thread implementation in the preceding chapter, we actually have three thread areas:

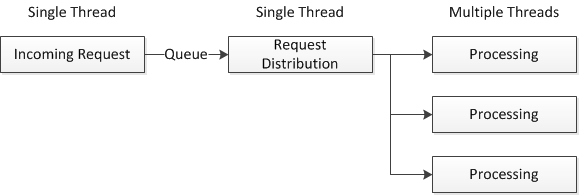


Figure 4: High Level Workflow

Inside each of these boxes, we might see something like this:

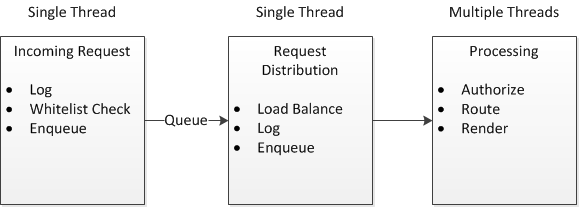


Figure 5: Low Level Workflow

What a thread-spanning workflow abstraction gains us is:

* define workflows declaratively
* decouple the thread from the work implementation
* allow the work implementation to determine how work should be continued:
  + on the same thread
  + deferred to another thread

The implementation requires that the “workflow continuation” be managed for every process as it sequences through the workflow steps, which is really the only “trick” to this implementation.

## Workflow Continuation State

Each workflow continuation can be in one of three states:

* Abort
* Continue
* Defer

|  |
| --- |
| /// <summary>  /// Workflow Continuation State  /// </summary>  public enum WorkflowContinuationState  {  /// <summary>  /// Terminate execution of the workflow.  /// </summary>  Abort,  /// <summary>  /// Continue with the execution of the workflow.  /// </summary>  Continue,  /// <summary>  /// Execution is deferred until Continue is called, usually by another thread.  /// </summary>  Defer,  } |

## Workflow Continuation

This class tracks the state of a workflow context and allows the workflow to continue when it is passed to another thread. What this does is:

1. We can define a single instance of a particular workflow pattern
2. We can use that instance simultaneously because we are effectively implementing Continuation Passing Style -- we are passing in the continuation state to each workflow function.
3. As a result, the workflow, as a process, is thread safe even though we are sharing instances amongst different threads.

|  |
| --- |
| /// <summary>  /// Thread-specific instance that preserves the workflow continuation context for that thread.  /// </summary>  public class WorkflowContinuation<T>  {  public int WorkflowStep { get; set; }  public bool Abort { get; set; }  public bool Defer { get; set; }  public Workflow<T> Workflow { get; protected set; }  public WorkflowContinuation(Workflow<T> workflow)  {  Workflow = workflow;  }  } |

## Workflow Item

A WorkflowItem is a lightweight container for the workflow function:

|  |
| --- |
| /// <summary>  /// A workflow item is a specific process to execute in the workflow.  /// </summary>  public class WorkflowItem<T>  {  protected Func<WorkflowContinuation<T>, T, WorkflowState> doWork;  /// <summary>  /// Instantiate a workflow item. We take a function that takes the   /// Workflow instance associated with this item  /// and a data item. We expect a WorkflowState to be returned.  /// </summary>  /// <param name="doWork"></param>  public WorkflowItem(Func<WorkflowContinuation<T>, T, WorkflowState> doWork)  {  this.doWork = doWork;  }  /// <summary>  /// Execute the workflow item method.  /// </summary>  public WorkflowState Execute(WorkflowContinuation<T> workflowContinuation, T data)  {  return doWork(workflowContinuation, data);  }  } |

## Workflow Class

Now that we have the pieces in place, we can see how a workflow is executed:

|  |
| --- |
| /// <summary>  /// The Workflow class handles a list of workflow items that we can use to  /// determine the processing of a request.  /// </summary>  public class Workflow<T>  {  protected List<WorkflowItem<T>> items;  public Workflow()  {  items = new List<WorkflowItem<T>>();  }  /// <summary>  /// Add a workflow item.  /// </summary>  public void AddItem(WorkflowItem<T> item)  {  items.Add(item);  }  /// <summary>  /// Execute the workflow from the beginning.  /// </summary>  public void Execute(T data)  {  WorkflowContinuation<T> continuation = new WorkflowContinuation<T>(this);  InternalContinue(continuation, data);  }  /// <summary>  /// Continue a deferred workflow, unless it is aborted.  /// </summary>  public void Continue(WorkflowContinuation<T> wc, T data)  {  // TODO: Throw exception instead?  if (!wc.Abort)  {  wc.Defer = false;  InternalContinue(wc, data);  }  }  /// <summary>  /// Internally, we execute workflow steps until:  /// 1. we reach the end of the workflow chain  /// 2. we are instructed to abort the workflow  /// 3. we are instructed to defer execution until later.  /// </summary>  protected void InternalContinue(WorkflowContinuation<T> wc, T data)  {  while ((wc.WorkflowStep < items.Count) && !wc.Abort && !wc.Defer)  {  WorkflowState state = items[wc.WorkflowStep++].Execute(wc, data);  switch (state)  {  case WorkflowState.Abort:  wc.Abort = true;  break;  case WorkflowState.Defer:  wc.Defer = true;  break;  }  }  }  } |

## Putting it All Together

As an example, I’ll illustrate a bit more of a robust website, capable of responding to different kinds of content requests. We’ll define a workflow that:

1. Logs the incoming IP address and web page request.
2. Checks that the requestor’s IP address is on our white list
3. Hands off the request to our single-threaded queue handler
4. Processes the requests, managing different file types.

The workflow is defined like this:

|  |
| --- |
| workflow = new Workflow<HttpListenerContext>();  workflow.AddItem(new WorkflowItem<HttpListenerContext>(LogIPAddress));  workflow.AddItem(new WorkflowItem<HttpListenerContext>(WhiteList));  workflow.AddItem(new WorkflowItem<HttpListenerContext>(handler.Process));  workflow.AddItem(new WorkflowItem<HttpListenerContext>(CommonHandler.StaticResponse)); |

And the logging and whitelist handler implementation is as follows:

|  |
| --- |
| /// <summary>  /// A workflow item, implementing a simple instrumentation of the  /// client IP address, port, and URL.  /// </summary>  static WorkflowState LogIPAddress(  WorkflowContinuation<HttpListenerContext> workflowContinuation,  HttpListenerContext context)  {  Console.WriteLine(context.Request.RemoteEndPoint.ToString() +   " : " + context.Request.RawUrl);  return WorkflowState.Continue;  }  /// <summary>  /// Only intranet IP addresses are allowed.  /// </summary>  static WorkflowState WhiteList(  WorkflowContinuation<HttpListenerContext> workflowContinuation,  HttpListenerContext context)  {  string url = context.Request.RemoteEndPoint.ToString();  bool valid = url.StartsWith("192.168") || url.StartsWith("127.0.0.1");  WorkflowState ret = valid ? WorkflowState.Continue : WorkflowState.Abort;  return ret;  } |

The actual response handler is implemented with a bit more intelligence – here we can specify the loader function to call based on the file extension in the request:

|  |
| --- |
| public static WorkflowState StaticResponse(  WorkflowContinuation<HttpListenerContext> workflowContinuation,  HttpListenerContext context)  {  // Get the request.  HttpListenerRequest request = context.Request;  HttpListenerResponse response = context.Response;  // Get the path, everything up to the first ? and excluding the leading "/"  string path = request.RawUrl.LeftOf("?").RightOf("/");  string ext = path.RightOfRightmostOf('.');  FileExtensionHandler extHandler;  if (extensionLoaderMap.TryGetValue(ext, out extHandler))  {  byte[] data = extHandler.Loader(context, path, ext);  response.ContentEncoding = Encoding.UTF8;  context.Response.ContentType = extHandler.ContentType;  context.Response.ContentLength64 = data.Length;  context.Response.OutputStream.Write(data, 0, data.Length);  response.StatusCode = 200; // OK  response.OutputStream.Close();  }  return WorkflowState.Continue;  } |

How the extension is routed to the static file loader handler is determined by the following mapping:

|  |
| --- |
| public static Dictionary<string, FileExtensionHandler> extensionLoaderMap =   new Dictionary<string, FileExtensionHandler>()  {  {"ico", new FileExtensionHandler()   {Loader=ImageLoader, ContentType="image/ico"}},  {"png", new FileExtensionHandler()   {Loader=ImageLoader, ContentType="image/png"}},  {"jpg", new FileExtensionHandler()   {Loader=ImageLoader, ContentType="image/jpg"}},  {"gif", new FileExtensionHandler()   {Loader=ImageLoader, ContentType="image/gif"}},  {"bmp", new FileExtensionHandler()   {Loader=ImageLoader, ContentType="image/bmp"}},  {"html", new FileExtensionHandler()   {Loader=PageLoader, ContentType="text/html"}},  {"css", new FileExtensionHandler()   {Loader=FileLoader, ContentType="text/css"}},  {"js", new FileExtensionHandler()   {Loader=FileLoader, ContentType="text/javascript"}},  {"json", new FileExtensionHandler()   {Loader=FileLoader, ContentType="text/json"}},  {"", new FileExtensionHandler()   {Loader=PageLoader, ContentType="text/html"}} }; |

And the three handlers are straightforward implementations – note how the page loader will append the extension “.html” if it is missing:

|  |
| --- |
| public static byte[] ImageLoader(  HttpListenerContext context,   string path,   string ext)  {  FileStream fStream = new FileStream(path, FileMode.Open, FileAccess.Read);  BinaryReader br = new BinaryReader(fStream);  byte[] data = br.ReadBytes((int)fStream.Length);  br.Close();  fStream.Close();  return data;  }  public static byte[] FileLoader(  HttpListenerContext context,   string path,   string ext)  {  string text = File.ReadAllText(path);  byte[] data = Encoding.UTF8.GetBytes(text);  return data;  }  public static byte[] PageLoader(  HttpListenerContext context,   string path,   string ext)  {  if (String.IsNullOrEmpty(ext))  {  path = path + ".html";  }  string text = File.ReadAllText(path);  byte[] data = Encoding.UTF8.GetBytes(text);  return data;  } |

Here we see the result of querying our server:

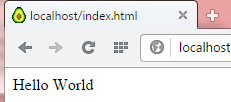


Figure 6: Result of a Workflow

Notice my cute little avocado icon is now rendering correctly!

