



ASP.NET Multitenant Applications

Succinctly

by Ricardo Peres

ASP.NET Multitenant Applications Succinctly

By

Ricardo Peres

Foreword by Daniel Jebaraj



Copyright © 2015 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com upon completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Chris Tune

Copy Editor: Courtney Wright

Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	8
About the Author.....	10
Chapter 1 Introduction.....	11
About this book	11
What is multitenancy and why should you care?	11
Multitenancy requirements	12
Application services	13
Frameworks	13
Chapter 2 Setup.....	14
Introduction	14
Visual Studio	14
NuGet.....	14
Address mapping	16
IIS.....	16
IIS Express.....	18
Chapter 3 Concepts	20
Who do you want to talk to?.....	20
Host header strategy	21
Query string strategy.....	23
Source IP address strategy	24
Source domain strategy	29
Getting the current tenant	32
What's in a name?	33
Finding tenants	35

Tenant location strategies.....	35
Bootstrapping tenants.....	43
Chapter 4 ASP.NET Web Forms	47
Introduction	47
Branding.....	48
Master pages	48
Themes and skins.....	51
Showing or hiding tenant-specific contents	55
Security	57
Authorization	57
Unit testing	61
Chapter 5 ASP.NET MVC	62
Introduction	62
Branding.....	62
Page layouts	62
View locations	63
CSS Bundles.....	65
Security	67
Unit Testing.....	68
Chapter 6 Web Services	69
Introduction	69
WCF	69
Web API.....	70
Unit Testing.....	70
Chapter 7 Routing	71
Introduction	71
Routing.....	71

Route handlers.....	71
Route constraints	72
IIS Rewrite Module.....	73
Chapter 8 OWIN	77
Introduction	77
Registering services.....	77
Web API.....	79
Replacing System.Web.....	80
Unit testing	81
Chapter 9 Application Services	82
Introduction	82
Inversion of Control.....	82
Caching.....	84
Configuration.....	86
Logging	88
Monitoring	93
Tracing	94
Performance Counters.....	98
Health Monitoring.....	104
Analytics.....	110
Chapter 10 Security	115
Introduction	115
Authentication	115
ASP.NET Membership and Role Providers.....	115
ASP.NET Identity	119
HTTPS	121
Application pools.....	123

Cookies	123
Chapter 11 Data Access	124
Introduction	124
NHibernate	125
Entity Framework Code First	132
Chapter 12 Putting It All Together	136
Class model	136
Choosing the right tenant identification strategy.....	136
Choosing the right tenant location strategy	137
Choosing the right data access atrategy	138
Monitoring	139
Layout	140
References	141

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

Ricardo Peres is a Portuguese developer who has been working with .NET since 2001. He's a technology enthusiast and has worked in many areas, from games to enterprise applications.

His main interests nowadays are enterprise application integration and web technologies. He works for a London-based company called [Simplifydigital](#), and is a contributor to the NHibernate community.

Ricardo keeps a [blog on technical subjects](#) and can be followed on Twitter as [@rjperes75](#). Ricardo is a [Microsoft Most Valuable Professional](#) (MVP) in ASP.NET/IIS.

Ricardo Peres also authored [Entity Framework Code First Succinctly](#), [NHibernate Succinctly](#), and [Microsoft Unity Succinctly](#) for the [Succinctly series](#).

Chapter 1 Introduction

About this book

This is a book about building multitenant applications using ASP.NET. We will be discussing what multitenancy means for a web application in general, and then we'll be looking at how ASP.NET frameworks like Web Forms and MVC can help us implement multitenant solutions.

You can expect to find here working solutions for the problems outlined, some of which certainly can be enhanced and evolved. Whenever appropriate, I will leave references so that interested readers can improve upon what I've built and further their understanding of these matters.

Some knowledge of ASP.NET (Web Forms or MVC) is required, but you do not have to be an expert. Hopefully, both beginners and experts will find something they can use here.

Throughout the book, we will be referring to two imaginary tenants/domains, **ABC.com** and **XYZ.net**, which will both be hosted in our server.

What is multitenancy and why should you care?

Multitenancy is a concept that has gained some notoriety in the last years as an alternative to multiple deployments. The concept is simple: a single web application can respond to clients in a way that can make them think they are talking to different applications. The different “faces” of this application are called **tenants**, because conceptually they live in the same space—the web application—and can be addressed individually. Tenants can be added or removed just by the flip of a configuration switch.

Why is this useful? Well, for one, it renders it unnecessary to have multiple servers online, with all the maintenance and costs they require. In general, among other things, one has to consider the following:

- **Resource isolation:** A dedicated server is better shielded from failures in other infrastructure components than a shared solution.
- **Cost:** Costs are higher when we need to have one physical server per service, application, or customer than with a shared server.
- **Quality of service:** You can achieve a higher level of quality and customization if you have separate servers, because you are not concerned with how it might affect different services, applications, or customers.
- **Complexity of customization:** We must take extra care in customizing multitenant solutions because we typically don't want things to apply to all tenants in the same way.
- **Operation and management:** “Hard” management, like backups, monitoring, physical allocation, power supplies, cooling, etc., are much easier when we only have one server.

