Web Servers

Succinctly

by Marc Clifton

Syncfusion | Technology Resource Portal

Web Servers Succinctly

By

Marc Clifton

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Peter Shaw
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Marc Clifton lives in a farming community near Harlemville, NY, providing consulting services to companies across the country. He specializes in software architecture and of course gets his hands dirty in implementation as well. Marc has been a multi-year Microsoft MVP and Code Project MVP (183 articles and counting!), and is dedicated to C#/F#/.NET client, server, and web development. He is also interested in anthroposophy, alternative education, and promoting local economies. He’s learning how to play the lyre and is working on a master's degree in psychology.

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. No Google, no Facebook, no Netflix; a world where instead of going to Wikipedia to learn about some foreign country, you have to pull out a volume of an encyclopedia or go to your local library. Forget texting and all the other social media to which we are so accustomed.

Then on Christmas Day, 1990,[1] the first ever web server went live. Who could imagine what that would mean, even ten years later? How strange to think that the only way to access that web server was through a browser program called *WorldWideWeb*,[2] where nowadays there are at least three major competitors, and probably five or six minor ones in the browser market—not to mention the accessibility of the web on your smart devices, appliances, and probably even watches.

The motivation for the World Wide Web (not to be confused with the first browser, the *WorldWideWeb*) dates back even further, to 1980, when Berners-Lee was working at CERN.[3] At CERN, where approximately 10,000 people were working, the exchange of information between numerous and disparate systems was nearly impossible.[4] To address this, Berners-Lee wrote a software project called ENQUIRE, a simple hypertext program that was similar to Apple's HyperCard[5] but with the advantage that it was portable and ran on different systems. ENQUIRE was sort of like a modern-day wiki.

However, management of the content within ENQUIRE was restricted to its user—it was the user's responsibility to keep the information up-to-date, which ultimately became quite a time-consuming process. 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,[6] consisting of 18 elements that 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.

---

[1] http://en.wikipedia.org/wiki/CERN_httpd

[2] http://en.wikipedia.org/wiki/WorldWideWeb

[3] http://en.wikipedia.org/wiki/CERN

[4] http://en.wikipedia.org/wiki/ENQUIRE

[5] HyperCard is an application program and programming tool for Apple Macintosh and Apple IIGS computers that is among the first successful hypermedia systems before the World Wide Web.

[6] http://en.wikipedia.org/wiki/HTML

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.[7] 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 real-time, dynamic, and some would say invasive, with constant notifications of social media, email, and calendar events. It's also become much more an entertainment tool, as opposed to a research tool, with the advent of chat rooms, YouTube, Netflix, any many other non-research based activities being responsible for this change.

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 1990s. 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.[8] 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.

Website developers 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 can now write and share documents in real time, 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, a web application now does many other things; for example, verifying that the person browsing the site has the right permissions to view the page, or managing secure information such as credit card numbers.

These issues are complicated and the lines for who handles what are fuzzy—what functionality does the web server provide versus the web application, and how do the two interact? These are questions that we will investigate further in this book as we build a highly flexible web server with hopefully clean lines of separation to the previous (and other) concerns.

---

[7] http://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html

[8] http://en.wikipedia.org/wiki/Web_server

Regarding the Terminology

There are a lot of technical terms and abbreviations in this book. If you're not familiar with how web servers work, all these new terms may be a bit daunting. I have attempted to provide footnotes for terms when they are encountered with references for learning more about the term, the technology, and so forth. It is suggested for all audiences that the references in the footnotes be used!

Regarding the Subject Matter

While I have a section in Chapter 2 called "Writing a web server is easy," the reality is that it isn't, and because I've tried to keep the material as succinct (pun intended) as possible, there are a lot of subjects covered in this book that are themselves worthy of a *Succinctly* e-book. So I strongly recommend that the reader supplement this book with other information, especially where the reader is unfamiliar with the topic being discussed.

Source Code

The source code for this book can be found on the Syncfusion Bitbucket account.

If you are familiar with Bitbucket, use your favorite method for cloning a repository. If you are not familiar with Bitbucket, you can learn more here.

Regardless, the home page for the code repository has two options on the left side where you can either clone the repository to your desktop or download a zip file of the repository.

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. 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, things did have to change. Each "functional" change has been placed in a separate folder in the repository so that you can easily view, build, and walk through the code as I proceed in the development of the server. In each section, I will indicate which folder has the current code base for the functionality that is being discussed. 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. They are missing from the code repository 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 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 cyclomatic complexity less than the threshold of 25—cyclomatic complexity greater than 25 would be a "violation" according to Microsoft[9]—and certainly no function exceeds eight.

[9] https://msdn.microsoft.com/en-us/library/ms182212.aspx

Hierarchy	Maintainability Index	Cyclomatic Comple...	Lines of Code
Clifton.WebServer (Debug)	86	183	358
Clifton.WebServer	86	183	358
WorkflowState	100	0	0
IRequestHandler	100	1	0
PathParams	100	1	1
PendingByteResponse	100	1	1
PendingFileResponse	100	1	1
RouteEntry	100	1	1
WorkflowItem<T>	85	2	4
WorkflowContext	93	3	5
PendingPageResponse	93	4	4
Server	73	4	14
FileExtensionHandler	94	5	5
ThreadSemaphore	90	5	9
RouteHandler	67	5	12
ContextWrapper	92	6	7
Response	94	6	6
SingleThreadedQueueingHandler	67	9	27
StaticContentLoader	64	9	30
Extensions	71	11	33
WorkflowContinuation<T>	93	11	12
SessionManager	72	16	39
RouteKey	81	16	27
Session	90	18	30
Workflow<T>	77	21	29
RouteTable	72	27	61

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 my book, *Unit Testing Succinctly*.

Chapter 1 Why Build a Web Server?

Modern web application development frameworks such as ASP.NET and its three flavors (Web Forms, MVC, and Web Pages) and non-Microsoft products such as Ruby on Rails all sit firmly between the web server and you, the web application builder. In the Microsoft world, we're used to working with IIS,[10] whereas in the Unix world, Apache[11] and Nginx[12] are commonly used.

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.[13]

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.[14]

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.

We can see right away that developing a web application is typically entangled with an opinionated framework that attempts to dictate how you should build that application.

---

[10] http://en.wikipedia.org/wiki/Internet_Information_Services
[11] http://httpd.apache.org/
[12] http://nginx.org/
[13] http://guides.rubyonrails.org/getting_started.html
[14] https://msdn.microsoft.com/en-us/library/4w3ex9c2%28v=vs.140%29.aspx

While it's a lot to take on at the beginning of this book, I would like to mention early on that there are additional complexities that affect the entire decision-making process as to which server and server "tool set" one chooses. For example, one can usually choose from a variety of view engines. A view engine is a server-side processing tool for creating dynamic webpages using a specific markup syntax, often leading to what is affectionately called "tag soup."[15] For example, Razor,[16] introduced in 2010,[17] 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 and supports other view engines such as Slim,[18] one of more than 24 different template engine offerings[19] in the Rails community.

Another consideration is that these frameworks come with their own ideas of how you should interface with a database. In other words, there is a strong push toward using an Object-Relational Mapper (ORM). In ASP.NET MVC, the preferred ORM is Entity Framework, and with Rails, the ORM is implemented with Active Record.

What does all of this have to do with writing your own web server? According to Wikipedia, "the primary function of a web server is to store, process, and deliver web pages to clients."[20] This, in my opinion, is not actually correct, but it is accurate with regards to today's concept of a web server. It's not correct because technically all a web server should do is hand off the incoming request to a worker process—the web application—and respond with whatever the application returns. However, Wikipedia's comment is accurate in that we see an entanglement of the concept "web server" with the supporting "server framework" and "web application."

In other words, serving content to a browser actually involves three pieces:

1. The web server (managing workers, also known as threads), and possibly handling upfront things like white lists and black lists.
2. Based on the request syntax, processing that request into meaningful entities such as session state and routing.
3. The application-specific response to a route, an authentication request, and so forth.

Once you move into item #2, you pretty much immediately encounter the opinionated framework. There really is no middle ground that provides a minimal but useful implementation for item #2, and that's what this book addresses. This book is about creating that middle ground. In this book, we look at options for threading and options for work processes, and we also provide flexible but minimal scaffolding to support application development, providing features such as session management, routing, and security. Implementing an application with the web server presented here puts you closer to the metal (which actually translates to higher performance and less code) without enforcing overly opinionated implementation requirements.

At least, that's my opinion!

---

[15] http://en.wikipedia.org/wiki/Tag_soup

[16] https://www.nuget.org/packages/Microsoft.AspNet.Razor/

[17] http://weblogs.asp.net/scottgu/introducing-razor

[18] http://slim-lang.com/

[19] https://www.ruby-toolbox.com/categories/template_engines

[20] http://en.wikipedia.org/wiki/Web_server

Chapter 2 Your First Web Server

The source code presented in this chapter is in the folder **Examples\Chapter 2\Demo** in the Bitbucket repository.

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 the following implementation.

Namespaces we need to use:

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
```

Code Listing 1

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;
}
```

Code Listing 2

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.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.

    try
    {
    // 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();
    }
    catch(Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

Code Listing 3

The previous code initializes 20 listeners. Using semaphores, when a request is received, the semaphore is released and a new listener is created. This code can therefore receive 20 requests simultaneously. We rely on the **`await`** mechanism to determine on what thread the continuation (the code after the **`await`**) executes. If you are unfamiliar with the use of **`Task`** and **`async`**/**`await`**, Stephan Cleary has an excellent discussion of **`async`**/**`await`** and execution contexts on his blog at http://blog.stephencleary.com/2012/02/async-and-await.html.

There are two more things we need to do.