# Exception Handling

Exception handling is a critical requirement of a web server – you don’t want your server crashing because of a poorly formatted request, a database error, and so forth. Besides an exception handler, we might as well take the opportunity to specify an “abort” handler in the workflow definition as well:

|  |
| --- |
| workflow = new Workflow<HttpListenerContext>(AbortHandler, OnException); |

Now our workflow continuation can call back to the abort and exception handlers:

|  |
| --- |
| protected void InternalContinue(WorkflowContinuation<T> wc, T data)  {  while ((wc.WorkflowStep < items.Count) && !wc.Abort && !wc.Defer)  {  try  {  WorkflowState state = items[wc.WorkflowStep++].Execute(wc, data);  switch (state)  {  case WorkflowState.Abort:  wc.Abort = true;  wc.Workflow.AbortHandler(data);  break;  case WorkflowState.Defer:  wc.Defer = true;  break;  }  }  catch (Exception ex)  {  // We need to protect ourselves from the user’s exception  // handler potentially throwing an exception.  try  {  wc.Workflow.ExceptionHandler(data, ex);  }  catch { }  wc.Done = true;  }  }  } |

And we can write a couple simple handlers – our abort handler simply terminates the connection whereas our exception handler returns the exception message.

|  |
| --- |
| static void AbortHandler(HttpListenerContext context)  {  HttpListenerResponse response = context.Response;  response.OutputStream.Close();  }  static void OnException(HttpListenerContext context, Exception ex)  {  HttpListenerResponse response = context.Response;  response.ContentEncoding = Encoding.UTF8;  context.Response.ContentType = "text/html";  byte[] data = Encoding.UTF8.GetBytes(ex.Message);  context.Response.ContentLength64 = data.Length;  context.Response.OutputStream.Write(data, 0, data.Length);  response.StatusCode = 200; // OK  response.OutputStream.Close();  } |

Now, for example, if I request a page whose corresponding file doesn’t exist, I get the exception message.

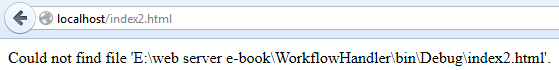


Figure 7: Error Handling Example

Of course, in real life, we probably want to redirect the user to the home page or a “page not found” page.

The salient point the above implementation is that, even if the specific workflow action doesn’t gracefully handle exceptions, the workflow engine itself manages the exception gracefully, giving your application options for notifying the user of the problem, and without bringing down the website.

# Context Extension Methods

Before going any further I need to introduce the extension methods that I’ve added to HttpListenerContext. You’ll see these extension methods used throughout the rest of this book:

|  |
| --- |
| public static class Extensions  {  /// <summary>  /// Return the URL path.  /// </summary>  public static string Path(this HttpListenerContext context)  {  return context.Request.RawUrl.LeftOf("?").RightOf("/").ToLower();  }  /// <summary>  /// Return the extension for the URL path's page.  /// </summary>  public static string Extension(this HttpListenerContext context)  {  return context.Path().RightOfRightmostOf('.').ToLower();  }  /// <summary>  /// Returns the verb of the request: GET, POST, PUT, DELETE, and so forth.  /// </summary>  public static string Verb(this HttpListenerContext context)  {  return context.Request.HttpMethod.ToUpper();  }  /// <summary>  /// Return the remote endpoint IP address.  /// </summary>  public static IPAddress EndpointAddress(this HttpListenerContext context)  {  return context.Request.RemoteEndPoint.Address;  }  /// <summary>  /// Returns a dictionary of the parameters on the URL.  /// </summary>  public static Dictionary<string, string> GetUrlParameters(  this HttpListenerContext context)  {  HttpListenerRequest request = context.Request;  string parms = request.RawUrl.RightOf("?");  Dictionary<string, string> kvParams = new Dictionary<string, string>();  parms.If(d => d.Length > 0,   (d) => d.Split('&').ForEach(keyValue =>   kvParams[keyValue.LeftOf('=').ToLower()] =   Uri.UnescapeDataString(keyValue.RightOf('='))));    return kvParams;  }  /// <summary>  /// Respond with an HTML string.  /// </summary>  public static void RespondWith(this HttpListenerContext context, string html)  {  byte[] data = Encoding.UTF8.GetBytes(html);  HttpListenerResponse response = context.Response;  response.ContentEncoding = Encoding.UTF8;  context.Response.ContentType = "text/html";  context.Response.ContentLength64 = data.Length;  context.Response.OutputStream.Write(data, 0, data.Length);  response.StatusCode = 200;  response.OutputStream.Close();  } } |

# Routing

It’s now time to talk about routing. The above examples are still for a static web server – we have no way of hooking into page requests, and equally importantly, doing different things based on the verb used in the request, which is vital for supporting AJAX and REST API’s.

Routing is also somewhat entangled with session state:

* Is the user authorized to view the page?
* Has the session expired?
* Does the user’s role give the user access to the page?

For example, in Ruby on Rails, authorization is often accomplished in the superclass of the controller whose methods are being invoked through a routing table. In ASP.NET MVC, whether the user must be authorized is determined by the “authorize” attribute decorating the controller method, again, the controller method is being invoked via a routing table. Role can also come into play as well other factors such as whether the session has expired or not.

Having worked with the above two approaches as well implementing specialized base class controllers such as “ExpirableController” and “AuthorizedRoleExpirableController”, the approach that I prefer decouples routing from session and authorization/role state and takes a more “functional programming” approach rather than object-oriented (Rails) or attribute decoration (MVC). This approach also works well with the workflow paradigm presented earlier and therefore has a nice consistent feel to it. But, without discussing the pros and cons of each approach, you should be getting a sense that there are places in a web server’s design that is really up to the designer and where you, as the “user” of the web server architecture, get very little say in those design decisions.

Happily, the workflow paradigm actually *does* give you considerable more say because you can actually implement your own routing and session state management. What’s provided here is an example, but if you wanted to use a more object-oriented approach or reflection to check the authorization requirement on a controller, you could certainly implement that.

However, the reason routing is entangled with authorization and session management is that, well, it makes sense. There are pages that are:

* Publically accessible
* Privately accessible with the right role
* Most, if not all private pages should be expirable

So, from a declarative perspective, it makes sense to define the constraints of a page (or a REST API endpoint) along with its route. What I’m proposing here as an implementation is to declaratively describe the routes and their constraints and implement the *process* of constraint checking and routing separately as opposed to an entangled implementation.

## A Routing Entry

For the reasons stated above, a route entry will consist of three “providers”:

1. SessionExpirationProvider
2. AuthorizationProvider
3. RoutingProvider

These providers are associated with whatever page / REST API path you want. For example:

You’ll note that the provider functions have the signature of a workflow process!

|  |
| --- |
| public class RouteEntry  {  public Func<WorkflowContinuation<HttpListenerContext>,   HttpListenerContext, Session, WorkflowState> SessionExpirationProvider;  public Func<WorkflowContinuation<HttpListenerContext>,   HttpListenerContext, Session, WorkflowState> AuthorizationProvider;  public Func<WorkflowContinuation<HttpListenerContext>,   HttpListenerContext, Session, WorkflowState> RoutingProvider;  } |

We’ll cover Session in the next chapter.