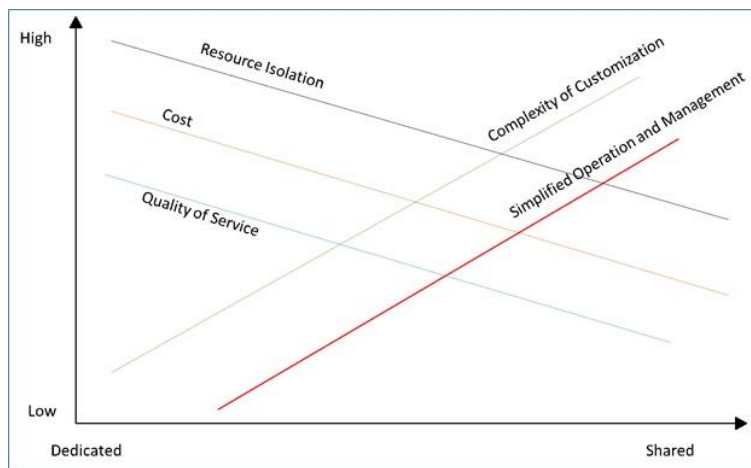


Figure 1: Decision criteria for dedicated versus shared (multitenant) deployments

In general, a multitenant framework should help us handle the following problems:

- **Branding:** Different brandings for different tenants; by branding I mean things like logo images, stylesheets, and layout in general.
- **Authentication:** Different tenants should have different user registrations; a user registered in a specific tenant should not be able to login to another tenant.
- **Workflow:** It makes sense that different tenants handle the same basic things in a (slightly) different way; a multitenant framework is supposed to make this as transparent as possible with regard to code.
- **Data Model:** Different tenants may have (slightly) different data needs; again, we should accommodate this need when possible, considering that we are talking about the same shared application.

Throughout the book we will give some attention to these problems.

Multitenancy requirements

As high-level requirements, our application framework should be able to:

- Load a number of tenants and their information automatically
- Identify, from each HTTP request, the tenant that it was meant to, or use a fallback value
- Apply a branding theme to each tenant's requested pages
- Allow users to log in only to the tenant they are associated with
- Offer common application services to all tenants that don't collide; for example, something stored in application cache for one tenant will not be overwritten by another tenant

Application services

Application services offer common functionality that .NET/ASP.NET provides out of the box, but not in a multitenant way. These will be discussed in Chapter 9, "Application Services."

Frameworks

ASP.NET is an umbrella under which different frameworks coexist: while the venerable [Web Forms](#) was the first public implementation that Microsoft put out, [MVC](#) has gained a lot of momentum since its introduction, and is now considered by some as a more modern approach to web development. [SharePoint](#) is also a strong contender, and even more so with [SharePoint Online](#), part of the [Office 365](#) product. Now it seems that [OWIN](#) (and [Katana](#)) is the new cool kid on the block, and some of the most recent frameworks that Redmond introduced recently even depend on it (take [SignalR](#), for example).

On the data side, [Entity Framework Code First](#) is now the standard for data access using the Microsoft .NET framework, but other equally solid—and some would argue, even stronger—alternatives exist, such as [NHibernate](#), and we will cover these too.

Authentication has also evolved from the venerable, but outdated, [provider mechanism](#) introduced in ASP.NET 2. A number of Microsoft (or otherwise) sponsored frameworks and standards (think [OAuth](#)) have been introduced, and the developer community certainly welcomes them. Here we will talk about the [Identity](#) framework, Microsoft's most recent authentication framework, designed to replace the old ASP.NET 2 providers and the [Simple Membership](#) API introduced in MVC 4 (not covered since it has become somewhat obsolete with the introduction of [Identity](#)).

Finally, the glue that will hold these things together, plus the code that we will be producing, is [Unity](#), Microsoft's [Inversion of Control](#) (IoC), [Dependency Injection](#) (DI), and [Aspect-Oriented Programming](#) (AOP) framework. This is mostly because I have experience with Unity, but, by all means, you are free to use whatever IoC framework you like. Unity is only used for registering components; its retrieval is done through the [Common Service Locator](#), a standard API that most IoC frameworks comply to. Most of the code will work with any other framework that you choose, provided it offers (or you roll out your own) integration with the Common Service Locator. Except the component registration code, which can be traced to a single bootstrap method, all the rest of the code relies on the Common Service Locator, not a particular IoC framework.

Chapter 2 Setup

Introduction

This chapter will walk you through setting up your development and test environment.

Visual Studio

In order to be able to follow the concepts and code described here, you will need a working installation of Visual Studio; any version from 2012 upwards will do, including the shiny new [Community Edition](#). I myself have used Visual Studio 2013, but you are free to use a different version; basically, you'll need a Visual Studio version that can handle .NET 4.x and [NuGet](#) packages.

NuGet

[NuGet](#) is Visual Studio's native packaging system. Most software companies, including Microsoft, as well as independent developers and communities (such as the [NHibernate community](#)) provide their libraries and frameworks through NuGet. It's an easy way to fetch software packages and their dependencies.

For the purpose of the examples in this book, you will need the following packages:

OWIN Self Host

```
PM> Install-Package Microsoft.Owin.Host.HttpListener
```

OWIN Web API

```
PM> Install-Package Microsoft.AspNet.WebApi.Owin
```

Entity Framework Code First

```
PM> Install-Package EntityFramework
```

NHibernate

```
PM> Install-Package NHibernate
```

Unity

```
PM> Install-Package Unity -Version 3.5.1404
```

Unity Bootstrapper for ASP.NET MVC

```
PM> Install-Package Unity.Mvc -Version 3.5.1404
```

Enterprise Library Logging Application Block

```
PM> Install-Package EnterpriseLibrary.Logging
```

ASP.NET Identity (Entity Framework)

```
PM> Install-Package  
Microsoft.AspNet.Identity.EntityFramework -Version  
2.1.0
```

ASP.NET Identity (NHibernate)

```
PM> Install-Package NHibernate.AspNet.Identity
```

OWIN Identity

```
PM> Install-Package Microsoft.AspNet.Identity.Owin
```

AspNet.Identity.EntityFramework.Multitenant

```
PM> Install-Package  
AspNet.Identity.EntityFramework.Multitenant
```

WatiN

```
PM> Install-Package WatiN
```

Address mapping

If you want to follow the examples described in this book, you will need to set up different host names for your development machine. This is so that we can make use of the [host header tenant identification pattern](#), described in an earlier chapter. If you have administrative access to the DNS server that serves your local connection (work or home), you can add different aliases to a static IP address and then use that IP address instead of using a DHCP-assigned one.