First, create an **index.html** file with the contents:

```
<p>Hello World</p>
```

Code Listing 4

The server we just wrote will run within the **bin\Debug** folder (assuming you haven't changed the build configuration from "Debug" to "Release"), so we need to put the **index.html** file into the **bin\Debug** folder so the application can find it when it tries to load the page associated with the URL.

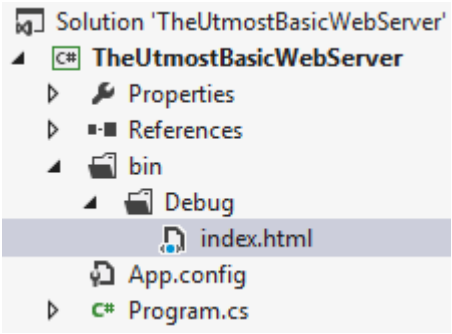


Figure 2: Solution Tree

Second, put an icon file named **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, and enter:

```
http://localhost/index.html
```

Code Listing 5

I am assuming here that you do not have a server already running on port 80 on your machine—if you do, the program will fail.

In the console window you'll see the path emitted, and in the browser you'll see the page rendered as shown in the following figure.

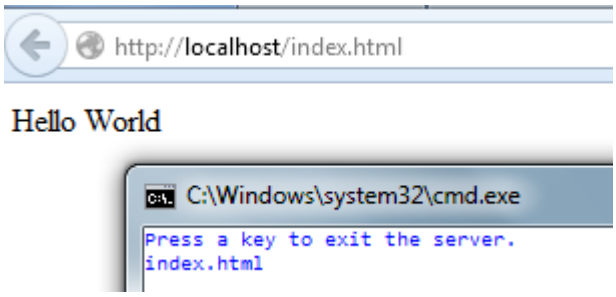


Figure 3: Serving Static Content

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!

We created a simple server that serves only a static HTML page, but there are many things wrong with it:

- Only the 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[21] include CSS, JavaScript, and of course media, such as images.
- It doesn't handle the common HTTP methods,[22] namely **`GET`**, **`POST`**, **`PUT`**, and **`DELETE`**.
- We have no exception handling.
- There's no support for Cross-Site Request Forgery[23] (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.

---

[21] http://www.freeformatter.com/mime-types-list.html

[22] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[23] http://en.wikipedia.org/wiki/Cross-site_request_forgery

- There is no support for routing requests to, say, a Model-View-Controller[24] (MVC) or Model-View-ViewModel[25] (MVVM) architecture.
- Our server implementation 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, and session expiration?
- What about model support?
- What about integration testing?

Request routing combined with some sort of a controller implementation is really useful when implementing a REST[26] API, something our web server should be able to do as well. REST is also at the center of AJAX[27] and AJAJ[28] requests (SOAP[29] is 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.[30]

We Need an Architecture

If you look at a few popular middleware frameworks, such as ASP.NET,[31] Ruby on Rails,[32] or NancyFx[33] (which can run standalone as a server or under IIS as middleware), you'll immediately get a sense that there is a sophisticated architecture supporting the web server. There's also some very clever built-in functionality that doesn't have anything to do with handling requests, but tends to make the job easier because there's a typical set of common tasks people need to perform when creating a professional website.

If you use any of these frameworks, you will almost immediately notice one or more of the following characteristics:

- Either enforces or at least defaults to creating a project with an MVC architecture.

---

[24] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[25] http://en.wikipedia.org/wiki/Model_View_ViewModel
[26] http://en.wikipedia.org/wiki/Representational_state_transfer
[27] http://en.wikipedia.org/wiki/Ajax_%28programming%29
[28] http://en.wikipedia.org/wiki/AJAJ
[29] http://en.wikipedia.org/wiki/SOAP
[30] https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations
[31] http://www.asp.net/
[32] http://rubyonrails.org/
[33] http://nancyfx.org/

- Has some sort of a view engine for rendering dynamic content at the server before the browser sees the final page, such as ASP.NET's Razor,[34] ASPX view engines, or NancyFx's SuperSimpleViewEngine.[35] Rails supports a wide range of view (also known as "template") engines.[36]
- Possibly includes some sort of Object-Relational Mapper (ORM). In the ASP.NET world, this is usually Entity Framework;[37] in Rails we find ActiveRecord.[38]

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

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

Note that in the web server implementation presented in this book, the MVC pattern is not baked into the architecture—you are free to use an MVC pattern or not for handling web requests.

Dynamic versus Static Content and the Single-Page Paradigm

The trend (especially as "push servers;" see SignalR[39]) 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.

---

[34] http://en.wikipedia.org/wiki/ASP.NET_Razor_view_engine
[35] https://github.com/grumpydev/SuperSimpleViewEngine
[36] https://www.ruby-toolbox.com/categories/template_engines
[37] http://en.wikipedia.org/wiki/Entity_Framework
[38] http://guides.rubyonrails.org/active_record_basics.html
[39] http://signalr.net

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,[40] Knockout,[41] Backbone,[42] Angular,[43] or any number of available options. Furthermore, you're not just writing "render this page" server-side and 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 I have 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 of whether to build 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, my motivations are to write a web server that not only meets his or her needs but also employs an architecture that does not hinder you from meeting *your* needs. 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.

---

[40] http://api.jquery.com/
[41] http://knockoutjs.com
[42] http://backbonejs.org
[43] https://angularjs.org

Chapter 3 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.

The source code presented in this section is in the folder **Examples\Chapter 3\Demo-AsyncAwait** in the Bitbucket repository.

Multiple Listeners

Let's look at instrumenting the **`StartConnectionListener`** function in the previous code so that we can get a sense of the processing times and threads. First, we'll add a couple basic instrumentation functions in the **`Program`** class:

```
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);
}
```

Code Listing 6

Next, we add the instrumentation to the **`StartConnectionListener`**, replacing the previous method with information on when and what thread the listener starts on. I also have replaced the handling of the response with a common "handler" object (described next).

```
/// <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);
}
```

Code Listing 7

Recall that these listeners are all initialized on a separate thread, but as noted previously, we let the .NET framework allocate a thread on the continuation. Here again is the code from Chapter 2 that initializes the listeners:

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

Code Listing 8

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);
  }
}
```

Code Listing 9

CommonResponse (a method of **ListenerThreadHandler**) artificially injects a one-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();
}
```

Code Listing 10

The handler object is instantiated in the **Main**:

```
static void Main(string[] args)
{
  // Supports 20 simultaneous connections.
  sem = new Semaphore(20, 20);
  handler = new ListenerThreadHandler();
…etc…
```

Code Listing 11

After initializing the listeners, we'll add a test to **Main** to see how the server responds to 10 effectively simultaneous, asynchronous requests:

```
TimeStampStart();

for (int i = 0; i < 10; i++)
{
  Console.WriteLine("Request #" + i);
```

```
  MakeRequest(i);
}
```

Code Listing 12

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);
}
```

Code Listing 13

RequestIssuer is an “awaitable” request and response function, meaning that it will issue a web request and return to the caller while awaiting the response. The response is handled in the **await** continuation:

```
public class RequestIssuer
{
  public static async Task<string> HttpGet(string url)
  {
    string ret;

    try
    {
      HttpWebRequest request = (HttpWebRequest)HttpWebRequest.Create(url);
      request.Method = "GET";

      using (WebResponse response = await request.GetResponseAsync())
      {
        using (StreamReader reader = new
StreamReader(response.GetResponseStream()))
        {
          ret = await reader.ReadToEndAsync();
        }
      }
    }
    catch (Exception ex)
    {
      ret = ex.Message;
```

```
    }

    return ret;
  }
}
```

Code Listing 14

In the previous 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 want to know 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 since 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
```

Code Listing 15

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
```

Code Listing 16

What's revealing here is that:

- The requests appear to be processed in batches of four (the computer I'm testing on has four 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" four 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
```

Code Listing 17

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

Why Async/Await is Not the Right Solution

From the previous 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

The source code presented in this section is in the **Examples\Chapter 3\Demo-Threading** folder in the Bitbucket repository.

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, replacing **`TaskRun`** and semaphores with the creation of 20 listener threads:

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

Code Listing 18

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);
  }
}
```

Code Listing 19

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
```

Code Listing 20

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
```

Code Listing 21

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

What about ThreadPool?

The source code presented in this section is in the **Examples\Chapter 3\Demo-ThreadPool folder** in the Bitbucket repository. But is the problem with **`async/await`** or the system **`ThreadPool`**? Using a **`ThreadPool`** is not ideal because we're implementing long-running threads, but we'll try it regardless:

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

Code Listing 22

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
```

Code Listing 23

We certainly experience what the MSDN documentation says regarding **ThreadPool**: “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 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
```

Code Listing 24

So, while they work, thread pools are also not the correct solution. And as the MSDN documentation indicates, a thread pool is not the right solution here because 1) we’re creating a number of threads in a very short time, and 2) 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—they are not short-lived threads.

Conclusion

It is now very clear that we should not use **`async`**/**`await`** to implement asynchronous connection requests. **`Async`**/**`await`** limits you to processing requests based on the number of cores, preventing you (and the CPU) from distributing request processing across more threads than you have cores. This will definitely be an issue, as it is common to query a database or third-party social media API in your request handler, and your thread will for the most part be waiting for a response, which should not stop other requests from being handled.

Single Thread Listener

The source code presented in this section is in the folder **Examples\Chapter 3\Demo-SingleThreadListener** in the Bitbucket repository.

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 de-queued and en-queued into a worker thread. We can implement different algorithms for determining which worker thread to en-queue the request, but in the implementation that follows, we use a simple round-robin algorithm.

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);
  }
}
```

Code Listing 25

Note the use of .NET's concurrent collection class, **`ConcurrentQueue`**, in Code Listing 25. These are high-performance collections that handle concurrent read/writes and alleviate the complexity of us having to write thread-safe collections.