## A Route Key

We also need a route key – the verb and path:

|  |
| --- |
| /// <summary>  /// A structure consisting of the verb and path, suitable as a key for the route table entry.  /// Key verbs are always converted to uppercase, paths are always converted to lowercase.  /// </summary>  public struct RouteKey  {  private string verb;  private string path;  public string Verb  {  get { return verb; }  set { verb = value.ToUpper(); }  }  public string Path  {  get { return path; }  set { path = value.ToLower(); }  }  public override string ToString()  {  return Verb + " : " + Path;  }  } |

## A Route Table

A route table maps the routing key (the verb and path) with a route entry. The ensure thread safety, we use a ConcurrentDictionary, even though technically, the route table should not be modified after initialization, but we don’t want to constrain the web server application to this – who knows, you may have a very good reason to modify the routing table via a route hander!

|  |
| --- |
| public class RouteTable  {  protected ConcurrentDictionary<RouteKey, RouteEntry> routes;  public RouteTable()  {  routes = new ConcurrentDictionary<RouteKey, RouteEntry>();  }  /// <summary>  /// True if the routing table contains the verb-path key.  /// </summary>  public bool ContainsKey(RouteKey key)  {  return routes.ContainsKey(key);  }  /// <summary>  /// True if the routing table contains the verb-path key.  /// </summary>  public bool Contains(string verb, string path)  {  return ContainsKey(NewKey(verb, path));  }  /// <summary>  /// Add a unique route.  /// </summary>  public void AddRoute(RouteKey key, RouteEntry route)  {  routes.ThrowIfKeyExists(key, "The route key " + key.ToString() +   " already exists.")[key] = route;  }  /// <summary>  /// Adds a unique route.  /// </summary>  public void AddRoute(string verb, string path, RouteEntry route)  {  AddRoute(NewKey(verb, path), route);  }  /// <summary>  /// Get the route entry for the verb and path.  /// </summary>  public RouteEntry GetRouteEntry(RouteKey key)  {  return routes.ThrowIfKeyDoesNotExist(key, "The route key " + key.ToString() +  " does not exist.")[key];  }  /// <summary>  /// Get the route entry for the verb and path.  /// </summary>  public RouteEntry GetRouteEntry(string verb, string path)  {  return GetRouteEntry(NewKey(verb, path));  }  /// <summary>  /// Returns true and populates the out entry parameter if the key exists.  /// </summary>  public bool TryGetRouteEntry(RouteKey key, out RouteEntry entry)  {  return routes.TryGetValue(key, out entry);  }  /// <summary>  /// Returns true and populates the out entry parameter if the key exists.  /// </summary>  public bool TryGetRouteEntry(string verb, string path, out RouteEntry entry)  {  return routes.TryGetValue(NewKey(verb, path), out entry);  }  /// <summary>  /// Create a RouteKey given the verb and path.  /// </summary>  public RouteKey NewKey(string verb, string path)  {  return new RouteKey() { Verb = verb, Path = path };  }  } |

## The Route Handler

The route handler vectors the request to the supplied handler, if one exists:

|  |
| --- |
| /// <summary>  /// Route requests to an application-defined handler.  /// </summary>  public class RouteHandler  {  protected RouteTable routeTable;  protected SessionManager sessionManager;  public RouteHandler(RouteTable routeTable, SessionManager sessionManager)  {  this.routeTable = routeTable;  this.sessionManager = sessionManager;  }  /// <summary>  /// Route the request. If no route exists, the workflow continues, otherwise,   /// we return the route handler's continuation state.  /// </summary>  public WorkflowState Route(WorkflowContinuation<HttpListenerContext>  workflowContinuation, HttpListenerContext context)  {  WorkflowState ret = WorkflowState.Continue;  RouteEntry entry = null;  Session session = sessionManager != null ? sessionManager[context] : null;  if (routeTable.TryGetRouteEntry(context.Verb(), context.Path(), out entry))  {  if (entry.RoutingProvider != null)  {  ret = entry.RoutingProvider(workflowContinuation, context, session);  }  }  return ret;  }  } |

Again, we’ll look at sessions and session management in the next chapter, but for now we can ignore the session management property.

## Try it Out

We can test this very simply, by writing a handler for a page we want to fault on:

|  |
| --- |
| public static void InitializeRouteHandler()  {  routeTable = new RouteTable();  routeTable.AddRoute("get", "restricted", new RouteEntry()   {   RoutingProvider = (continuation, context) =>   {   throw new ApplicationException("You can’t do that.");   }   });  routeHandler = new RouteHandler(routeTable);  } |

Here we’re leveraging the previously implemented exception handler to display the message in the browser window. When we request (via “get”) this page, we’ll get the message “You can’t do that.”

We add the routing handler to our workflow:

|  |
| --- |
| public static void InitializeWorkflow(string websitePath)  {  StaticContentLoader sph = new StaticContentLoader(websitePath);  workflow = new Workflow<HttpListenerContext>(AbortHandler, OnException);  workflow.AddItem(new WorkflowItem<HttpListenerContext>(LogIPAddress));  workflow.AddItem(new WorkflowItem<HttpListenerContext>(WhiteList));  workflow.AddItem(new WorkflowItem<HttpListenerContext>(requestHandler.Process));  **workflow.AddItem(new WorkflowItem<HttpListenerContext>(routeHandler.Route));**  workflow.AddItem(new WorkflowItem<HttpListenerContext>(sph.GetContent));  } |

And voila!

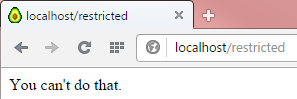


Figure 8: Routing Example

## Qualifying Routes by Content Type

It may also be useful to qualify a route handler by the content type. Let’s say you have a route where you need to handle both “application/json” (say, from an AJAX call) and “application/x-www-form-urlencoded” (say, from a form post). It could be useful to qualify the route by the content type in addition to the verb and path. As it turns out, some web servers don’t actually support that ability, but as we see in the chapter “Form Parameters and AJAX”, content type can be a useful qualifier.

IMPORTANT: Because not all web servers support qualifying routes by content type, you may discover that your web application all of a sudden breaks! Use this feature with care. Marc LaFleur wrote an excellent article[[35]](#footnote-35) on adding content type routing to ASP.NET Web API.

## Conclusion

Routing is great example of the different ways one can write the handlers – you can use anonymous methods, like I did above, or an instance method, or a static method. You can add extension methods or just define methods that promote session and authentication check re-use, which we’ll explain in the next chapter on Sessions. Also, you should be getting a sense of the repeatability of the workflow pattern. We will take advantage of the same pattern for Session and Authorization in the next chapter.

# Sessions

Handlers are stateless – they have to be because the same code could be executing on hundreds of threads. However, there obviously is a need to maintain information between requests, typically an authorization token, perhaps a user name, the last request time, and so forth. These pieces of information are all managed in a stateful session which is associated with the user’s IP address.

The session management provided here is really a basic implementation, and I certainly don’t want presume what your authorization, session expiration, and user role management needs to be. So, as we saw with routing, you can use the routing handler that is “in the can” or you can provide a different routing handler to the workflow.

In this chapter we’ll add two separate workflow steps, one for checking session expiration and the other for checking authorization. As mentioned in the previous chapter, these are entwined with the request verb and path, so, like the route provider, we’ll be implementing a test to see if a provider exists, and if so, continue or terminate the workflow based on the provider’s response.

We should also talk about Cross-Site Request Forgery (CSRF) when we’re discussing sessions, as this is a token that is preserved within the context of a session.

## Session

First, we need a container for the concept of a session. In the implementation below, note that the session provides three things:

1. A way to manage whether the session has expired or not
2. A way to manage whether the user is authorized or not
3. A general collection of key-value pairs that application may want to preserve in a session across requests.

|  |
| --- |
| /// <summary>  /// Sessions are associated with the client IP.  /// </summary>  public class Session  {  public DateTime LastConnection { get; set; }  /// <summary>  /// Is the user authorized?  /// </summary>  public bool Authorized { get; set; }  /// <summary>  /// This flag is set by the session manager if the session has expired between  /// the last connection timestamp and the current connection timestamp.  /// </summary>  public bool Expired { get; set; }  /// <summary>  /// Can be used by controllers to add additional information that needs  /// to persist in the session.  /// </summary>  private ConcurrentDictionary<string, object> Objects { get; set; }  // Indexer for accessing session objects. If an object isn't found,   // null is returned.  public object this[string objectKey]  {  get  {  object val = null;  Objects.TryGetValue(objectKey, out val);  return val;  }  set { Objects[objectKey] = value; }  }  /// <summary>  /// Object collection getter with type conversion.  /// Note that if the object does not exist in the session, the default  /// value is returned.  /// Therefore, session objects like "isAdmin" or "isAuthenticated"  /// should always be true for their "yes" state.  /// </summary>  public T GetObject<T>(string objectKey)  {  object val = null;  T ret = default(T);  if (Objects.TryGetValue(objectKey, out val))  {  ret = (T)Converter.Convert(val, typeof(T));  }  return ret;  }  public Session()  {  Objects = new ConcurrentDictionary<string, object>();  UpdateLastConnectionTime();  }  public void UpdateLastConnectionTime()  {  LastConnection = DateTime.Now;  }  /// <summary>  /// Returns true if the last request exceeds the specified expiration  /// time in seconds.  /// </summary>  public bool IsExpired(int expirationInSeconds)  {  return (DateTime.Now - LastConnection).TotalSeconds > expirationInSeconds;  }  /// <summary>  /// De-authorize the session.  /// </summary>  public void Expire()  {  Authenticated = false;  Expired = true;  }  } |

Note also that we’re using a ConcurrentDictionary, as it is possible that the application may, unbeknownst to us, set up its own worker threads in a request, where each thread might be concurrently accessing session information.