Another option is to use a free service such as [xip.io](#), which translates host names in domain **xip.io** into IP addresses; for example, **127.0.0.1.xip.io** will automatically translate to **127.0.0.1**, **192.168.1.1.xip.io** becomes **192.168.1.1**, and so on. This is so we can use the strategy for identifying tenants based on the host header mentioned earlier, but it only works with IP addresses, not host names.

Another option is to configure static IP-name mappings through the `%WINDIR%\System32\Drivers\Etc\hosts` file. This file contains lines in the format:

Code Sample 1

```
<IP address> <host name> <alias 1>    ...    <alias n>
```

For example, you can add the following entry:

Code Sample 2

```
127.0.0.1    localhost    abc.com        xyz.net
```

This will tell the IP stack on Windows that the hosts named **abc.com** and **xyz.net** will both translate to the loopback address **127.0.0.1**, without the need for a DNS lookup.

IIS

Setting up a multitenant site in **Internet Information Services (IIS)** is easy: just open the **IIS Manager** applet and select **Sites > Add website**. Then add a host name that you wish to handle:

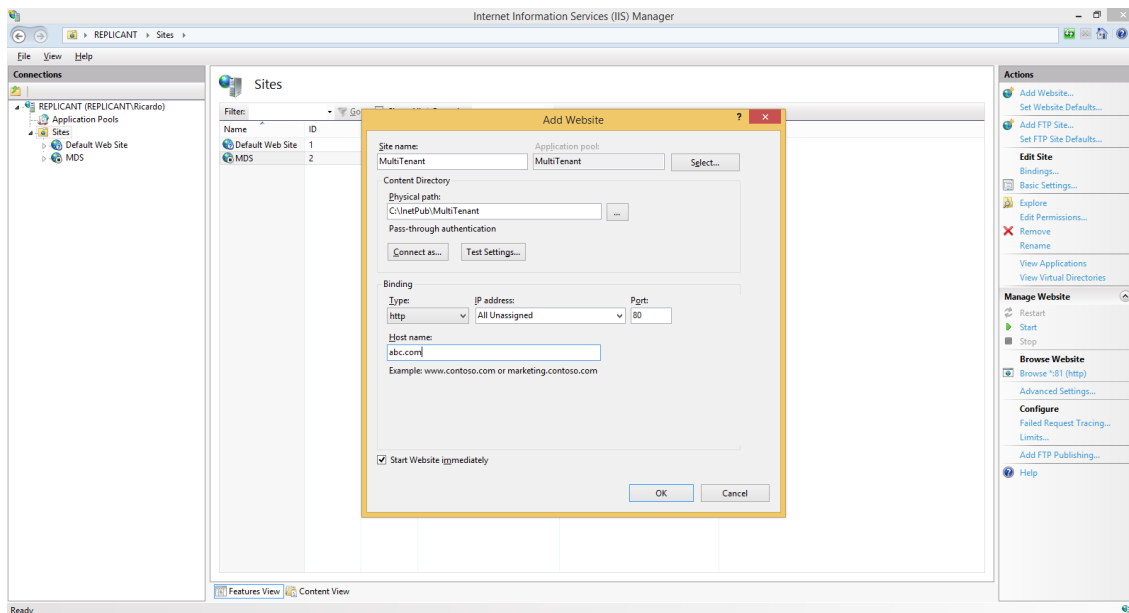


Figure 2: Setting up a multitenant site in IIS

Now add a host name for all the additional tenants, select the site, and click **Bindings**.

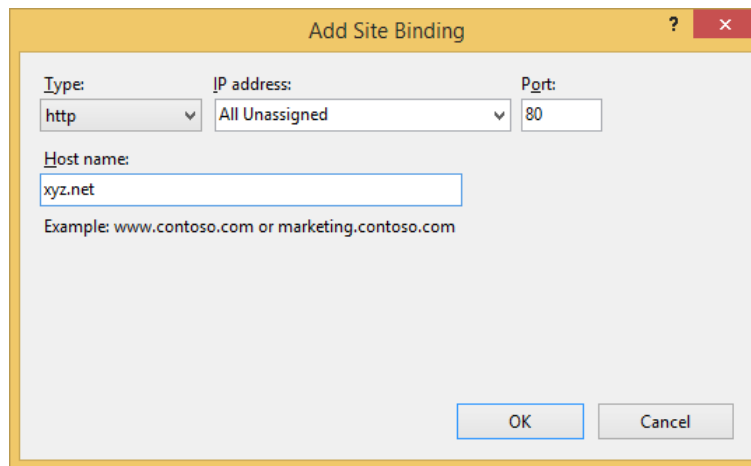


Figure 3: Adding additional host names to a site

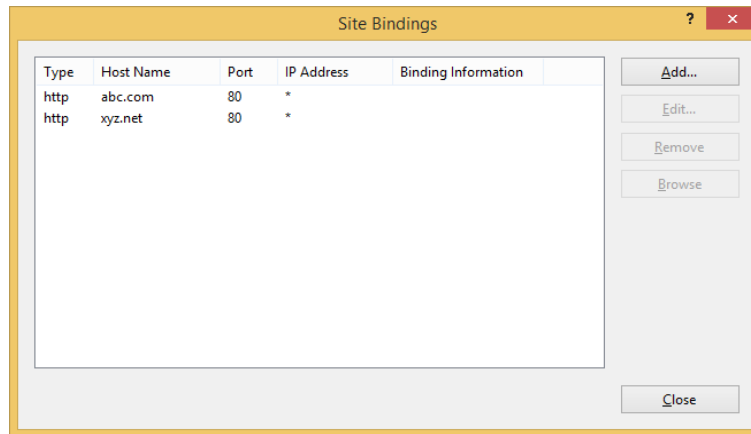


Figure 4: All the site's host names

Also important is the application pool: you can have different application pools serving each of the different tenants (the site bindings). This provides better isolation, in the sense that if something goes wrong for one of the tenants—perhaps an unhandled exception that renders the site unusable—it will not affect the others. Also, you can have the application pools running under different identities, which is nice if you wish to access different databases, or have different access levels. Just create as many application pools as you like in IIS Manager:

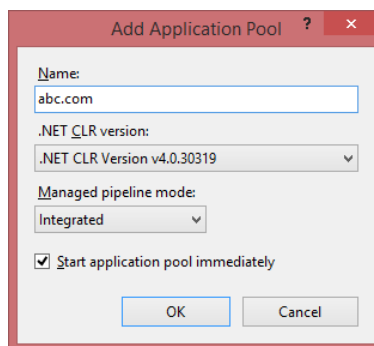


Figure 5: Creating an application pool for a tenant

IIS Express

In case you will be using **IIS Express**, besides DNS or **hosts** file, you will also need to configure the website's bindings to allow for other host names. Unfortunately, you will have to do it by hand, since IIS Express does not offer any graphical tool for that purpose; the site's configuration file is located at:

%HOMEPATH%\Documents\IISExpress\Config\ApplicationHost.config. Open it and locate the site entry that you wish to change; it should look like the following (minus **name**, **id**, **physicalPath**, and **bindingInformation**):

Code Sample 3

```
<sites>
  <site name="Multitenant" id="1">
    <application path="/" applicationPool="Clr4IntegratedAppPool">
```

```
        <virtualDirectory path="/"
            physicalPath="C:\InetPub\Multitenant" />
    </application>
    <bindings>
        <binding protocol="http" bindingInformation="*:80:localhost" />
    </bindings>
</site>
</sites>
```

You will probably see some sites already configured. In that case, you have to find the one you're interested in, maybe through the **physicalPath** attribute, and add a **binding** element with a **bindingInformation** attribute pointing to the proper host name and port.

Code Sample 4

```
<bindings>
    <binding protocol="http" bindingInformation="*:80:localhost" />
    <binding protocol="http" bindingInformation="*:80:abc.com" />
    <binding protocol="http" bindingInformation="*:80:xyz.net" />
</bindings>
```

This instructs IIS Express to also accept requests for hosts **abc.com** and **xyz.net**, together with **localhost**.

Chapter 3 Concepts

Who do you want to talk to?

As we've seen, the main premise of multitenancy is that we can respond differently to different tenants. You might have asked yourself: how does the ASP.NET know which tenant's contents it should be serving? That is, who is the client trying to reach? How can ASP.NET find out?

There may be several answers to this question:

- From the requested host name; for example, if the browser is trying to reach **abc.com** or **xyz.net**, as stated in the request URL
- From a query string parameter, like **http://host.com?tenant=abc.com**
- From the originating (client) host IP address
- From the originating client's domain name

Probably the most typical (and useful) use case is the first one; you have a single web server (or server farm) which has several DNS records ([A](#) or [CNAME](#)) assigned to it, and it will respond differently depending on how it was reached, say, **abc.com** and **xyz.net**. Being developers, let's try to define a generic contract that can give an answer to this question. Consider the following method signature:

Code Sample 5

```
String GetCurrentTenant(RequestContext context)
```

We can interpret it as: "given some request, give me the name of the corresponding tenant."



Note: If you want to know the difference between DNS A and CNAME records, you can find a good explanation [here](#).

The [RequestContext](#) class is part of the [System.Web.Routing](#) namespace, and it encapsulates all of a request's properties and context. Its [HttpContext](#) property allows easy access to the common HTTP request URL, server variables, query string parameters, cookies and headers and [RouteData](#) to routing information, if available. I chose this class for the request instead of what might be more obvious ones—[HttpContext](#) and [HttpContextBase](#)—precisely because it eases access to route data, in case we need it.



Note: [HttpContextBase](#) was introduced in .NET 3.5 to allow easy mocking, because it is not sealed, and outside of the ASP.NET pipeline. It basically mimics the properties and behavior of [HttpContext](#), which is sealed.

As for the return type, it will be the name of a tenant. More on that later.

In the .NET world, we have two main options if we are to reuse such a method signature:

- Define a method delegate
- Define an interface or an abstract base class

In our case, we'll go with an interface, enter **ITenantIdentifierStrategy**:

Code Sample 6

```
public interface ITenantIdentifierStrategy
{
    String GetCurrentTenant(RequestContext context);
}
```

Host header strategy

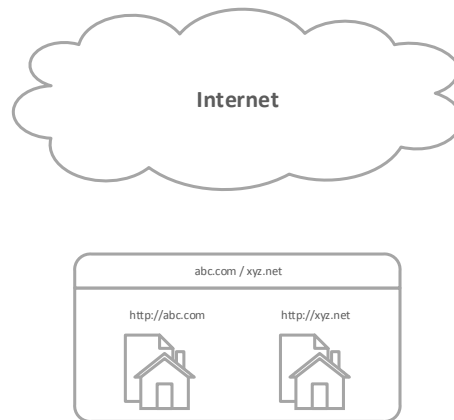


Figure 6: Host header strategy

A simple implementation of **ITenantIdentifierStrategy** for using the requested host name ([Host HTTP header](#)) as the tenant identifier is probably the one you'll use more often in public-facing sites. A single server with a single IP address and multiple host and domain names will differentiate tenants by the requested host, in the HTTP request:

Table 1: Mapping of HTTP Host headers to tenants

HTTP Request Headers	Tenant
GET /Default.aspx HTTP/1.1 Host: abc.com	abc.com
GET /Default.aspx HTTP/1.1 Host: xyz.net	xyz.net



Note: For more information on the Host HTTP header, [see RFC 2616, HTTP Header Field Definitions](#).