Instead of processing the request immediately, our handler queues the request and returns. A separate thread de-queues the request and assigns it, round-robin, to a worker thread.

```
public class SingleThreadedQueueingHandler
{
  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);
      }
    }
  }
}
```

Code Listing 26

The result is what we should expect—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
```

Code Listing 27

Conclusion

The advantage of the single-threaded connection queuing 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 all the worker threads become busy. 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.

Chapter 4 Thread-Spanning Workflows

The source code presented in this section is in the folder **Examples\Chapter 4** in the Bitbucket repository. The Visual Studio solution file is in the **Chapter 4\Clifton.WebServer** folder.

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 checks
- Routing
- Rendering (i.e. a view engine)

Therefore, we'll look at requests as sequential workflows and implement them so 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:

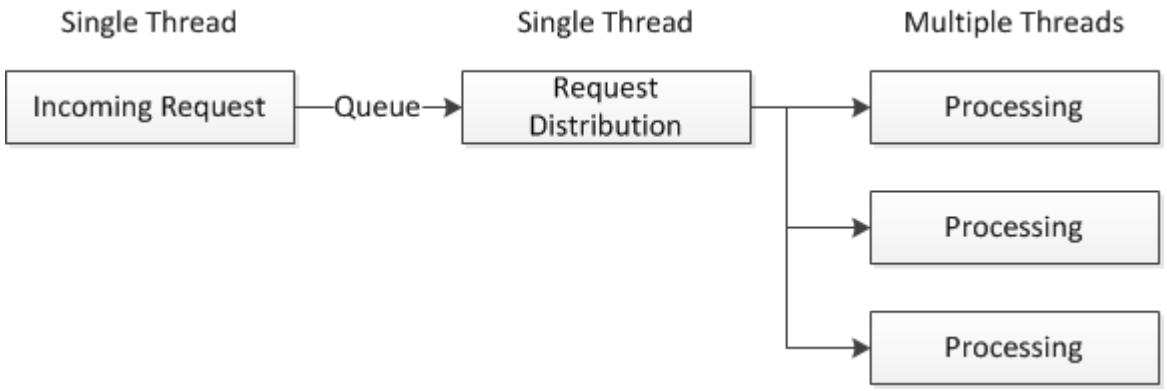


Figure 4: High-Level Workflow

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

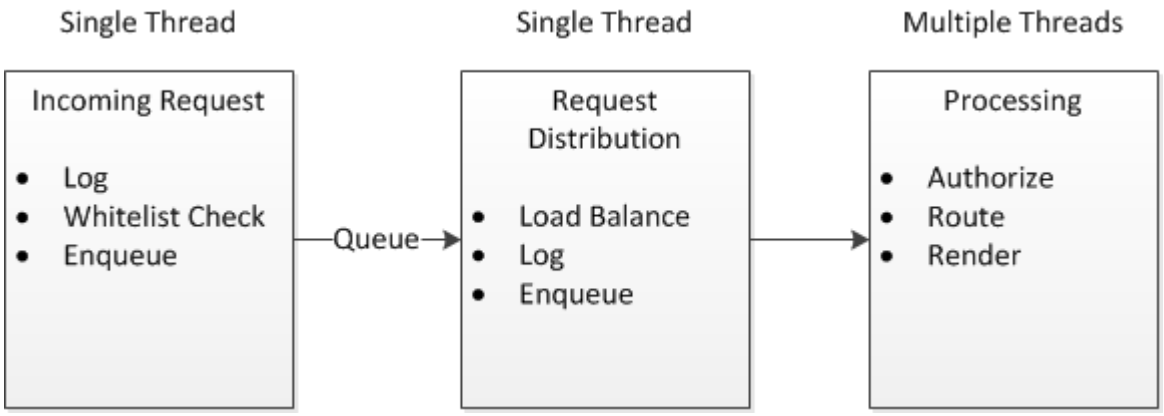


Figure 5: Low-Level Workflow

A thread-spanning workflow abstraction gives us is the following:

- The ability to define workflows declaratively.
- The ability to decouple the thread from the work implementation.
- The allowance of the work implementation to determine how work should be continued: on the same thread, or 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 WorkflowState
{
  /// <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,
}
```

Code Listing 28

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. Defines a single instance of a particular workflow pattern.
2. Uses that instance simultaneously.

We are effectively implementing continuation-passing style—we are passing in the continuation state to each workflow function. The workflow, as a process, is thread-safe, even though we are sharing instances among 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;
  }
}
```

Code Listing 29

WorkflowItem

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);
  }
}
```

Code Listing 30

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)
  {
    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 &&
!wc.Done)
    {
      WorkflowState state = items[wc.WorkflowStep++].Execute(wc, data);

      switch (state)
      {
        case WorkflowState.Abort:
          wc.Abort = true;
```

```
          break;

        case WorkflowState.Defer:
          wc.Defer = true;
          break;

        case WorkflowState.Done:
          wc.Done = true;
          break;
      }
    }
  }
}
```

Code Listing 31

Putting It All Together

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

1. Logs the incoming IP address and webpage request.
2. Checks that the requester's IP address is on our whitelist.
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>( StaticResponse));
```

Code Listing 32

And the logging and white list 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") ||
url.StartsWith("[::1]");
  WorkflowState ret = valid ? WorkflowState.Continue : WorkflowState.Abort;

  return ret;
}
```

Code Listing 33

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;
}
```

Code Listing 34

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"}}
};
```

Code Listing 35

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;
}
```

Code Listing 36

Here we see the result of querying our server:

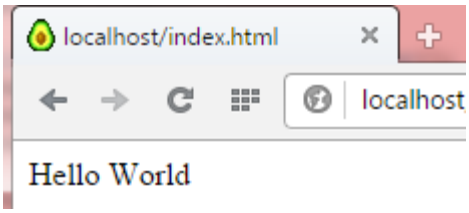


Figure 6: Result of a Workflow

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);
```

Code Listing 37

We'll also refactor the **`Workflow<T>`** class:

```
…
public Action<T> AbortHandler { get; protected set; }
public Action<T, Exception> ExceptionHandler { get; protected set; }

public Workflow(Action<T> abortHandler, Action<T, Exception>
exceptionHandler)
{
  items = new List<WorkflowItem<T>>();
  AbortHandler = abortHandler;
  ExceptionHandler = exceptionHandler;
}
…
```

Code Listing 38

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;
    }
  }
}
```

Code Listing 39

And we can write a couple of handlers—our abort handler 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();
}
```

Code Listing 40

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

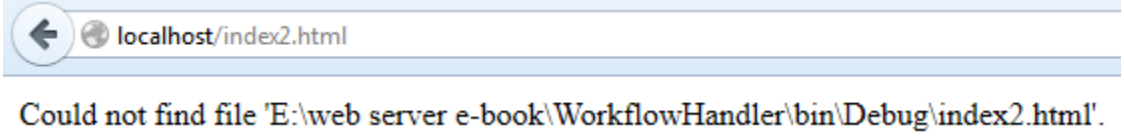


Figure 7: Error Handling Example

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 in this 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();
  }
}
```

Code Listing 41

Chapter 5 Routing

The source code presented in this section is in the folder **Examples\Chapter 5** in the Bitbucket repository. The Visual Studio solution file is in the **Chapter 5\Clifton.WebServer** folder.

Now it's time to talk about routing. The previous examples are still for a static web server—we have no way of hooking into page requests, and equally important, doing different things based on the verb used in the request, which is vital for supporting AJAX and REST APIs.

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 as other factors, such as whether the session has expired or not.

Having worked with the previous two approaches, as well as 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 an object-oriented or attribute-decoration approach.

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 are 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 publicly accessible, or privately accessible with the right role. Most, if not all, private pages can be expired.

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 previously, a route entry will consist of three “providers”:

- **SessionExpirationProvider**
- **AuthorizationProvider**
- **RoutingProvider**

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

```
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;
}
```

Code Listing 42

Note that the provider functions have the signature of a workflow process.

We’ll cover **Session** in the next chapter.

A Route Key

We also need a route “key,” which is the lookup key for the route dictionary—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;
  }
}
```

Code Listing 43

A Route Table

A route table maps the routing key (the verb and path) with a route entry. To ensure thread safety, we use .NET's **`ConcurrentDictionary`**, even though technically, the route table should not be modified after initialization. However, 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 handler!

```
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 };
  }
}
```

Code Listing 44

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;
  }
}
```

Code Listing 45

Remember, we'll look at sessions and session management in the next chapter, so 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);
}
```

Code Listing 46

Here we're leveraging the previously implemented exception handler to display the message in the browser window. When we request this page (via **get**), 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));
}
```

Code Listing 47

And voilà!

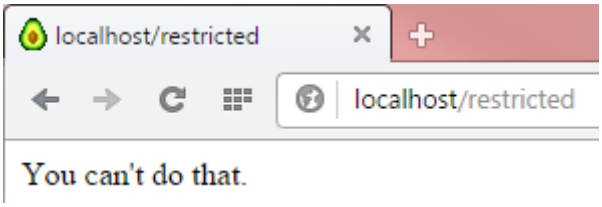


Figure 8: Routing Example

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 Chapter 10, "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 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, as I did previously, 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.

Chapter 6 Sessions

The source code presented in this section is in the folder **Examples\Chapter 6** in the Bitbucket repository. The Visual Studio solution file is in the **Chapter 6\Clifton.WebServer** folder.

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, a user name, the last request time, and so forth. These pieces of information are all managed in a stateful session that is associated with the user's IP address.

The session management provided here is really a basic implementation, and I certainly don't want to presume what your authorization, session expiration, and user role management needs might be. As we saw in the chapter covering 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 following implementation, note that the session provides three things:

- A way to manage whether the session has expired or not.
- A way to manage whether the user is authorized or not.
- A general collection of key-value pairs that the 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;
  }
}
```

Code Listing 48

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 set to 10 minutes by default.