## Session Manager

Next, we need a session manager. The session manager creates the session if it doesn’t exist. If it does exist, it updates the Expired flag if the session has expired – this is based on whether the time since the last request exceeds the expiration time which is by default set to 10 minutes.

|  |
| --- |
| public class SessionManager  {  public string CsrfTokenName { get; set; }  public int ExpireInSeconds { get; set; }   protected RouteTable routeTable;  /// <summary>  /// Track all sessions.  /// </summary>  protected ConcurrentDictionary<IPAddress, Session> sessionMap;  public SessionManager(RouteTable routeTable)  {  this.routeTable = routeTable;  sessionMap = new ConcurrentDictionary<IPAddress, Session>();  CsrfTokenName = "\_CSRF\_";  ExpireInSeconds = 10 \* 60;  }  public WorkflowState Provider(  WorkflowContinuation<HttpListenerContext> workflowContinuation,  HttpListenerContext context)  {  Session session;  IPAddress endpointAddress = context.EndpointAddress();  if (!sessionMap.TryGetValue(endpointAddress, out session))  {  session = new Session();  session[CsrfTokenName] = Guid.NewGuid().ToString();  sessionMap[endpointAddress] = session;  }  else  {  // If the session exists, set the expired flag before we   // update the last connection date/time.  // Once set, stays set until explicitly cleared.  session.Expired |= session.IsExpired(ExpireInSeconds);  }  session.UpdateLastConnectionTime();  WorkflowState ret = CheckExpirationAndAuthorization(  workflowContinuation, context, session);  return ret;  }  protected WorkflowState CheckExpirationAndAuthorization(  WorkflowContinuation<HttpListenerContext> workflowContinuation,  HttpListenerContext context,   Session session)  {  // Inspect the route to see if we should do session   // expiration and/or session authorization checks.  WorkflowState ret = WorkflowState.Continue;  RouteEntry entry = null;  if (routeTable.TryGetRouteEntry(context.Verb(), context.Path(), out entry))  {  if (entry.SessionExpirationProvider != null)  {  ret = entry.SessionExpirationProvider(workflowContinuation, context, session);  }  if (ret == WorkflowState.Continue)  {  if (entry.AuthorizationProvider != null)  {  ret = entry.AuthorizationProvider(workflowContinuation, context, session);  }  }  }  return WorkflowState.Continue;  }  } |

## CSRF Token

For new sessions, a CSRF token is registered. We can use this token when we render pages, embedding it into put/post/delete requests to the server to protect the data on the server from someone maliciously forging user activity. We’ll discuss this in a later chapter on view engines.

Again, note the use of the ConcurrentDictionary, as we are most likely dealing with concurrent access to the server-wide session manager.

## Try it Out

First, let’s create our session manager instance and add it to the workflow:

|  |
| --- |
| public static void InitializeSessionManager()  {  sessionManager = new SessionManager(routeTable);  } |

Let’s set up a couple web pages that let us play with explicitly setting expiration and authorization. We want a page that tells us whether we have an expired or unauthorized request. As with the routing example, we’ll just throw an exception if the session is expired or unauthorized.

|  |
| --- |
| routeTable.AddRoute("get", "testsession", new RouteEntry()  {  SessionExpirationProvider = (continuation, context, session) =>  {  if (session.Expired)  {  throw new ApplicationException("Session has expired!");  }  else  {  return WorkflowState.Continue;  }  },  AuthorizationProvider = (continuation, context, session) =>  {  if (!session.Authorized)  {  throw new ApplicationException("Not authorized!");  }  else  {  return WorkflowState.Continue;  }  },  RouteHandler = (continuation, context, session) =>  {  context.RespondWith("<p>Looking good!</p>");  return WorkflowState.Done;  }  }); |

We also want a page that lets us set and clear the expired and authorized flags. Note that for the purposes of this demonstration we do not test this page for expiration or authentication! Here we’ll have a little fun with URL parameters in the route handler:

|  |
| --- |
| routeTable.AddRoute("get", "SetState", new RouteEntry()  {  RoutingProvider = (continuation, context, session) =>  {  Dictionary<string, string> parms = context.GetUrlParameters();  session.Expired = GetBooleanState(parms, "Expired", false);  session.Authorized = GetBooleanState(parms, "Authorized", false);  context.RespondWith(  "<p>Expired has been set to " + session.Expired + "</p>"+  "<p>Authorized has been set to "+session.Authorized + "</p>");  return WorkflowState.Done;  }  }); |

And we have a little helper function to convert some different ways of expressing yes and no to a boolean:

|  |
| --- |
| public static bool GetBooleanState(  Dictionary<string, string> parms,   string key,   bool defaultValue)  {  bool ret = defaultValue;  string val;  if (parms.TryGetValue(key.ToLower(), out val))  {  switch(val.ToLower())  {  case "false":  case "no":  case "off":  ret = false;  break;  case "true":  case "yes":  case "on":  ret = true;  break;  }  }  return ret;  } |

We’ll first test a non-expired, authorized site, to which we get back:

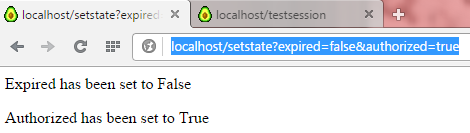


Figure 9: Not Expired, Authorized

Now when we test our state with the “testsession” URL, we get:

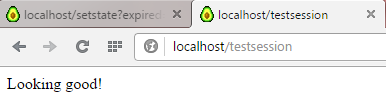


Figure 10: Looking Good!

We can expire the session:

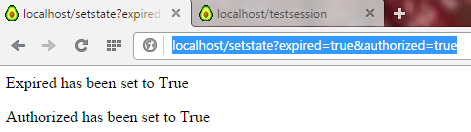


Figure 11: Expire the Session

To which we get:

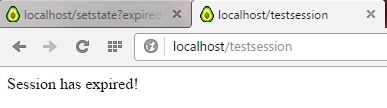


Figure 12 Session has Expired

Lastly, we can de-authorize the session:

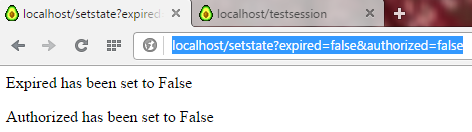


Figure 13: De-authorize the Session

And we get:

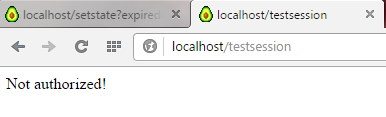


Figure 14: The Session is no Longer Authorized

## Automatically Cleaning Up Expired Sessions

It’s important that we clean up expired sessions, however the question becomes, when do we really delete any knowledge of the session, vs. potentially giving the user some feedback like “you’re session has expired, please log in again”? Truly deleting a session should happen sometime after it has expired, but ultimately, this is a decision for the developer creating the web application. At best, we can offer this function in the session manager that cleans up sessions with a specific “haven’t seen any user activity since this date/time” criteria:

|  |
| --- |
| public void CleanupDeadSessions(int deadAfterSeconds)  {  sessionMap.Values.Where(s =>   s.IsExpired(deadAfterSeconds)).ForEach(s =>   sessionMap.Remove(s.EndpointAddress));  } |

It’s really up to you to decide when you want to call that function, but I suggest a worker thread that fires every minute or so.

## Re-Use

In the above code, I embed the session expiration and authorization checks as anonymous functions. This isn’t an easily re-usable pattern – I certainly do not recommend that you copy and paste a couple anonymous methods for every route handler—the above is simply to keep the code example tight.

You could, for example, add some extension methods to the RouteTable:

|  |
| --- |
| public static class RouteTableExtensions  {  /// <summary>  /// Add a route with session expiration checking.  /// </summary>  public static void AddExpirableRoute(this RouteTable routeTable,  string verb,  string path,  Func<WorkflowContinuation<HttpListenerContext>, HttpListenerContext, Session,   PathParams, WorkflowState> routeHandler)  {  routeTable.AddRoute(verb, path, new RouteEntry()  {  SessionExpirationHandler = (continuation, context, session, parms) =>  {  /\* Your expiration check \*/  return WorkflowState.Continue;  },  RouteHandler = routeHandler,  });  }  /// <summary>  /// Add a route with session expiration and authorization checking.  /// </summary>  public static void AddExpirableAuthorizedRoute(this RouteTable routeTable,  string verb,  string path,  Func<WorkflowContinuation<HttpListenerContext>, HttpListenerContext, Session,  PathParams, WorkflowState> routeHandler)  {  routeTable.AddRoute(verb, path, new RouteEntry()  {  SessionExpirationHandler = (continuation, context, session, parms) =>  {  /\* Your expiration check \*/  return WorkflowState.Continue;  },  AuthorizationHandler = (continuation, context, session, parms) =>  {  /\* Your authentication check \*/  return WorkflowState.Continue;  },  RouteHandler = routeHandler,  });  }  } |

You can now re-use the expiration check and authorization check more easily, for example:

|  |
| --- |
| routeTable.AddExpirableRoute("get", "somepath", myRouteHandler);  routeTable.AddExpirableAuthorizedRoute("get", "someotherpath", myRouteHandler); |

## Conclusion

At this point, our web server is providing a lot of capability. We can:

* Manage session state.
* Incorporate session expiration and authorization into our routes.
* Route requests to custom handlers.
* Implement behaviors based on the URL and request body parameters.
* We can respond with a default page, custom HTML, and/or a custom response body.
* We can handle REST endpoint calls.

However, there’s still a few things that are left to do, such as:

* Parameterized routes (routes with id’s embedded in them)
* Better error handling – throwing exceptions for things like “page not found” and “expired session” is not ideal!
* Support for HTTPS
* View engines
* Some AJAX examples would be nice

We’ll look at these issues in the remaining chapters.

# HTTPS

HTTPS is a requirement today for all web servers. For this reason, I’ve put the chapter on HTTPS here, so that it is not ignored or left as “optional reading” at the end of the book. This chapter will show you how to create your own certificate to enable HTTPS communication with your web server. Creating your own certificate requires users to accept an unknown publisher certificate, but none-the-less, this is still informative for how to set up your web server to handle HTTPS.

Please note that these instructions are written for Visual Studio 2012 and Windows 7.