A class for using the host header as the tenant name might look like the following:

Code Sample 7

```
public class HostHeaderTenantIdentifierStrategy : ITenantIdentifierStrategy
{
    public String GetCurrentTenant(RequestContext context)
    {
        return context.HttpContext.Request.Url.Host.ToLower();
    }
}
```



Tip: This code is meant for demo purposes only; it does not have any kind of validation and just returns the requested host name in lowercase. Real-life code should be slightly more complex.

Query string strategy

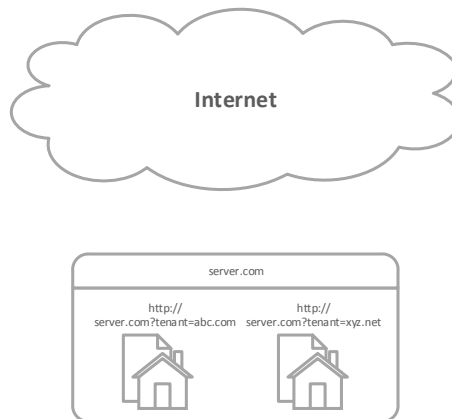


Figure 7: Query string strategy

We might want to use a query string parameter to differentiate between tenants, like **host.com?tenant=abc.com**:

Table 2: Mapping of HTTP query strings to tenants

HTTP Request URL	Tenant
http://host.com?Tenant=abc.com	abc.com
http://host.com?Tenant=xyz.net	xyz.net

This strategy makes it really easy to test with different tenants; no need to configure anything—just pass a query string parameter in the URL.

We can use a class like the following, which picks the **Tenant** query string parameter and assigns it as the tenant name:

```
public class QueryStringTenantIdentifierStrategy : ITenantIdentifierStrategy
{
    public String GetCurrentTenant(RequestContext context)
    {
        return (context.HttpContext.Request.QueryString["Tenant"] ??
```

```
        String.Empty).ToLower();  
    }  
}
```



Tip: Even if this technique may seem interesting at first, it really isn't appropriate for real-life scenarios. Just consider that for all your internal links and postbacks you will have to make sure to add the Tenant query string parameter; if for whatever reason it is lost, you are dropped out of the desired tenant.



Note: A variation of this pattern could use the HTTP variable `PATH_INFO` instead of `QUERY_STRING`, but this would have impacts, namely, with MVC.

Source IP address strategy

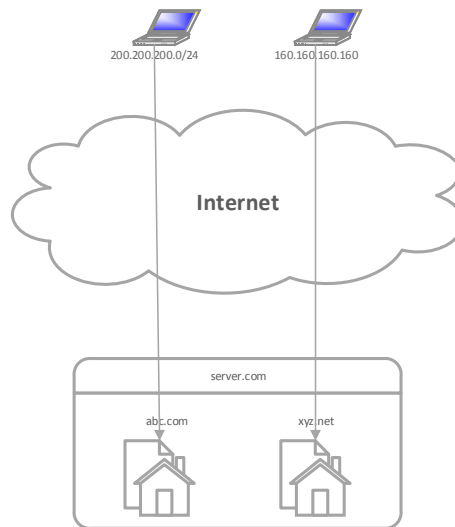


Figure 8: Source IP strategy

Now suppose we want to determine the tenant's name from the IP address of the originating request. Say a user located at a network whose address is **200.200.200.0/24** will be assigned tenant **abc.com** and another one using a static IP of **160.160.160.160** will get **xyz.net**. It gets slightly trickier, because we need to manually register these assignments, and we need to do some math to find out if a request matches a list of registered network addresses.

We have two possibilities for associating a network address to a tenant name:

- We use a single IP address
- We use an IP network and a subnet mask.

Say, for example:

Table 3: Mapping of source IP addresses to tenants

Source Network / IP	Tenant
200.200.200.0/24 (200.200.200.1-200.200.200.254)	abc.com
160.160.160.1	xyz.net

The .NET Base Class Library does not offer an out-of-the-box API for IP network address operations, so we have to build our own. Consider the following helper methods:

Code Sample 8

```
public static class SubnetMask
{
    public static IPAddress CreateByHostBitLength(Int32 hostPartLength)
    {
        var binaryMask = new Byte[4];
        var netPartLength = 32 - hostPartLength;
        if (netPartLength < 2)
        {
            throw new ArgumentException
                ("Number of hosts is too large for IPv4.");
        }
        for (var i = 0; i < 4; i++)
        {
            if (i * 8 + 8 <= netPartLength)
            {
                binaryMask[i] = (Byte) 255;
            }
            else if (i * 8 > netPartLength)
            {
                binaryMask[i] = (Byte) 0;
            }
            else
            {
                var oneLength = netPartLength - i * 8;
                var binaryDigit = String.Empty
                    .PadLeft(oneLength, '1').PadRight(8, '0');
                binaryMask[i] = Convert.ToByte(binaryDigit, 2);
            }
        }
    }
}
```

```

        return new IPAddress(binaryMask);
    }

    public static IPAddress CreateByNetBitLength(Int32 netPartLength)
    {
        var hostPartLength = 32 - netPartLength;
        return CreateByHostBitLength(hostPartLength);
    }

    public static IPAddress CreateByHostNumber(Int32 numberOfHosts)
    {
        var maxNumber = numberOfHosts + 1;
        var b = Convert.ToString(maxNumber, 2);
        return CreateByHostBitLength(b.Length);
    }
}

public static class IPAddressExtensions
{
    public static IPAddress[] ParseIPAddressAndSubnetMask(String ipAddress)
    {
        var ipParts = ipAddress.Split('/');
        var parts = new IPAddress[] { ParseIPAddress(ipParts[0]),
            ParseSubnetMask(ipParts[1]) };
        return parts;
    }

    public static IPAddress ParseIPAddress(String ipAddress)
    {
        return IPAddress.Parse(ipAddress.Split('/').First());
    }

    public static IPAddress ParseSubnetMask(String ipAddress)
    {
        var subnetMask = ipAddress.Split('/').Last();
        var subnetMaskNumber = 0;
        if (!Int32.TryParse(subnetMask, out subnetMaskNumber))
        {
            return IPAddress.Parse(subnetMask);
        }
        else
        {
            return SubnetMask.CreateByNetBitLength(subnetMaskNumber);
        }
    }

    public static IPAddress GetBroadcastAddress(this IPAddress address,
        IPAddress subnetMask)
    {
        var ipAdressBytes = address.GetAddressBytes();