```
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;
  }
}
```

Code Listing 49

CSRF Token

For new sessions, a CSRF token is registered. We can use this token when we render pages, embedding it into **`put`**, **`post`**, and **`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 **`sessionManager`** instance and add it to the workflow:

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

Code Listing 50

Next we set up a couple of webpages 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;
  }
});
```

Code Listing 51

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;
  }
});
```

Code Listing 52

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;
}
```

Code Listing 53

We'll first test a non-expired, authorized site, which gives us back:

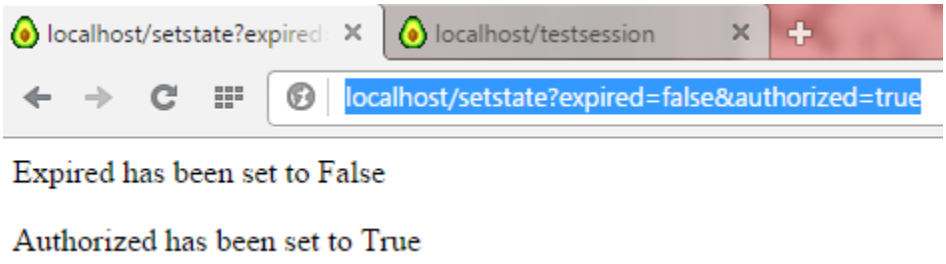


Figure 9: Not Expired, Authorized

Now when we test our state with the **`testsession`** URL, we get:

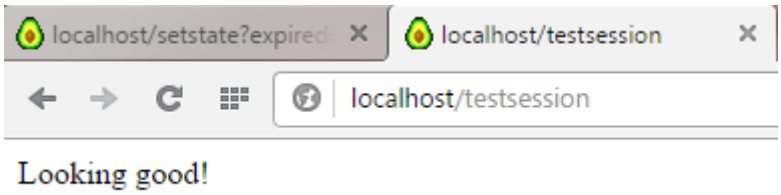


Figure 10: Looking Good!

We can expire the session:

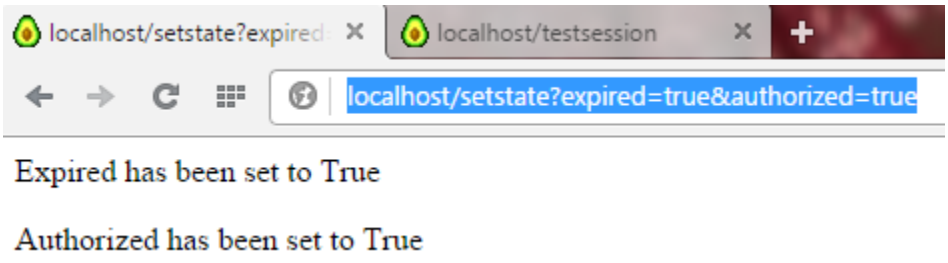


Figure 11: Expire the Session

Which gives us:

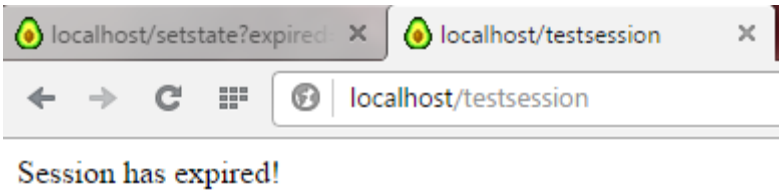


Figure 12: Session Has Expired

Lastly, we can de-authorize the session:

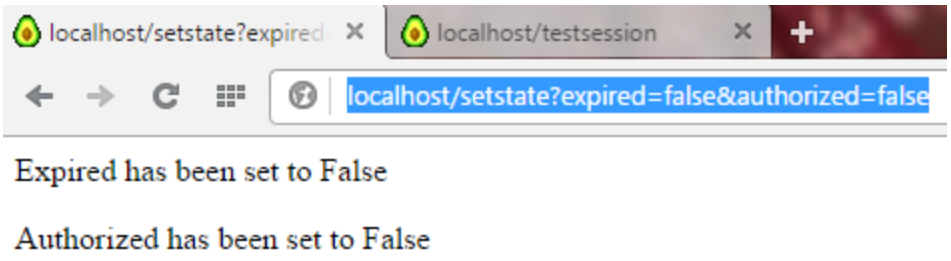


Figure 13: De-authorize the Session

And we get:

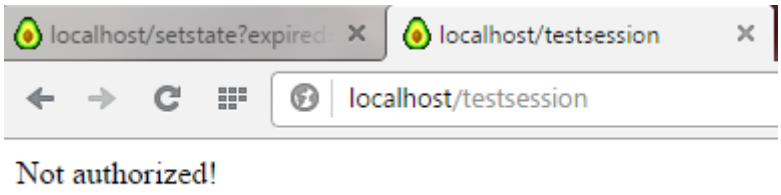


Figure 14: The Session is No Longer Authorized

Automatically Cleaning Up Expired Sessions

It's important that we clean up expired sessions. To do that the question becomes: when do we really delete any knowledge of the session, versus potentially giving the user some feedback, such as "your 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));
}
```

Code Listing 54

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 previous 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 of anonymous methods for every route handler—it 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,
    });
  }
}
```

Code Listing 55

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);
```

Code Listing 56

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.
- Respond with a default page, custom HTML, and/or a custom response body.
- Handle REST endpoint calls.

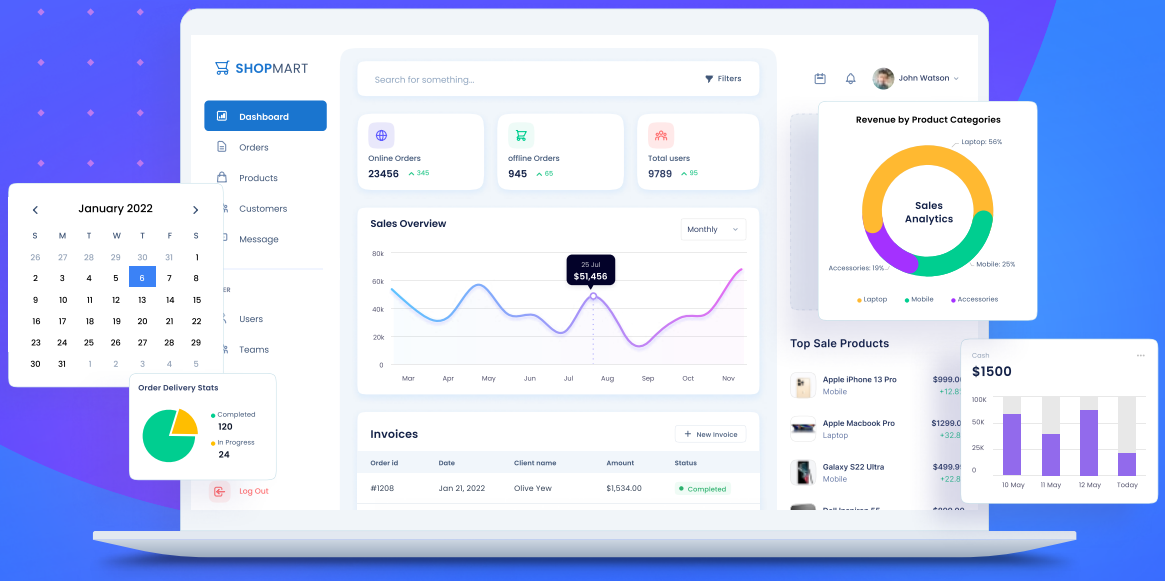
However, there are still a few things left to do, such as:

- Parameterized routes (routes with IDs 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.

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

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Apple IBM SIEMENS intel VISA

Syncfusion

Chapter 7 HTTPS

Chapter 7 does not have a source code example in the repository, as this chapter deals mainly with setting up an SSL certificate.

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 nonetheless, 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:

- Domain Validation
- Organization Validation
- Extended Validation

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

Domain Validation

Domain Validation (DV) establishes 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 it is easily obtainable and only secures the communication between your browser and the server.

Organization Validation

Organization Validation (OV) 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

The Extended Validation (EV) level of certification requires the company to go 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 for testing though, 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 **All Programs** on the **Start** menu. Choose **Microsoft Visual Studio 2012**, click **Visual Studio Tools**, and choose **Developer Command Prompt for VS2012**.

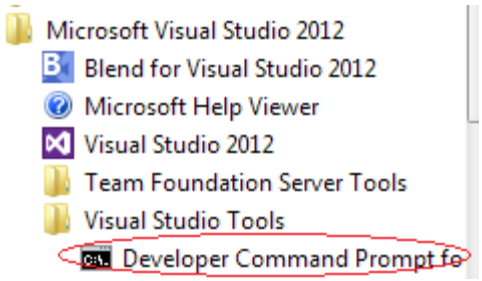


Figure 15: Developer Command Prompt

At the console window, type the following (all on one line), but replace `[computername]` with either your computer name or a domain name. You can read more about **`makecert`** options on MSDN.