There are three levels of SSL certificates:

1. Domain Validation
2. Organization Validation
3. Extended Validation

Among other things, the more validation the certificate provides, the more expensive it is to obtain.

## Domain Validation

Domain Validation (DV) establish a baseline level of trust with a website, ensuring that the website you are visiting is really the website you intend to visit. This is the certificate we’ll be creating here. The disadvantage of this certificate is that they are usually easily obtainable and only secure the communication between your browser and the server.

## Organization Validation

This is a more secure certificate because it requires some company information to be verified along with the domain and owner information. In addition to encrypting data, you have an added level of trust about the company that runs the website.

## Extended Validation

This level of certification requires that the company goes through a vetting process in which all the details of the company are verified. Only companies that pass a thorough vetting can use this level of certificate.

## How to Make a Domain Level Certificate

In this process, we’ll make our own certificate. Because we are not a certificate authority, the browser will still question whether the certificate is legitimate. It will do this the first time you visit a page on your web server. Creating your own certificate is useful though for testing and to enable SSL so that the client’s data is at least encrypted.

The steps described here look daunting but have been thoroughly tested and if you follow them precisely, you should not have any issues.

To begin, launch the Visual Studio Command Line prompt. For example, if you're using VS2012 in Windows 7, click on:

1. Start
2. All Programs
3. Microsoft Visual Studio 2012
4. Visual Studio Tools
5. Developer Command Prompt for VS2012

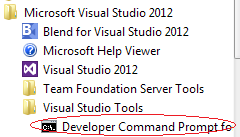


Figure 15: Developer Command Prompt

at the console window, type in (all on one line), but replace [computername] with either your computer name or a domain name. Read more about makecert options on [MSDN](http://msdn.microsoft.com/en-us/library/bfsktky3.aspx).

|  |
| --- |
| Makecert -r -pe -n CN="[computername]" -eku 1.3.6.1.5.5.7.3.1 -ss my -sr localmachine -sky exchange -sp "Microsoft RSA SChannel Cryptographic Provider" -sy 12 |

For example (I'm using "test" and the computer name, this is just an example):

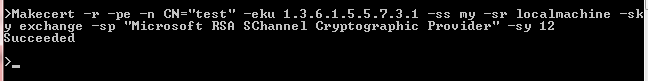


Figure 16: Makecert Example

### Make the Certificate Trusted

From the console window above, after the certificate has been created, type:

MMC

to launch the Microsoft Management Console.

### Add the “Certificates” Snap In

From the File menu, select "Add/Remove Snap-in..."

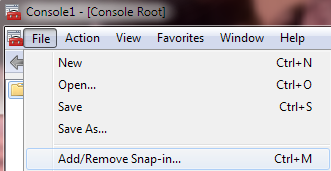


Figure 17: Add Certificates Snap In

Select "Certificates" and click on "Add":

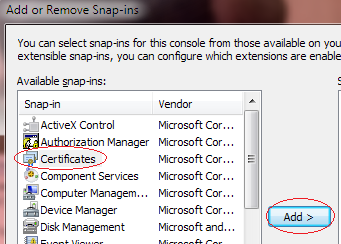


Figure 18: Select "Certificates"

Select "Computer account":

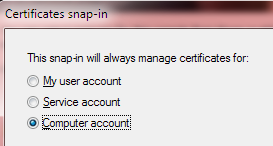


Figure 19: Select "Computer account"

Then click on Next -> Finish -> OK.

### Verify Certificate Creation

Double click on the Personal folder, then the Certificates folder.

Double click on the certificate, and you should see a dialog stating "This CA Root certificate is not trusted."

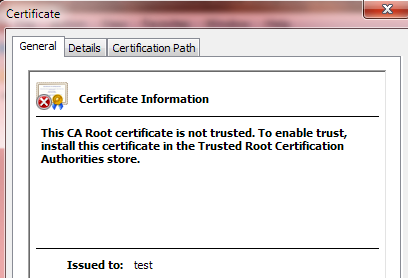


Figure 20: Certificate Information

If you get any other message, the certificate will not work correctly.

### Get the Certificate Thumbprint

Click on the Details tab and scroll down to select the "Thumbprint" field. Copy the value into notepad as this will be used later:

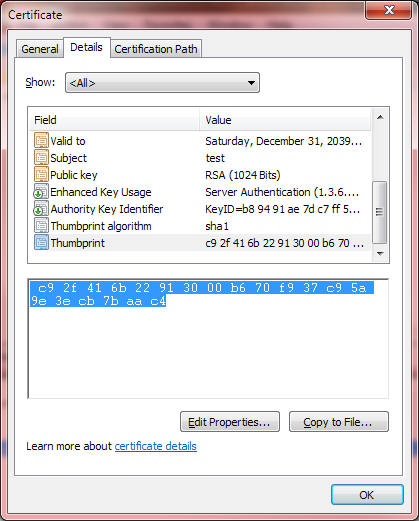


Figure 21: Certificate Thumbprint

Use notepad's search/replace to remove all the whitespace:

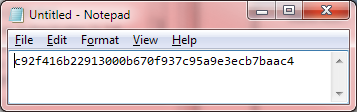


Figure 22: Remove Whitespace

### Copy the Certificate to the Trusted Root Certification Authorities Folder

1. In the same Details tab, click on "Copy to File..."
2. Click Next
3. Click Next (do not export the private key)
4. Click Next (DER encoded binary X.509)
5. Enter a filename for your certificate, such as "c:\temp\mycert.cer"
6. Click Finish
7. Close the Certficate dialog (click OK)
8. Now, open the Trusted Root Certification Authorities\Certificates folder:

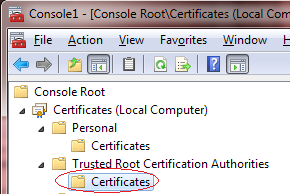


Figure 23: Certificates Folder

Right-click on the Certificates folder and select All Tasks -> Import...

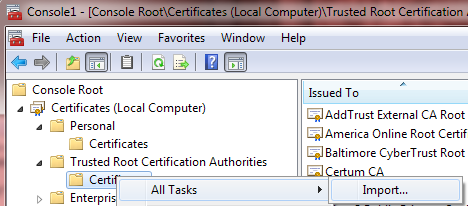


Figure 24: Import Certificate

1. Click Next
2. Enter in the filename of the certificate you just exported.
3. Click Next
4. Click Next (Place all certificates in the follow store --. Trusted Root Certification Authorities)
5. Click Finish

### Verify the Certificate is now Trusted

Go back to the Personal/Certificate folder and double-click on the certificate that you created earlier. Verify that you now see "This certificate is intended for the following purpose(s):"

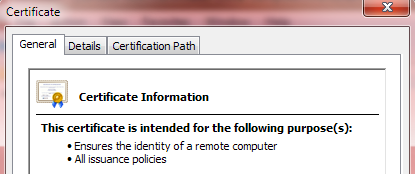


Figure 25: Verify Certficate Is Trusted

### Bind the Certificate to all IP addresses and Port on the Machine

In the VS2012 command line console window we opened earlier, use netsh (read more on [MSDN](https://msdn.microsoft.com/en-us/library/cc307236(VS.85).aspx)), bind the certificate to all IP addresses and the SSL port:

|  |
| --- |
| netsh http add sslcert ipport=0.0.0.0:443 certhash=[yourhash] appid={[your app ID]} |

replacing [yourhash] with the thumbprint we obtained above. The value of [your app ID] should be a GUID associated with your application. For example, I used the GUID in the Properties\AssemblyInfo.cs folder for the "assembly: Guid" key.

So, you netsh command, using the thumbprint we acquired above and a GUID from an application, would look like this (all on one line):

|  |
| --- |
| netsh http add sslcert ipport=0.0.0.0:443 certhash=‎c92f416b22913000b670f937c95a9e3ecb7baac4 appid={1a1af1ff-1663-4e58-915a-6ea844508a33} |

### That’s All!

Your computer is now ready to respond to HTTPS web requests (assuming of course you set up the web server to listen to HTTPS.) The first time you browse to a page, Windows will prompt you to accept the certificate from an unknown authority. For example, Opera gives you this message:

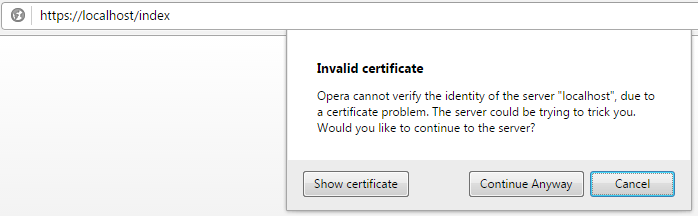


Figure 26: Certificate Warning

To avoid this warning, you would have to obtain a certificate from a trusted authority, such as [Verisign](http://www.verisign.com). There are a variety of such authorities and the pricing for a certificate varies tremendously, so it is worth the effort to investigate the differences. Also, your host provider may provide SSL certificates as well.

## Enabling The Web Server to Receive Port 443 Requests

We do, however, have to set up our web server to listen to SSL requests, which by default are on port 443:

|  |
| --- |
| listener.Prefixes.Add("https://localhost:443/"); |

It’s literally as simple as that.

# Error Handling / Redirecting

Throwing an exception is costly and displaying the exception message isn’t the most user friendly thing to do, and can potentially reveal the inner details of your server, making it more vulnerable to attack. Instead of throwing exceptions, we should redirect the user to an error page. Because redirecting is a common action by a route handler, we’ll implement this in a general purpose way and discover something interesting in the process.