```

```

        var subnetMaskBytes = subnetMask.GetAddressBytes();
        if (ipAdressBytes.Length != subnetMaskBytes.Length)
        {
            throw new ArgumentException
                ("Lengths of IP address and subnet mask do not match.");
        }
        var broadcastAddress = new Byte[ipAdressBytes.Length];
        for (var i = 0; i < broadcastAddress.Length; i++)
        {
            broadcastAddress[i] = (Byte)(ipAdressBytes[i] |
                (subnetMaskBytes[i] ^ 255));
        }
        return new IPAddress(broadcastAddress);
    }

    public static IPAddress GetNetworkAddress(this IPAddress address,
        IPAddress subnetMask)
    {
        var ipAdressBytes = address.GetAddressBytes();
        var subnetMaskBytes = subnetMask.GetAddressBytes();
        if (ipAdressBytes.Length != subnetMaskBytes.Length)
        {
            throw new ArgumentException
                ("Lengths of IP address and subnet mask do not match.");
        }
        var broadcastAddress = new Byte[ipAdressBytes.Length];
        for (var i = 0; i < broadcastAddress.Length; i++)
        {
            broadcastAddress[i] = (Byte)(ipAdressBytes[i]
                & (subnetMaskBytes[i]));
        }
        return new IPAddress(broadcastAddress);
    }

    public static Boolean IsInSameSubnet(this IPAddress address2,
        IPAddress address, Int32 hostPartLength)
    {
        return IsInSameSubnet(address2, address, SubnetMask
            .CreateByHostBitLength(hostPartLength));
    }

    public static Boolean IsInSameSubnet(this IPAddress address2,
        IPAddress address, IPAddress subnetMask)
    {
        var network1 = address.GetNetworkAddress(subnetMask);
        var network2 = address2.GetNetworkAddress(subnetMask);
        return network1.Equals(network2);
    }
}

```



Note: This code is based in code publicly available [here](#) (and slightly modified).

Now we can write an implementation of `ITenantIdentifierStrategy` that allows us to map IP addresses to tenant names:

Code Sample 9

```
public class SourceIPTenantIdentifierStrategy : ITenantIdentifierStrategy
{
    private readonly Dictionary<Tuple<IPAddress, IPAddress>, String> networks = new Dictionary<Tuple<IPAddress, IPAddress>, String>();

    public IPTenantIdentifierStrategy Add(IPAddress ipAddress,
        Int32 netmaskBits, String name)
    {
        return this.Add(ipAddress, SubnetMask.CreateByNetBitLength(
            netmaskBits), name);
    }

    public IPTenantIdentifierStrategy Add(IPAddress ipAddress,
        IPAddress netmaskAddress, String name)
    {
        this.networks
        [new Tuple<IPAddress, IPAddress>(ipAddress, netmaskAddress)] = name.ToLower();
        return this;
    }

    public IPTenantIdentifierStrategy Add(IPAddress ipAddress, String name)
    {
        return this.Add(ipAddress, null, name);
    }

    public String GetCurrentTenant(RequestContext context)
    {
        var ip = IPAddress.Parse(context.HttpContext.Request
            .UserHostAddress);
        foreach (var entry in this.networks)
        {
            if (entry.Key.Item2 == null)
            {
                if (ip.Equals(entry.Key.Item1))
                {
                    return entry.Value.ToLower();
                }
            }
        }
    }
}
```

```

        else
        {
            if (ip.IsInSameSubnet(entry.Key.Item1,
                                entry.Key.Item2))
            {
                return entry.Value;
            }
        }
    }
    return null;
}
}

```

Notice that this class is not thread safe; if you wish to make it so, one possibility would be to use a [ConcurrentDictionary<TKey, TValue>](#) instead of a plain [Dictionary<TKey, TValue>](#).

Before we can use [IPTenantIdentifierStrategy](#), we need to register some mappings:

Code Sample 10

```

var s = new SourceIPTenantIdentifierStrategy();
s.Add(IPAddress.Parse("200.200.200.0", 24), "abc.com");
s.Add(IPAddress.Parse("160.160.160.1"), "xyz.net");

```

In this example we see that tenant **xyz.net** is mapped to a single IP address, **160.160.160.1**, while tenant **abc.com** is mapped to a network of **200.200.200.0** with a **24**-bit network mask, meaning all hosts ranging from **200.200.200.1** to **200.200.200.254** will be included.