```
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
```

Code Listing 57

For example (I'm using **`"test"`** and the computer name):

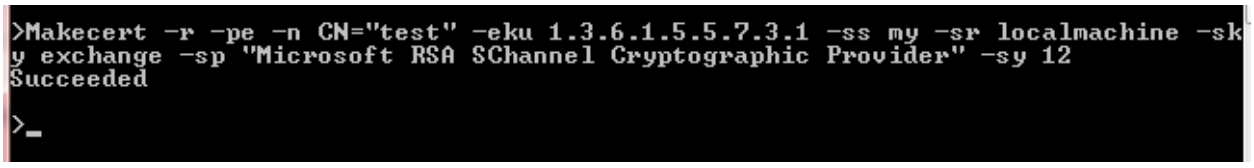


Figure 16: Makecert Example

Make the Certificate Trusted

From the console window, 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**.

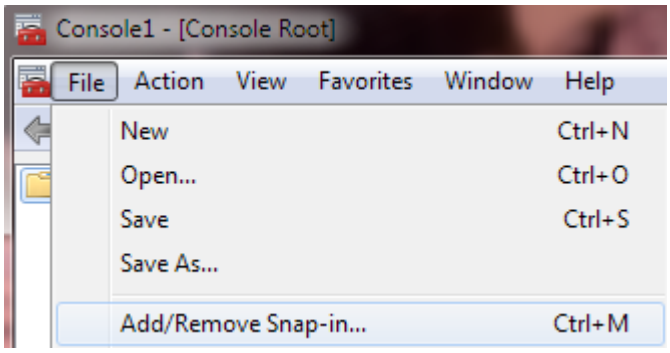


Figure 17: Add Certificates Snap-in

Select **Certificates** and click **Add**.

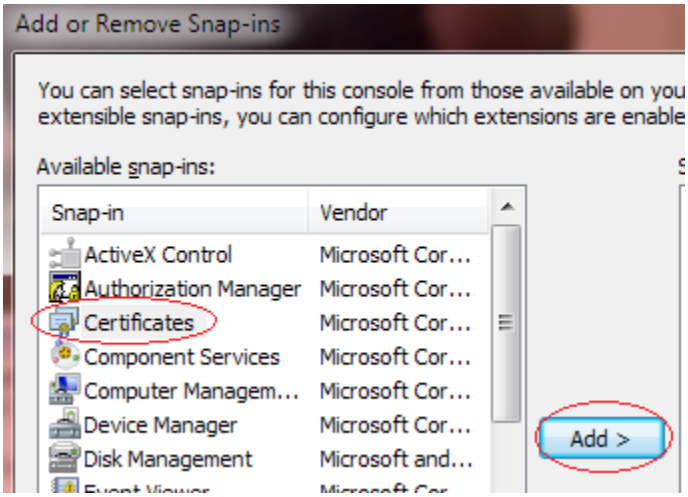


*Figure 18: Select **Certificates***

Select **Computer account**.

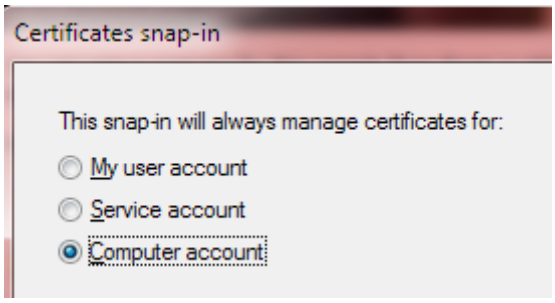


*Figure 19: Select **Computer account***

Then click **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."

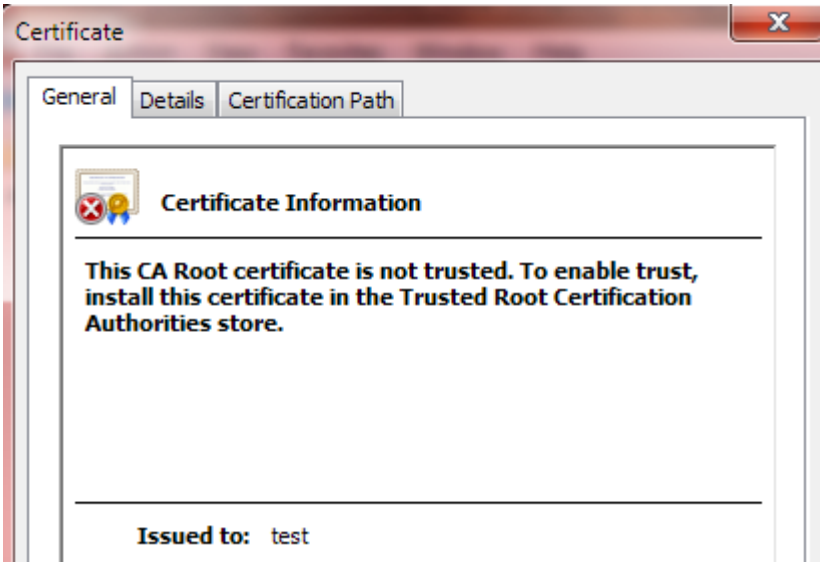


Figure 20: Certificate Information

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:

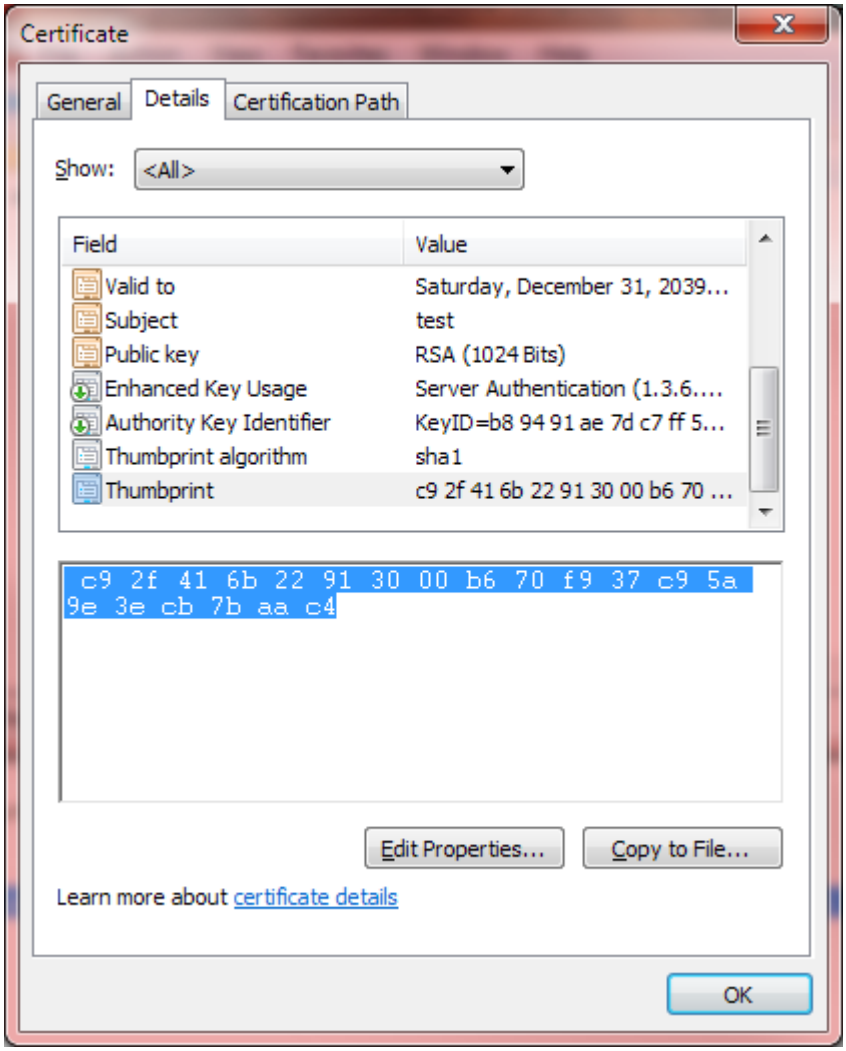


Figure 21: Certificate Thumbprint

Use Notepad's Search and Replace functionality to remove all the whitespace:

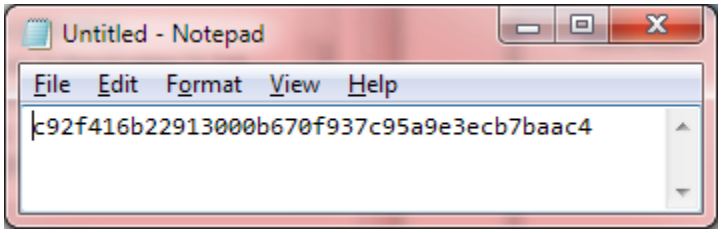


Figure 22: Remove Whitespace

Copy the Certificate to the Trusted Root Certification Authorities Folder

1. In the same **Details** tab, click **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 file name for your certificate, such as **c:\temp\mycert.cer**.
6. Click **Finish**.
7. Close the Certificate dialog by clicking **OK**.

Now, open the **Trusted Root Certification Authorities\Certificates** folder.

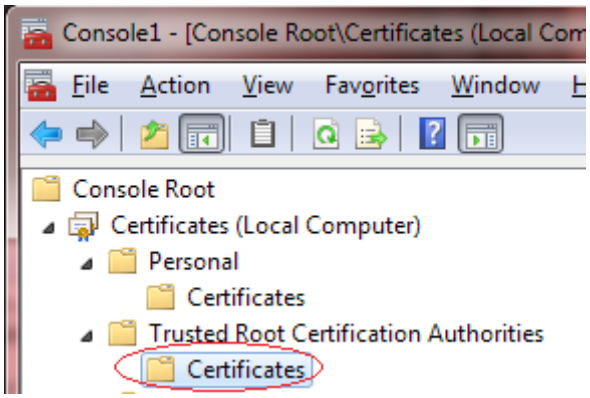


Figure 23: Certificates Folder

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

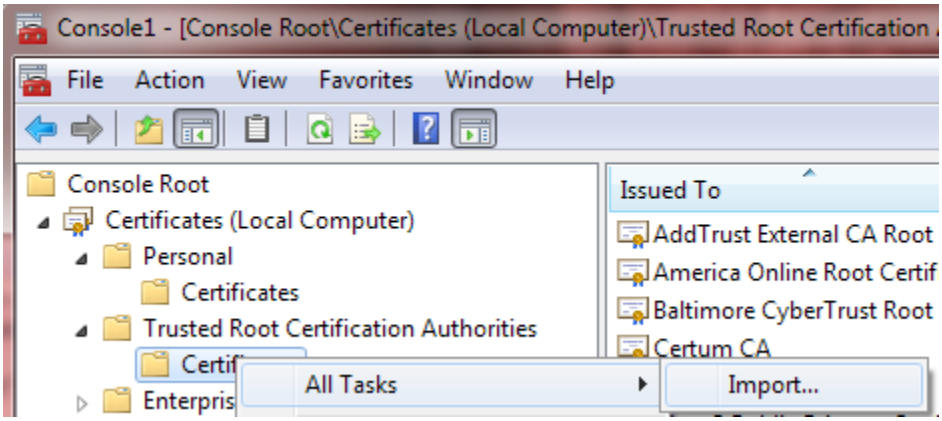


Figure 24: Import Certificate

1. Click **Next**.
2. Enter the file name of the certificate you just exported.
3. Click **Next**.
4. Click **Next** (Place all certificates in the following 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):**".

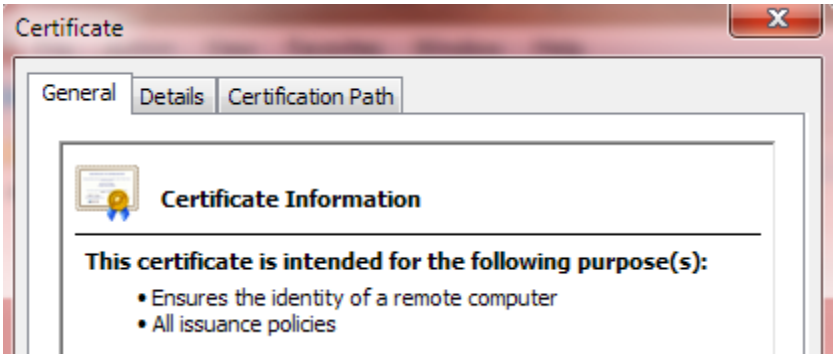


Figure 25: Verify Certficate is Trusted

Bind the Certificate to All IP Addresses and Port on the Machine

In the VS2012 command line console window we opened earlier, use **`netsh`** (you can read more about it on MSDN), bind the certificate to all IP addresses and the SSL port, replacing **`[yourhash]`** with the thumbprint we obtained:

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

Code Listing 58

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, your **`netsh`** command, using the thumbprint we acquired 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}
```