Typical error pages include:

* Session Expired
* Page Not Found
* File Not Found
* Not Authorized
* Server Error

Where “Server Error” is the catch-all for actual exceptions thrown by the server code.

You can put these pages wherever you like for your website—I tend to put them in a Website\ErrorPages folder.

We’ll refactor the routes “restricted” and “testsession” that we’d created above earlier to do a page redirect instead:

|  |
| --- |
| // Test session expired and authorization flags  routeTable.AddRoute("get", "testsession", new RouteEntry()  {  SessionExpirationHandler = (continuation, context, session, parms) =>  {  if (session.Expired)  {  // Redirect instead of throwing an exception.  context.Redirect(@"ErrorPages\expiredSession");  return WorkflowState.Abort;  }  else  {  return WorkflowState.Continue;  }  },  AuthorizationHandler = (continuation, context, session, parms) =>  {  if (!session.Authorized)  {  // Redirect instead of throwing an exception.  context.Redirect(@"ErrorPages\notAuthorized");  return WorkflowState.Abort;  }  else  {  return WorkflowState.Continue;  }  },  RouteHandler = (continuation, context, session, parms) =>  {  context.RespondWith("<p>Looking good!</p>");  return WorkflowState.Done;  }  }); |

You’ll note in the above code that I’m using the Windows path \ separator. We have to fix that in the Redirect extension method:

|  |
| --- |
| /// <summary>  /// Redirect to the designated page.  /// </summary>  public static void Redirect(this HttpListenerContext context, string url)  {  url = url.Replace('\\', '/');  HttpListenerRequest request = context.Request;  HttpListenerResponse response = context.Response;  response.StatusCode = (int)HttpStatusCode.Redirect;  string redirectUrl = request.Url.Scheme + "://" + request.Url.Host + "/" + url;  response.Redirect(redirectUrl);  response.OutputStream.Close();  } |

Notice in the above code how the response StatusCode must be set to Redirect. In my testing on different browsers, if we don’t do this, some browsers will not update the URL on the address bar.

Now let’s test it out. We’ll set the session state to expired with our test URL:

|  |
| --- |
| <http://localhost/setstate?expired=true&authorized=true> |

And we see, when we navigate to localhost/testsession:

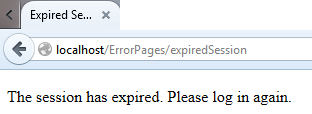


Figure 27: Redirecting

Similarly, we’ll set the session state to unauthorized:

|  |
| --- |
| http://localhost/setstate?expired=false&authorized=false |

And we see:

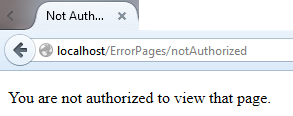


Figure 28: Redirect

We now have a way of handling errors gracefully. We can also replace the “something really bad happened” exception handler with:

|  |
| --- |
| static void OnException(HttpListenerContext context, Exception ex)  {  if (ex is FileNotFoundException)  {  // Redirect to page not found  context.Redirect(@"ErrorPages\pageNotFound");  }  else  {  // Redirect to server error  context.Redirect(@"ErrorPages\serverError");  }  } |

Note here how we’re redirecting to two different pages, depending on the exception.

# Parameterized Routes

A common practice is to add parameters within a route. You’ll see this used frequently in Ruby on Rails applications, though in my personal opinion, I much prefer putting the parameters in the parameter section of a URL. Regardless, there’s no reason to make a “Marc Clifton” opinionated server, so we should support this feature.

What is a parameterized route? It could look like this:

localhost/items/1/subitems

where “1” is the ID of an item in the items collection. Or, another example:

localhost/items/groceries/subitems

where “groceries” is the name of an item in the items collection. There are a variety of assumptions that a router will make with regards to the second form to replace “groceries” with the id value. Here are some of the possible assumptions:

* There’s a table called “Items”
* There’s a model called “Item” in the singular
* The model may define (or not) a way to map a non-numeric parameter to a lookup field
* The router has the ability to look up an ID from a non-numeric parameter, either
  + Directly from the database
  + Indirectly through a model

The issue can be considerably more complex. Consider the routing options that NancyFx[[36]](#footnote-36) supports:

* Literal segments (like mypage/mystuff/foobar)
* Capture segments (what I’m calling parameterized URLs) like “/tasks/{tasked}”
* Optional capture segments
* Capture segments with default values
* RegEx segments
* Greedy segments
* Greedy RegEx segments
* Multiple Captures Segment

These are all potentially useful ways of pattern matching a URL on a route handler. This should give you a sense of the complexities one could introduce into routing. For the purposes of this chapter, we’ll keep it fairly simple and focus on simply capturing the parameter and passing it into the route handler in the PathParams collection. But it does suggest that there be a way to call back to the application for very specialized routing requirements.

## Agreeing on a Syntax

We can use whatever syntax we for how parameters in the path are specified. For example, we might require a form like this (used by Rails):

|  |
| --- |
| param/:p1/subpage/:p2 |

However, we’ll use the ASP.NET MVC and NancyFx form:

|  |
| --- |
| param/{p1}/subpage/{p2} |

## Handling ID’s

Recall in our RouteHandler that we make a call in an attempt to acquire the route handler:

|  |
| --- |
| if (routeTable.TryGetRouteEntry(context.Verb(), context.Path(), out entry)) |

Currently implemented as a few overloaded methods of these two names:

|  |
| --- |
| public RouteEntry GetRouteEntry(RouteKey key)  {  return routes.ThrowIfKeyDoesNotExist(key, "The route key " + key.ToString() + " does not exist.")[key];  }  public bool TryGetRouteEntry(string verb, string path, out RouteEntry entry)  {  return routes.TryGetValue(NewKey(verb, path), out entry);  } |

Here we expect an exact match between the request path and the route’s definition. We need to refactor this code (and some other areas of the code which I will not show because they’re trivial) to match on a parameterized URL and, we would like those parameters returned in a key-value dictionary, for which I’ve simply derived a specific type:

|  |
| --- |
| public class PathParams : Dictionary<string, string>  {  } |

We refactor the GetRouteEntry methods to a form similar to this (not all overloads shown):

|  |
| --- |
| public RouteEntry GetRouteEntry(RouteKey key, out PathParams parms)  {  parms = new PathParams();  RouteEntry entry = Parse(key, parms);  if (entry == null)  {  throw new ApplicationException("The route key " + key.ToString() + " does not exist.");  }  return entry;  } |

We implement simple parser that iterates through the routes, find the first one that matches. This method has to parts: the iterator and the matcher. The iterator:

|  |
| --- |
| /// <summary>  /// Parse the browser's path request and match it against the routes.  /// If found, return the route entry (otherwise null).  /// Also if found, the parms will be populated with any segment parameters.  /// </summary>  protected RouteEntry Parse(RouteKey key, PathParams parms)  {  RouteEntry entry = null;  string[] pathSegments = key.Path.Split('/');  foreach (KeyValuePair<RouteKey, RouteEntry> route in routes)  {  // Above all else, verbs must match.  if (route.Key.Verb == key.Verb)  {  string[] routeSegments = route.Key.Path.Split('/');  // Then, segments must match  if (Match(pathSegments, routeSegments, parms))  {  entry = route.Value;  break;  }  }  }  return entry;  } |

And the matcher (note how we could add additional behaviors here for matching a capture segment):

|  |
| --- |
| /// <summary>  /// Return true if the path and the route segments match. Any parameters in the path  /// get put into parms. The first route that matches will win.  /// </summary>  protected bool Match(string[] pathSegments, string[] routeSegments, PathParams parms)  {  // Basic check: # of segments must be the same.  bool ret = pathSegments.Length == routeSegments.Length;  if (ret)  {  int n = 0;  // Check each segment  while (n < pathSegments.Length && ret)  {  string pathSegment = pathSegments[n];  string routeSegment = routeSegments[n];  ++n;  // Is it a parameterized segment (aka "capture segment") ?  if (routeSegment.BeginsWith("{"))  {  string parmName = routeSegment.Between('{', '}');  string value = pathSegment;  parms[parmName] = value;  }  else // We could perform other checks, such as regex  {  ret = pathSegment == routeSegment;  }  }  }  return ret;  } |

## Test It Out!

Let’s write a route handler that expects two parameters and gives us our parameter values back in the browser. The implementation looks like this (note, the RouteHandler was also refactored to add a PathParams parameter):

|  |
| --- |
| routeTable.AddRoute("get", "param/{p1}/subpage/{p2}", new RouteEntry()  {  RouteHandler = (continuation, context, session, parms) =>  {  context.RespondWith("<p>p1 = " +   parms["p1"] + "</p><p>p2 = " +   parms["p2"] + "</p>");    return WorkflowState.Done;  }  }); |

Now, when we visit that page and substitute some parameters directly into the URL, we see the server echoing back our captured parameters:

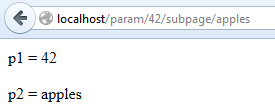


Figure 29: Path Parameters

Notice how we don’t care about the parameter type: it can be a integer, a float, a string, as long as it contains valid characters for the path portion of the URL.

## Conclusion

While relatively easy to implement, parameterized routes add complexity to resolving routes and therefore degrade the performance of the application, especially if you have hundreds of routes and thousands of simultaneous requests. This is why, at the beginning of this chapter, I stated that I do not prefer parameterized URL’s.