Source domain strategy

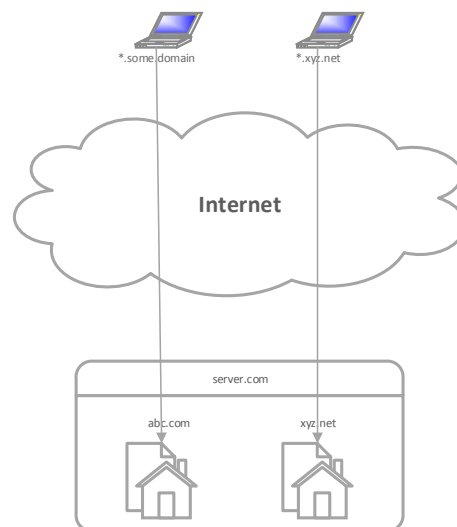


Figure 9: Source domain strategy

We may not know about IP addresses, but instead, domain names; hence a strategy based on client domain names comes in order. We want to get the tenant's name from the domain name part of the requesting host, something like:

Table 4: Mapping source domain names to tenants

Source Domain	Tenant
*.some.domain	abc.com
*.xyz.net	xyz.net

Subdomains should also be included. Here's a possible implementation of such a strategy:

Code Sample 11

```
public class SourceDomainTenantIdentifierStrategy : ITenantIdentifierStrategy
{
    private readonly Dictionary<String, String> domains = new
Dictionary<String, String>(StringComparer.OrdinalIgnoreCase);

    public DomainTenantIdentifierStrategy Add(String domain, String name)
    {
        this.domains[domain] = name;
        return this;
    }

    public DomainTenantIdentifierStrategy Add(String domain)
    {
        return this.Add(domain, domain);
    }

    public String GetCurrentTenant(RequestContext context)
    {
        var hostName = context.HttpContext.Request.UserHostName;
        var domainName = String.Join(".", hostName.Split('.'))
            .Skip(1).ToLower();
        return this.domains.Where(domain => domain.Key == domainName)
            .Select(domain => domain.Value).FirstOrDefault();
    }
}
```

For **DomainTenantIdentifierStrategy**, of course, we also need to enter some mappings:

Code Sample 12

```
var s = new SourceDomainTenantIdentifierStrategy();
s.Add("some.domain", "abc.com");
```

```
s.Add("xyz.net");
```

The first entry maps all client requests coming from the **some.domain** domain (or a subdomain of) to a tenant named **abc.com**. The second does an identical operation for the **xyz.net** domain, where we skip the tenant's name because it should be identical to the domain name.

As you can see, two of the previous strategies' implementations—host header and query string parameter—are basically stateless and immutable, so instead of creating new instances every time, we can instead have static instances of each in a well-known location. Let's create a structure for that purpose:

Code Sample 13

```
public static class TenantsConfiguration
{
    public static class Identifiers
    {
        public static readonly HostHeaderTenantIdentifierStrategy
            HostHeader = new HostHeaderTenantIdentifierStrategy();
        public static readonly QueryStringTenantIdentifierStrategy
            QueryString = new QueryStringTenantIdentifierStrategy();
    }
}
```



Tip: Notice the *DefaultTenant* property. This is what will be used if the tenant identification strategy is unable to map a request to a tenant.

Two other strategies—by source IP address and by domain name—require configuration, so we shouldn't have them as constant instances, but, to allow for easy finding, let's add some static factories to the **TenantsConfiguration** class introduced just now:

Code Sample 14

```
public static class TenantsConfiguration
{
    //rest goes here
    public static class Identifiers
    {
        //rest goes here

        public static SourceDomainTenantIdentifierStrategy SourceDomain()
        {
            return new SourceDomainTenantIdentifierStrategy();
        }

        public static SourceIPTenantIdentifierStrategy SourceIP()
        {
            return new SourceIPTenantIdentifierStrategy();
        }
    }
}
```



Note: In Chapter 12, we will see how all these strategies are related.

Getting the current tenant

We've looked at some strategies for obtaining the tenant's name from the request; now, we have to pick one, and store it somewhere where it can be easily found.

Static property

One option is to store this as a static property in the **TenantsConfiguration** class:

Code Sample 15

```
public static class TenantsConfiguration
{
    public static ITenantIdentifierStrategy TenantIdentifier { get; set; }

    //rest goes here
}
```

Now, we can choose whatever strategy we want, probably picking one from the **TenantsConfiguration**'s static members. Also, we need to set the **DefaultTenant** property, so that if the current strategy is unable to identify the tenant to use, we have a fallback:

Code Sample 16

```
TenantsConfiguration.TenantIdentifier =
    TenantsConfiguration.Identifiers.HostHeader;

TenantsConfiguration.DefaultTenant = "abc.com";
```


Unity and the Common Service Locator

Another option is to use an [Inversion of Control](#) (IoC) framework to store a singleton reference to the tenant identifier instance of our choice. Better yet, we can use the [Common Service Locator](#) to abstract away the IoC that we're using. This way, we are not tied to a particular implementation, and can even change the one to use without impacting the code (except, of course, some bootstrap code). There are several IoC containers for the .NET framework. Next, we'll see an example using a well-known IoC framework, [Microsoft Unity](#), part of Microsoft's [Enterprise Library](#). This approach has the advantage that we can register new strategies dynamically through code (or through the Web.config file) without changing any code, should we need to do so. We will be using this approach throughout the book.

Code Sample 17

```
//set up Unity
var unity = new UnityContainer();
//register instances
unity.RegisterInstance<ITenantIdentifierStrategy>(TenantsConfiguration.Identifiers.HostHeader);
unity.RegisterInstance<String>("DefaultTenant", "abc.com");
//set up Common Service Locator with the Unity instance
ServiceLocator.SetLocatorProvider(() => new UnityServiceLocator(unity));
//resolve the tenant identifier strategy and the default tenant
var
identifier = ServiceLocator.Current.GetInstance<ITenantIdentifierStrategy>();
var defaultTenant = ServiceLocator.Current.GetInstance<String>("DefaultTenant");
```

Chapter 6 also talks about Unity and how to use it to return components specific to the current tenant.



Note: *Unity is just one among tens of IoC containers, offering similar functionality as well as more specific ones. Not all have adapters for the Common Service Locator, but it is generally easy to implement one. For a more in-depth discussion of IoC, the Common Service Locator, and Unity, please see [Microsoft Unity Succinctly](#).*

What's in a name?

Now that we have an abstraction that can give us a tenant's **name**, let's think for a moment what else a tenant needs.