Code Listing 59

That's All

Your computer is now ready to respond to HTTPS web requests (assuming 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:

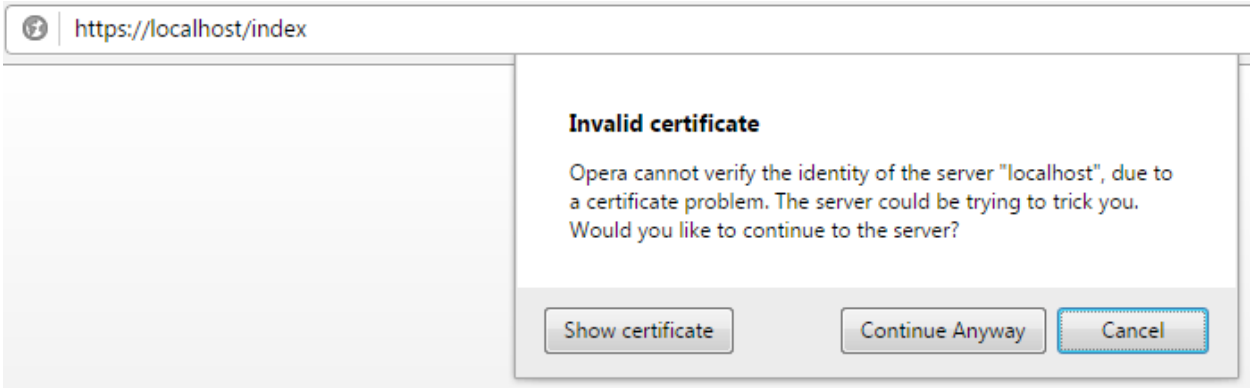


Figure 26: Certificate Warning

To avoid this warning, you would have to obtain a certificate from a trusted authority, such as Verisign. 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/");
```

Code Listing 60

Conclusion

As stated at the beginning of this chapter, HTTPS is a requirement today for all web servers—not just for handling credit card information (if you're putting together a merchant website, for example), but ideally for handling all data transactions between the browser and the server. However, using HTTPS is not the end of the picture when it comes to securing your user's data: for example, you might consider encrypting sensitive data, or if the data never needs to be decrypted (passwords are a good example), you might use a one-way hash. There are also legal requirements that you need to be familiar with, depending on the kinds of data your website handles.

It cannot be overemphasized that security should be the first consideration of a website, which includes all levels in which the data can be intercepted, starting with the browser, continuing with the transport layer, and ending with the server and any persistent store of sensitive information.

Chapter 8 Error Handling and Redirecting

The source code presented in this section is in the folder **Examples\Chapter 8** in the Bitbucket repository. The Visual Studio solution file is in the **Chapter 8\Clifton.WebServer** folder.

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

"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 **`restricted`** and **`testsession`** routes that we created 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;
  }
});
```

Code Listing 61

You'll note in the previous code listing 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();
}
```

Code Listing 62

Notice in the previous code listing 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
```

Code Listing 63

When we navigate to **`localhost/testsession`**, we see this:

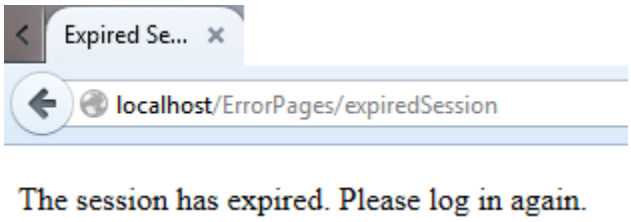


Figure 27: Redirecting

Similarly, we'll set the session state to **`unauthorized`**:

```
http://localhost/setstate?expired=false&authorized=false
```

Code Listing 64

And we see:

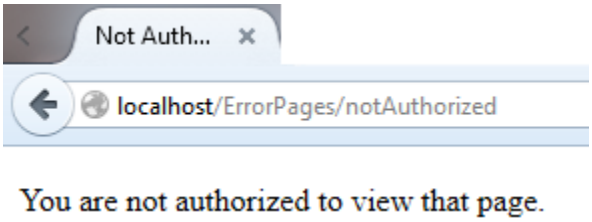


Figure 28: Redirect

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");
  }
}
```

Code Listing 65

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

Logging Services

You may want to consider a logging service such as PaperTrail, which I've written about on Code Project. Sending a UDP message to PaperTrail is fast and easy:

```
private static void SendUdpMessage(
  IPAddress address,
  int port,
  string message)
{
  Socket socket = new Socket(
           AddressFamily.InterNetwork,
           SocketType.Dgram,
           ProtocolType.Udp);
  IPEndPoint endPoint = new IPEndPoint(address, port);
  byte[] buffer = Encoding.ASCII.GetBytes(message);
  socket.SendTo(buffer, endPoint);
  socket.Close();
}
```

Code Listing 66

The log can be viewed in your browser, appearing similar to this:

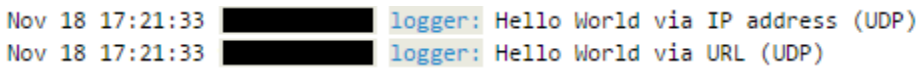


Figure 29: Example Paper Trail Log Message

PaperTrail complies with the Syslog Protocol described in RFC-5424, so you can format your message using this protocol. For example:

```
logger.Log("<22>" +
    DateTime.Now.ToString("MMM d H:mm:ss") +
    " Marc Test: This is a test message");
```

Code Listing 67

This results in the following log entry (note the highlighting that PaperTrail does):

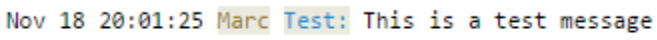


Figure 30: Syslog Protocol

Chapter 9 Parameterized Routes

The source code presented in this section is in the folder **Examples\Chapter 9** in the Bitbucket repository. The Visual Studio solution file is in the **Chapter 9\Clifton.WebServer** folder.

A common practice is to add parameters within a route. You'll see this used frequently in Ruby on Rails applications, though 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 (or may not) define 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, or indirectly through a model.

The issue can be considerably more complex. Consider the routing options that NancyFx 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 want for how parameters in the path are specified. For example, we might require a form like this (used by Rails):

```
param/:p1/subpage/:p2
```

Code Listing 68

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

```
param/{p1}/subpage/{p2}
```

Code Listing 69

Handling IDs

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))
```

Code Listing 70

It's 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);
}
```

Code Listing 71

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>
{
}
```

Code Listing 72

We refactor the **`GetRouteEntry`** methods to a form similar to this (not all overloads are 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;
}
```

Code Listing 73

We implement a simple parser that iterates through the routes, and finds the first one that matches. This method has two parts: the iterator and the matcher. First, 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;
}
```

Code Listing 74

Followed by the matcher (note how we could add additional behaviors here for matching a capture segment should we wish to):

```
/// <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 (also known as a "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;
}
```

Code Listing 75

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;
  }
});
```

Code Listing 76

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

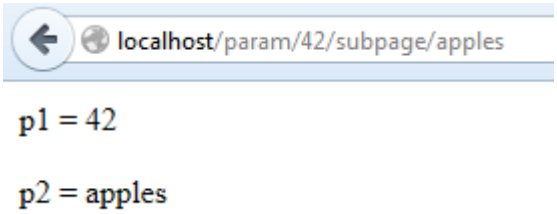


Figure 31: Path Parameters

Notice how we don't care about the parameter type: it can be an integer, a float, or 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 URLs.

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.

Chapter 10 Form Parameters and AJAX

The source code presented in this section is in the folder **Examples\Chapter 10** in the Bitbucket repository. The Visual Studio solution file is in the **Chapter 10\Clifton.WebServer** folder.

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 an 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: 
            <input name="username"/>
        </div>
        <div class="center-inner top-margin-10">
            Password: 
            <input type="password" name="password"/>
        </div>
        <div class="center-inner top-margin-10">
            <input type="submit" value="Login"/>
        </div>
    </form>
</body>
</html>
```