While it’s useful to support parameterized routes, we should still support the more optimized lookup implemented earlier. This is accomplished by first checking against the route table with a path “as is”. The implementation of this is not shown here, but is in the source code repo for this book.

# Form Parameters and AJAX

Let’s look at a route handler for a common client-side practice, entering data into a form. We can take a couple approaches here. We’ll first look at a form postback, and next we’ll look at a AJAX post and compare the differences.

## Form Parameters

We’ll start with a basic login HTML form:

|  |
| --- |
| <!DOCTYPE html>  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">  <head>  <meta charset="utf-8" />  <title>Login</title>  <link type="text/css" rel="Stylesheet" href="/CSS/demo.css"/>  </head>  <body>  <form name="myform" action="/login" method="post">  <div class="center-inner top-margin-50">  Username:&nbsp;  <input name="username"/>  </div>  <div class="center-inner top-margin-10">  Password:&nbsp;  <input type="password" name="password"/>  </div>  <div class="center-inner top-margin-10">  <input type="submit" value="Login"/>  </div>  </form>  </body>  </html> |

Backed by a simple route handler (we’re not actually authenticating the user here!):

|  |
| --- |
| // Test a form post  routeTable.AddRoute("post", "login", "application/x-www-form-urlencoded", new RouteEntry()  {  RouteHandler = (continuation, context, session, pathParams) =>  {  string data = new StreamReader(context.Request.InputStream,  context.Request.ContentEncoding).ReadToEnd();  context.Redirect("welcome");  return WorkflowState.Done;  }  }); |

Note how here we’re qualifying the same path with the content type. In the AJAX example below, we will use the same path, but for JSON content.

When we click on the Login button:

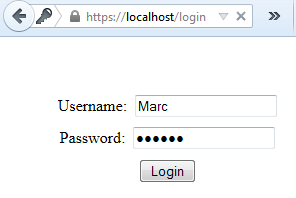


Figure 30: Login

We see that the form parameters are passed in via the request input stream:



Figure 31: Form Parameters

Note how the key in each key-value pair is the value associated with the HTML control’s “name” attribute.

## AJAX Post

We’ll be using jQuery in this example, where we send the username and password as an AJAX POST request:

|  |
| --- |
| <!DOCTYPE html>  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">  <head>  <meta charset="utf-8" />  <title>AJAX Login</title>  <script type="text/javascript" src="/Scripts/jquery-1.11.2.min.js"></script>  <link type="text/css" rel="Stylesheet" href="/CSS/demo.css"/>  <script type="text/javascript">  $(document).ready(function () {  $("#btnLogin").click(function () {  $.ajax({  url: “/login”,  async: true,  cache: false,  type: "post",  data: {  username: $("#username").val(),  password: $("#password").val()  },  success: function (data, status) {  alert(data);  }  });  });  });  </script>  </head>  <body>  <div class="center-inner top-margin-50">  Username:&nbsp;  <input id="username"/>  </div>  <div class="center-inner top-margin-10">  Password:&nbsp;  <input type="password" id="password"/>  </div>  <div class="center-inner top-margin-10">  <input type="submit" value="Login" id ="btnLogin"/>  </div>  </body>  </html> |

Here the input stream’s parameters are exactly the same as in the from POST request and is handled by the same route handler.

But let’s put the data into JSON format and specify the content type:

|  |
| --- |
| contentType: "application/json", data: JSON.stringify({  username: $("#username").val(),  password: $("#password").val()  }), |

We now need a new route handler for the same path (because of we declared the url in the AJAX command) but for a different content type:

|  |
| --- |
| // Test a form post with JSON content  routeTable.AddRoute("post", "login", "application/json; charset=UTF-8", new RouteEntry()  {  RouteHandler = (continuation, context, session, pathParams) =>  {  string data = new StreamReader(context.Request.InputStream, context.Request.ContentEncoding).ReadToEnd();  context.RespondWith("Welcome!");  return WorkflowState.Done;  }  }); |

Here we note that we receive a JSON string:

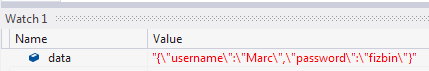


Figure 32: JSON String

The point for bringing this up is that an issue such as using form post vs. an AJAX post or the post data format, while being independent of the server implementation, does impact your web application route handlers. Also, the content type and path, as qualifiers to your route handler, further complicate the design decisions when working with forms, AJAX, and content.

# View Engines

I am not a proponent of view engines, preferring instead to put the rendering logic into the client-side. As I’ve developed web applications, I’ve come to the conclusion that a significant portion of server-side development should actually simply be REST API’s that support the client. This is often the ideal approach for Single Page Applications (SPA’s)[[37]](#footnote-37). If you’re using a view engine for page rendering, then you are most likely doing lots of server-side refreshes which degrades the user experience. In fact, the MVC approach heralded by ASP.NET and Rails is “opinionated” to such a degree that one easily falls into the paradigm of creating a multi-page web app that is server-side-rendered and where updates require a post-back. This is in stark contrast to an SPA web application. This is also why I think the MVC concept is for the most part inappropriate *on the server-side.* And interestingly, you can google lots of articles on how to convert an ASP.NET MVC application to an SPA application.

That said, again I don’t want to impose my opinion on the user, so supporting a view engine such as Razor[[38]](#footnote-38) should definitely be possible in the web server. Now, Razor (like other view engines such as nHaml[[39]](#footnote-39) and Spark[[40]](#footnote-40)) are somewhat weighty in that they will actually create an on-the-fly code file that must be runtime compiled to generate the HTML to be sent to the client. Many times (such as for CSRF replacement or even working with a master page) this is unnecessary – simple keyword replacement will suffice. We’ll look at both approaches, first using the open source templating engine RazorEngine[[41]](#footnote-41), which is based on Microsoft’s Razor parsing engine. Second, we’ll look at CSRF handling with simple string replacement. If you’re interested in RazorEngine, also check out Rick Strahl’s Westwind.RazorHosting[[42]](#footnote-42) as well.

## First, Some Refactoring

In the code presented earlier, the workflow effectively stops once the page file is loaded or the route handler is invoked. This was an design flaw because it doesn’t allow for any further processing of the data before it is sent to the response stream. To fix this, the HttpListenerContext instance needs a wrapper so that we can also include the pending response:

|  |
| --- |
| public class ContextWrapper  {  public HttpListenerContext Context { get; protected set; }  public Response PendingResponse { get; set; }  public ContextWrapper(HttpListenerContext context)  {  Context = context;  }  /// <summary>  /// Text or HTML response, suitable for input to a view engine.  /// </summary>  public void SetPendingResponse(string text)  {  PendingResponse = new PendingPageResponse() { Html = text };  }  } |

This required touching a lot of files to replace the references to HttpListenerContext with ContextWrapper, but it now allows us to define an explicit responder workflow step:

|  |
| --- |
| public static void InitializeWorkflow(string websitePath)  {  StaticContentLoader sph = new StaticContentLoader(websitePath);  workflow = new Workflow<ContextWrapper>(AbortHandler, OnException);  workflow.AddItem(new WorkflowItem<ContextWrapper>(LogIPAddress));  workflow.AddItem(new WorkflowItem<ContextWrapper>(WhiteList));  workflow.AddItem(new WorkflowItem<ContextWrapper>(sessionManager.Provider));  workflow.AddItem(new WorkflowItem<ContextWrapper>(requestHandler.Process));  workflow.AddItem(new WorkflowItem<ContextWrapper>(routeHandler.Route));  workflow.AddItem(new WorkflowItem<ContextWrapper>(sph.GetContent));  workflow.AddItem(new WorkflowItem<ContextWrapper>(Responder));  } |

The responder is implemented as follows:

|  |
| --- |
| /// <summary>  /// Final step is to actually issue the response.  /// </summary>  public static WorkflowState Responder(WorkflowContinuation<ContextWrapper>  workflowContinuation, ContextWrapper wrapper)  {  wrapper.Context.Response.ContentEncoding = wrapper.PendingResponse.Encoding;  wrapper.Context.Response.ContentType = wrapper.PendingResponse.MimeType;  wrapper.Context.Response.ContentLength64 = wrapper.PendingResponse.Data.Length;  wrapper.Context.Response.OutputStream.Write(  wrapper.PendingResponse.Data,   0,  wrapper.PendingResponse.Data.Length);  wrapper.Context.Response.StatusCode = 200;  wrapper.Context.Response.OutputStream.Close();  return WorkflowState.Continue;  } |

## Adding The View Engine

Here we see at last the full beauty of the workflow and how it lets us create a workflow tailored to our web application’s needs. To add Razor view engine processing, we need these two assembly references:

|  |
| --- |
| using RazorEngine;  using RazorEngine.Templating; |

And the implementation:

|  |
| --- |
| /// <summary>  /// Apply the Razor view engine to a page response.  /// </summary>  public static WorkflowState ViewEngine(  WorkflowContinuation<ContextWrapper> workflowContinuation,   ContextWrapper wrapper)  {  PendingPageResponse pageResponse = wrapper.PendingResponse as PendingPageResponse;    // Only send page responses to the templating engine.  if (pageResponse != null)  {  string html = pageResponse.Html;  string templateKey = html.GetHashCode().ToString();  pageResponse.Html = Engine.Razor.RunCompile(html, templateKey, null,   new { /\* your dynamic model \*/ });  }  return WorkflowState.Continue;  } |

Here we initialize a template key has the hash of the HTML. The template key is used for caching purposes – if the template is exactly the same, there’s no reason to re-build the assembly -- we can simply execute it again. Please refer to the excellent RazorEngine GitHub site[[43]](#footnote-43) for details on the additional usage of Engine.Razor.RunCompile.