Code Listing 77

It's 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;
  }
});
```

Code Listing 78

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

When we click on the **Login** button:

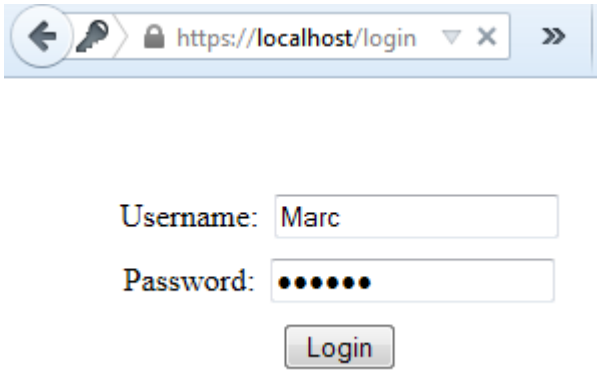


Figure 32: Login

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

	Value
data	"username=Marc&password=fizbin"

Figure 33: 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: 
        <input id="username"/>
    </div>
    <div class="center-inner top-margin-10">
        Password: 
        <input type="password" id="password"/>
    </div>
    <div class="center-inner top-margin-10">
        <input type="submit" value="Login" id ="btnLogin"/>
    </div>
</body>
</html>
```

Code Listing 79

Here the input stream's parameters are exactly the same as in the from POST request, and are 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()
}),
```

Code Listing 80

We now need a new route handler for the same path (because 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;
  }
});
```

Code Listing 81

Here we note that we receive a JSON string:

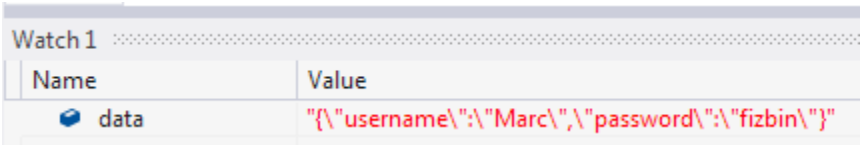


Figure 34: JSON String

The point of bringing this up is that an issue such as using a form post versus 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.

Chapter 11 View Engines

The source code presented in this section is in the folder **Examples\Chapter 11** in the Bitbucket repository. The Visual Studio solution file is in the **Chapter 11\Clifton.WebServer** folder.

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 simply be REST APIs that support the client. This is often the ideal approach for Single Page Applications (SPAs).[44] 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 postback. 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 find on Google lots of articles on how to convert an ASP.NET MVC application to an SPA application.

That said, I don't want to impose my opinion on the user, so supporting a view engine such as Razor[45] should definitely be possible in the web server. Now, Razor (like other view engines such as nHaml[46] and Spark[47]) are somewhat weighty in that they will actually create an on-the-fly code file that must be compiled at runtime 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, [48] 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, check out Rick Strahl's Westwind.RazorHosting[49] as well.

Note that special keywords like @Html and @Url are actually not part of Razor, but are implementation details of the MVC and WebPages frameworks, so the functions they implement are not available in the templating engine.

In order to build the code for this chapter, I suggest cloning the repository from the Bitbucket site and building the code directly. Installing from the NuGet package will probably result in a conflict between the RazorEngine version and the System.Web.Razor version. The code example in the **Chapter 11** folder includes a pre-built version of RazorEngine.Core.dll, but again, depending on the version of .NET's System.Web.Razor, the version that I built for this chapter might result in a "different version" compiler error when you try to build the code. For that reason, always reference the RazorEngine built on from the source code rather than the NuGet package.

---

[44] SPA and the Single Page Myth
[45] http://en.wikipedia.org/wiki/ASP.NET_Razor_view_engine
[46] http://code.google.com/p/nhaml/
[47] https://github.com/SparkViewEngine/spark
[48] https://antaris.github.io/RazorEngine/index.html
[49] https://github.com/RickStrahl/Westwind.RazorHosting

First, Some Refactoring

In the code presented previously, the workflow effectively stops once the page file is loaded or the route handler is invoked. This was a 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 };
  }
}
```

Code Listing 82

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));
}
```

Code Listing 83

The responder is implemented as follows:

```
/// <summary>
/// The 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;
}
```

Code Listing 84

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;
```

Code Listing 85

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;
}
```

Code Listing 86

Here we initialize a template key that 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[50] 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));
}
```

Code Listing 87

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>
```

[50] http://antaris.github.io/RazorEngine/

```
@for (int i=0; i<10; i++)
{
  <li>@i</li>
}
  </ul>
</body>
</html>
```

Code Listing 88

Here's the result:

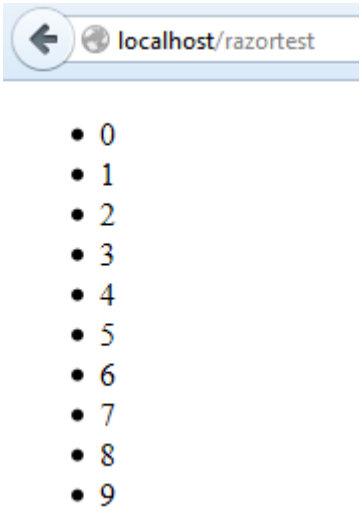


Figure 35: Razor View Engine

Great! We've added a sophisticated view engine to our web server. Because the view engine is a workflow step, the implementation is actually done in our web application rather than in the web server, leaving our web server un-opinionated with regard to the view engine being used. One thing you may notice, however, is that the responsiveness of the webpage is degraded.

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 webpage that displays data from many different models. In classic MVC, the workaround to this is to create a "View Model" (hence we're actually implementing the Model-View-ViewModel, or MVVM pattern). This is, at best, an awkward workaround, but it is what we have to live with in regard 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 ...
```

Code Listing 89

We'll use this collection as the model:

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

Code Listing 90

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>
```

Code Listing 91

We get a partial screenshot (apologies if your name is not on this screenshot):

Figure 36: Razor Model

Now the question becomes, how do we associate the HTML page (the view!) with the desired model that we want to pass in to the view engine?

While the answer is left as an exercise for the reader (as it really is outside the purview of a web server not entangled with MVC or MVVM patterns), the simplest answer that I have is to write a route handler for each template page that acts as a “controller” and instantiates the desired model. With a little refactoring, the model can then be assigned to the **`PendingResponse`** instance and used by the view engine workflow step, giving you something that very closely resembles the existing MVC paradigms in ASP.NET and Rails.

CSRF

The interesting thing about “owning” the web server is you can do basically whatever you want in the HTML before you send it down to the browser. For example, in ASP.NET MVC, you specify a CSRF token in a form like this:

```
@Html.AntiForgeryToken()
```

Code Listing 92

In Rails, the **`ApplicationController`**, if it’s included:

```
protected_from_forgery
```

Code Listing 93

This will automatically reset the session if the CSRF token does not match. One then adds the following into the **`head`** section of the application layout page:

```
<%= csrf_meta_tag %>
```

Code Listing 94

As you can see, there is no standard for how this token should be handled. In the chapter on sessions, we’re actually creating the token for a new session. Instead of using a heavy-weight view engine, we can do a simple search and replace for this token in the HTML. For example, in our earlier login page, we could add our own keyword **`%AntiForgeryToken%`** to be replaced with a hidden field containing the token value:

```
<body>
    <form name="myform" action="/login" method="post">
        %AntiForgeryToken%
        <div class="center-inner top-margin-50">
            Username: 
            <input name="username"/>
        </div>
        <div class="center-inner top-margin-10">
            Password: 
            <input type="password" name="password"/>
        </div>
        <div class="center-inner top-margin-10">
            <input type="submit" value="Login"/>
        </div>
    </form>
</body>
```

Code Listing 95

Instead of (or in addition to, if you're using the view engine) calling the **ViewEngine** workflow step, we can call a new method, **CsrfInjector**, as part of the workflow:

```
workflow.AddItem(new WorkflowItem<ContextWrapper>(CsrfInjector));
```

Code Listing 96

Implemented as:

```
public static WorkflowState
CsrfInjector(WorkflowContinuation<ContextWrapper> workflowContinuation,
ContextWrapper wrapper)
{
PendingPageResponse pageResponse = wrapper.PendingResponse as
PendingPageResponse;
if (pageResponse != null)
{
pageResponse.Html = pageResponse.Html.Replace("%AntiForgeryToken%", "<input
name=" + "csrf".SingleQuote() +
" type='hidden' value=" +
wrapper.Session["_CSRF_"].ToString().SingleQuote() +
" id='__csrf__'/>");
}

return WorkflowState.Continue;
}
```

Code Listing 97

Note how we're actually injecting the markup to define a hidden field, **csrf**.

Now, when we log in with this page, the form's **POST** request will include the CSRF token:

```
"csrf=ca64e53c-a9c5-4fde-ba15-e2fad4a334b9&username=admin&password=admin"
```

Code Listing 98

For non-**GET** routes, we can make this a standard check as part of the route handler validation. If you want to put CSRF validation into an AJAX call (highly recommended), this should be done in the header of the request. For example:

```
headers: {
            'RequestVerificationToken': '%CsrfValue%'
         }
```

Code Listing 99

And we would modify the **CsrfInjector** to also replace these keywords:

```
pageResponse.Html = pageResponse.Html.Replace(
    "%CsrfValue%",
    wrapper.Session["_CSRF_"].ToString().SingleQuote());
```

Code Listing 100

As mentioned previously, there is no standard for how to do this. As with view engines, anyone who writes a web server is free to define how these special cases are handled. Except for significant changes in the pre-rendered HTML, such as when using HAML or SLIM, the syntax of the HTML is fairly portable between servers.

Chapter 12 Stress Testing

The source code presented in this section is in the folder **Examples\Chapter 12** in the Bitbucket repository. The Visual Studio solution file is in the **Chapter 12\Clifton.WebServer** folder.

Stress testing, or load testing, is a Pandora's box. Once you open it, questions arise regarding whether the test results are accurate, whether the test itself is correct, whether it's testing the right thing, and how to even understand the test results. Given that, we'll spend a little time exploring this rocky terrain.

First, let's create a minimal workflow for each request:

```
public static void InitializeWorkflow(string websitePath)
{
  StaticContentLoader sph = new StaticContentLoader(websitePath);
  workflow = new Workflow<ContextWrapper>(AbortHandler, OnException);

  workflow.AddItem(new WorkflowItem<ContextWrapper>(sph.GetContent));
  workflow.AddItem(new WorkflowItem<ContextWrapper>(Responder));
}
```

Code Listing 101

This is the workflow for a static page server. All it does is respond to the request with the content of a file associated with the URL. As it turns out, additional workflow routines such as routing have negligible impact on the performance tests (this is a clue to something!).

Here's my test code, intended to be able to hit the server from multiple threads. Replace the IP address with your server's IP address. I use an IP address instead of **`localhost`** because I want to run these tests on a separate machine.

```
class Program
{
  static int n = 0;

  static void Main(string[] args)
  {
    List<Thread> threads = new List<Thread>();

    for (int i = 0; i < 1; i++)
    {
      Thread thread = new Thread(new ThreadStart(RunForOneSecond));
      thread.IsBackground = true;
      threads.Add(thread);
    }
    threads.ForEach(t => t.Start());
```

```
    Thread.Sleep(1250);

    Console.WriteLine("Made {0} requests.", n);
    Console.WriteLine("Press ENTER to exit.");
    Console.ReadLine();
  }

  static void RunForOneSecond()
  {
    DateTime now = DateTime.Now;
    WebClient client = new WebClient();
    client.Proxy = null;

    try
    {
      while ((DateTime.Now - now).TotalMilliseconds < 1000)
      {
        Interlocked.Increment(ref n);
        string downloadString =
client.DownloadString("http://192.168.1.21/");
      }
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

Code Listing 102

We'll start with one thread making requests to the server:

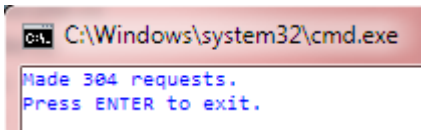


Figure 377: Single Thread Load Test

These numbers, by the way, are quite consistent. Let's try two threads:

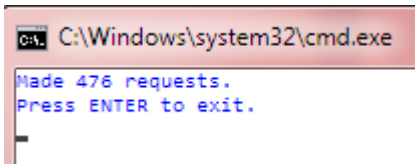


Figure 388: Two Threads Doing Load Testing

Basically, the number of requests that can be processed peaks around four load testing threads (I have eight cores on the test machine, by the way) and with a maximum throughput of around 600 requests per second. Incidentally, these numbers don't change regardless of what technique is used for listening to a request or how many listeners are listening.

But what are we actually measuring? Certainly we can say we're measuring the entire request flow, from initiating the request to receiving the response. There's a lot in the middle here. Let's perform instrumentation (without using workflows, so we have minimal impact) on the request-received to response-given time so we can see how much time is actually spent in the workflow itself. Note how we start right after obtaining a context, and we stop the clock right before sending the context. This eliminates all the .NET and operating system pieces before and after our server code. First we'll refactor the **`ContextWrapper`** to add a stopwatch that starts running the instant the wrapper is instantiated, which happens right after we receive a context.

```
public class ContextWrapper
{
  public HttpListenerContext Context { get; protected set; }
  public Response PendingResponse { get; set; }
  public Session Session { get; set; }
  public System.Diagnostics.Stopwatch Stopwatch { get; set; }

  public ContextWrapper(HttpListenerContext context)
  {
    Context = context;
    Stopwatch = new System.Diagnostics.Stopwatch();
    Stopwatch.Start();
  }
// ... etc ...
```

Code Listing 103

Then, in the responder, we'll stop the stopwatch and add the time to our cumulative count (I've verified that I'm using the high-resolution performance counter for the stopwatch):

```
public static WorkflowState Responder(
  WorkflowContinuation<ContextWrapper> workflowContinuation,
  ContextWrapper wrapper)
{
  wrapper.Stopwatch.Stop();
  Server.CumulativeTime += wrapper.Stopwatch.ElapsedTicks;
  ++Server.Samples;

// ... etc ...
```

Code Listing 104

Now, I'm going to add the router back into the workflow so we can display an average of the processing time through a URL, with:

```
routeTable.AddRoute("get", "loadtests", new RouteEntry()
{
  RouteHandler = (continuation, wrapper, session, pathParams) =>
  {
    long nanosecondsPerTick = (1000L * 1000L * 1000L) /
            System.Diagnostics.Stopwatch.Frequency;

    if (Server.Samples == 0)
    {
      wrapper.SetPendingResponse("<p>No samples!</p>");
    }
    else
    {
      long avgTime = Server.CumulativeTime * nanosecondsPerTick /
Server.Samples;
      string info = String.Format("<p>{0} responses, avg. response time =
{1}ns</p><p>Resetting sample info.</p>", Server.Samples,
avgTime.ToString("N0"));
      Server.CumulativeTime = 0;
      Server.Samples = 0;
      wrapper.SetPendingResponse(info);
    }

    return WorkflowState.Continue;
  }
});
```

Code Listing 105

Now we should see what our processing time inside the server is:

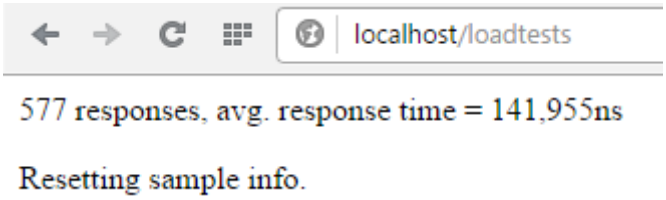


Figure 39: Internal Processing Time

So, let's call that 150 *microseconds* to process the request. Contrast this to our load test, which is telling us that an average request takes more than 1.5 *milliseconds.*

I interpret this to mean that the overhead of our load testing is 10 times the actual processing time of the request. I would make the conclusion then that our server, doing something rather minimal, could actually handle some 6,000 requests per second.

Let's try something different to vet our tests further. Instead of loading the index.html file, let's simply return that data in a specific route, and change our test to use that route:

```
routeTable.AddRoute("get", "sayhi", new RouteEntry()
{
  RouteHandler = (continuation, wrapper, session, pathParams) =>
  {
    wrapper.SetPendingResponse("<p>hello</p>");

    return WorkflowState.Continue;
  }
});
```

Code Listing 106

And, in our test program (you will have to change the IP address):

```
string downloadString = client.DownloadString("http://192.168.1.21/sayhi");
```

Code Listing 107

Now look at the results (the counts are off by two because we're also counting browsing to the **`loadtests`** page):

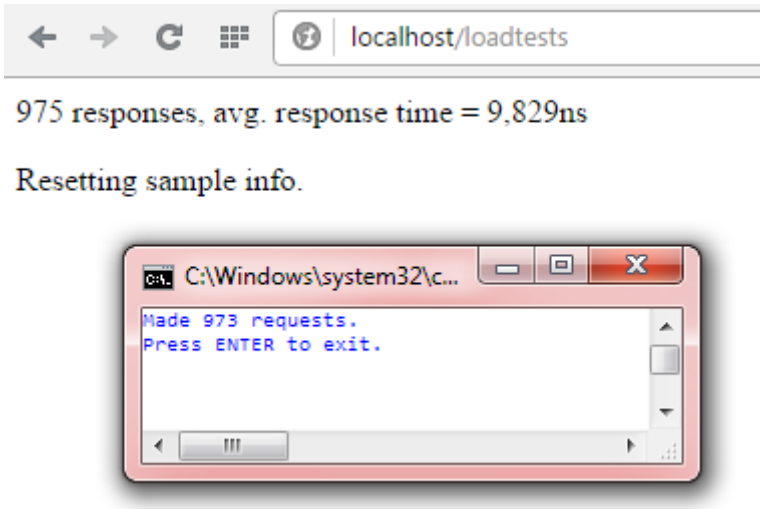


Figure 40: Load Test Without A File Read

Wow, we get a third more responses with four threads making queries, and the response time in our server is down to 9 *microseconds,* translating, at least in theory, to the ability to process more than 100,000 requests per second.

This gives you an idea of the overhead of….what? The operating system? The .NET framework? The test process? It's really hard to say.

What Can We Take Away From This?

The takeaway here is if you want a high-performance server, watch very carefully what your route handlers are doing, and your access to the file system, databases, and more. Lots of things in your application contribute to degrading the performance of the web server.

For example, adding the view engine to our last test resulted in an almost hundredfold increase in response time: 700 microseconds on average. When you consider the number of requests that may be coming in to your website, that adds up to a lot of potentially unnecessary overhead.

Conclusion

From my perspective, it's really interesting how one can write more than a hundred pages on less than 400 lines of code. That said, I hope I intrigued you in the nuances of writing a web server, especially with regard to how you choose an architecture, how you make it un-opinionated, and how complicated it is to measure the performance of the resulting work.

I hope it's also interesting to you to see what a non-MVC web application (at least, through my examples) would look like. It isn't necessary to buy into the MVC approach, but as I mentioned in the beginning of book, there aren't a lot of middle-ground web server applications one can turn to. If you haven't encountered it before, definitely take a look at NancyFx.[51] As the website puts it: "Nancy is a lightweight, low-ceremony framework for building HTTP-based services on .NET and Mono. The goal of the framework is to stay out of the way as much as possible and provide a super-duper-happy-path to all interactions." It's well worth a look.

As always, I look forward to reader feedback!

[51] https://github.com/NancyFx/Nancy/wiki/Introduction