We’ll talk about models in the next section. For now, we can add the view engine to our workflow:

|  |
| --- |
| public static void InitializeWorkflow(string websitePath)  {  // ... all the previous steps ...  workflow.AddItem(new WorkflowItem<ContextWrapper>(ViewEngine));  workflow.AddItem(new WorkflowItem<ContextWrapper>(Responder));  } |

Now let’s write a simple test page:

|  |
| --- |
| <!DOCTYPE html>  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">  <head>  <meta charset="utf-8" />  <title>Razor Test</title>  </head>  <body>  <ul>  @for (int i=0; i<10; i++)  {  <li>@i</li>  }  </ul>  </body>  </html> |

Here’s the result:

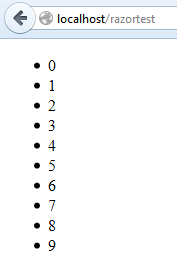


Figure 33: Razor View Engine

Great! We’ve added a sophisticated view engine to our web server. Because the view engine is a workflow step, the implementation actually is done in our web application rather than in the web server, leaving our web server un-opinionated with regards to any view engine.

### Models

Here we run into an interesting problem, simply because our web server is MVC agnostic. We haven’t done anything with models. Furthermore, the view engine lets us specify only one model. This is rather unrealistic – I may have a web page that displays data from many different models. In classic MVC, the workaround to this is to create a “View Model” (hence we’re actually implemented the Model-View-ViewModel, or MVVM pattern). This is, at best, an awkward workaround, but it is what we have to live with with regards to the Razor view engine.

Let’s create a simple model consisting of the names of 2015 Code Project MVP winners:

|  |
| --- |
| public class Person  {  public string Name {get;set;}  public Person(string name)  {  Name=name;  }  }  public class Program  {  public static List<Person> codeProject2015Mvp = new List<Person>()  {  new Person("Christian Graus"),  new Person("BillWoodruff"),  new Person("Richard Deeming"),  new Person("Marc Clifton"),  // ... etc ...  }  // ... etc ... |

We’ll use this collection as the model:

|  |
| --- |
| pageResponse.Html = Engine.Razor.RunCompile(  html,   templateKey,   null,   new { People = codeProject2015Mvp }); |

Now with a little Razor markup:

|  |
| --- |
| <!DOCTYPE html>  <html lang="en" xmlns="http://www.w3.org/1999/xhtml">  <head>  <meta charset="utf-8" />  <title>2015 MVP’s</title>  </head>  <body>  <table>  @foreach (var person in Model.People)  {  <tr>  <td>@person.Name</td>  </tr>  }  </table>  </body>  </html> |

We get (a partial screenshot, apologies if you’re name is not on this screenshot):

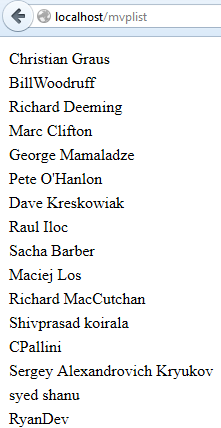


Figure 34: Razor Model

Now the question becomes, how do we associate the HTML page (the view!) with the model that we want to pass into the view engine? The answer is left as an exercise for the reader, as it really is outside the purview of an MVC-agnostic web server implementation.

|  |
| --- |
|  |

### Cleaning up Temporary Files

It’s worth noting that we should run the application in its own application domain so that the temporary files that are created are properly cleaned up. Currently, we see this warning generated by RazorEngine:

E:\WebServersSuccinctly\e-book\razor2.png

Figure 35: View Engine Temp Files

To fix this, I will borrow almost verbatim the code in the RazorEngine quickstart guide to add an application domain wrapper. This wrapper restarts the application in a custom application domain. When that domain exits, RazorEngine will clean up the temporary files. It’s a bit counter-intuitive just looking at the code until you realize that Main runs once in the default domain, and again (falling through the “if” statement) in the new domain:

|  |
| --- |
| static void Main(string[] args)  {  if (AppDomain.CurrentDomain.IsDefaultAppDomain())  {  Console.WriteLine("Switching to secound AppDomain, for RazorEngine...");  AppDomainSetup adSetup = new AppDomainSetup();  adSetup.ApplicationBase = AppDomain.CurrentDomain.SetupInformation.ApplicationBase;  var current = AppDomain.CurrentDomain;  var domain = AppDomain.CreateDomain(  "MyMainDomain", null,  current.SetupInformation,   new PermissionSet(PermissionState.Unrestricted),   null);  var exitCode = domain.ExecuteAssembly(Assembly.GetExecutingAssembly().Location);  // RazorEngine will cleanup.  AppDomain.Unload(domain);  return;  }  // Continue with our sever initialization... |

|  |
| --- |
|  |

|  |
| --- |
|  |

TODO: What if the response’s data is not in the OutputStream but rather in the response body?

# CSRF

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

# A Simple Website Example

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

1. http://en.wikipedia.org/wiki/CERN\_httpd [↑](#footnote-ref-1)
2. http://en.wikipedia.org/wiki/WorldWideWeb [↑](#footnote-ref-2)
3. http://en.wikipedia.org/wiki/CERN [↑](#footnote-ref-3)
4. http://en.wikipedia.org/wiki/ENQUIRE [↑](#footnote-ref-4)
5. http://en.wikipedia.org/wiki/HTML [↑](#footnote-ref-5)
6. http://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html [↑](#footnote-ref-6)
7. http://en.wikipedia.org/wiki/Web\_server [↑](#footnote-ref-7)
8. <https://msdn.microsoft.com/en-us/library/ms182212.aspx> [↑](#footnote-ref-8)
9. http://guides.rubyonrails.org/getting\_started.html [↑](#footnote-ref-9)
10. https://msdn.microsoft.com/en-us/library/4w3ex9c2%28v=vs.140%29.aspx [↑](#footnote-ref-10)
11. https://www.nuget.org/packages/Microsoft.AspNet.Razor/ [↑](#footnote-ref-11)
12. http://weblogs.asp.net/scottgu/introducing-razor [↑](#footnote-ref-12)
13. http://slim-lang.com/ [↑](#footnote-ref-13)
14. https://www.ruby-toolbox.com/categories/template\_engines [↑](#footnote-ref-14)
15. http://php.net/ [↑](#footnote-ref-15)
16. <http://www.freeformatter.com/mime-types-list.html> [↑](#footnote-ref-16)
17. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html> [↑](#footnote-ref-17)
18. <http://en.wikipedia.org/wiki/Cross-site_request_forgery> [↑](#footnote-ref-18)
19. [http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller](http://en.wikipedia.org/wiki/Model–view–controller) [↑](#footnote-ref-19)
20. <http://en.wikipedia.org/wiki/Model_View_ViewModel> [↑](#footnote-ref-20)
21. <http://en.wikipedia.org/wiki/Representational_state_transfer> [↑](#footnote-ref-21)
22. [http://en.wikipedia.org/wiki/Ajax\_%28programming%29](http://en.wikipedia.org/wiki/Ajax_(programming)) [↑](#footnote-ref-22)
23. <http://en.wikipedia.org/wiki/AJAJ> [↑](#footnote-ref-23)
24. <http://en.wikipedia.org/wiki/SOAP> [↑](#footnote-ref-24)
25. <https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations> [↑](#footnote-ref-25)
26. <http://www.asp.net/> [↑](#footnote-ref-26)
27. <http://rubyonrails.org/> [↑](#footnote-ref-27)
28. <http://nancyfx.org/> [↑](#footnote-ref-28)
29. <http://en.wikipedia.org/wiki/ASP.NET_Razor_view_engine> [↑](#footnote-ref-29)
30. <https://github.com/grumpydev/SuperSimpleViewEngine> [↑](#footnote-ref-30)
31. <https://www.ruby-toolbox.com/categories/template_engines> [↑](#footnote-ref-31)
32. <http://en.wikipedia.org/wiki/Entity_Framework> [↑](#footnote-ref-32)
33. <http://guides.rubyonrails.org/active_record_basics.html> [↑](#footnote-ref-33)
34. https://msdn.microsoft.com/en-us/library/0ka9477y%28v=vs.95%29.aspx [↑](#footnote-ref-34)
35. <http://massivescale.com/pages/web-api-routing-by-content-type/> [↑](#footnote-ref-35)
36. https://github.com/NancyFx/Nancy/wiki/Defining-routes [↑](#footnote-ref-36)
37. [SPA and the Single Page Myth](http://www.johnpapa.net/pageinspa/) [↑](#footnote-ref-37)
38. <http://en.wikipedia.org/wiki/ASP.NET_Razor_view_engine> [↑](#footnote-ref-38)
39. <http://code.google.com/p/nhaml/> [↑](#footnote-ref-39)
40. <https://github.com/SparkViewEngine/spark> [↑](#footnote-ref-40)
41. <https://antaris.github.io/RazorEngine/index.html> [↑](#footnote-ref-41)
42. <https://github.com/RickStrahl/Westwind.RazorHosting> [↑](#footnote-ref-42)
43. <http://antaris.github.io/RazorEngine/> [↑](#footnote-ref-43)