We might need a **theme** as well, something that aggregates things like color schemes and fonts. Different tenants might want different looks.

In the ASP.NET world, the two major frameworks, MVC and Web Forms, offer the concept of master pages or layout pages (in MVC terminology), which are used to define the global layout of an HTML page. With this, it is easy to enforce a consistent layout throughout a site, so, let's consider a **master page** (or layout page, in MVC terms) property.

It won't hurt having a collection of key/value pairs that are specific to a tenant, regardless of having a more advanced configuration functionality, so we now have a general-purpose **property bag**.

Windows offers a general purpose, operating-system-supported mechanism for monitoring applications: [performance counters](#). Performance counters allow us to monitor in real time certain aspects of our application, and even react automatically to conditions. We shall expose a collection of **counter instances** to be created automatically in association with a tenant.

It might be useful to offer a general-purpose extension mechanism; before the days IoC became popular, .NET already included a generic interface for resolving a component from a type in the form of the [IServiceProvider](#) interface. Let's also consider a **service resolution** mechanism using this interface.

Finally, it makes sense to have a tenant **initialize** itself when it is registering with our framework. This is not data, but behavior.

So, based on what we've talked about, our tenant definition interface, **ITenantConfiguration**, will look like this:

Code Sample 18

```
public interface ITenantConfiguration
{
    String Name { get; }
    String Theme { get; }
    String MasterPage { get; }
    IServiceProvider ServiceProvider { get; }
    IDictionary<String, Object> Properties { get; }
    IEnumerable<String> Counters { get; }
    void Initialize();
}
```

For example, a tenant called **xyz.net** might have the following configuration:

Code Sample 19

```
public sealed class XyzNetTenantConfiguration : ITenantConfiguration
{
    public XyzNetTenantConfiguration()
    {
        //do something productive
        this.Properties = new Dictionary<String, Object>();
        this.Counters = new List<String> { "C", "D", "E" };
    }

    public void Initialize()
```

```

{
    //do something productive
}

public String Name { get { return "xyz.net"; } }
public String MasterPage { get { return this.Name; } }
public String Theme { get { return this.Name; } }
public IServiceProvider ServiceProvider { get; private set; }
public IDictionary<String, Object> Properties { get; private set; }
public IEnumerable<String> Counters { get; private set; }
}

```

In this example, we are returning a **MasterPage** and a **Theme** that are identical to the tenant's **Name**, and are not returning anything really useful in the **Counters**, **Properties** and **ServiceProvider** properties, but in real life, you would probably do something else. Any counter names you return will be automatically created as numeric performance counter instances.



***Note:** There are lots of other options for providing this kind of information, like with attributes, for example.*

Finding tenants

Tenant location strategies

What's a house without someone to inhabit it?

Now, we have to figure out a strategy for finding tenants. Two basic approaches come to mind:

- Manually configuring individual tenants explicitly
- Automatically finding and registering tenants

Again, let's abstract this functionality in a nice interface, **ITenantLocationStrategy**:

Code Sample 20

```

public interface ITenantLocationStrategy
{
    IDictionary<String, Type> GetTenants();
}

```

This interface returns a collection of **names** and **types**, where the **types** are instances of some non-abstract class that implements **ITenantConfiguration** and the **names** are unique tenant identifiers.

We'll also keep a static property in **TenantsConfiguration**, **DefaultTenant**, where we can store the name of the default tenant, as a fallback if one cannot be identified automatically:

Code Sample 21

```
public static class TenantsConfiguration
{
    public static String DefaultTenant { get; set; }
    //rest goes here
}
```

Here are the strategies for locating and identifying tenants:

Code Sample 22

```
public static class TenantsConfiguration
{
    public static String DefaultTenant { get; set; }
    public static ITenantIdentifierStrategy TenantIdentifier
    { get; set; }
    public static ITenantLocationStrategy TenantLocation { get; set; }
    //rest goes here
}
```

Next, we'll see some ways to register tenants.

Manually Registering Tenants

Perhaps the most obvious way to register tenants is to have a configuration section in the **Web.config** file where we can list all the types that implement tenant configurations. We would like to have a simple structure, something like this:

Code Sample 23

```
<tenants default="abc.com">

<add name="abc.com" type="MyNamespace.AbcComTenantConfiguration, MyAssembly"
/>

<add name="xyz.net" type="MyNamespace.XyzNetTenantConfiguration, MyAssembly"
/>

</tenants>
```

Inside the **tenants** section, we have a number of elements containing the following attributes:

- **Name:** the unique tenant identifier; in this case, it will be the same as the domain name that we wish to answer to (see Host header strategy)
- **Type:** the fully qualified type name of a class that implements **ITenantConfiguration**
- **Default:** the default tenant, if no tenant can be identified from the current tenant identifier strategy

The corresponding configuration classes in .NET are:

Code Sample 24

```
[Serializable]
public class TenantsSection : ConfigurationSection
{
    public static readonly TenantsSection Section = ConfigurationManager
        .GetSection("tenants") as TenantsSection;

    [ConfigurationProperty("", IsDefaultCollection = true, IsRequired = true)]
    public TenantsElementCollection Tenants
    {
        get
        {
            return base[String.Empty] as TenantsElementCollection;
        }
    }
}

[Serializable]
public class TenantsElementCollection : ConfigurationElementCollection,
    IEnumerable<TenantElement>
{
    protected override String ElementName { get { return String.Empty; } }

    protected override ConfigurationElement CreateNewElement()
    {
        return new TenantElement();
    }

    protected override Object GetElementKey(ConfigurationElement element)
    {
        var elm = element as TenantElement;
        return String.Concat(elm.Name, ":", elm.Type);
    }

    IEnumerator<TenantElement> IEnumerable<TenantElement>.GetEnumerator()
    {
        foreach (var elm in this.OfType<TenantElement>())
        {
            yield return elm;
        }
    }
}
```

```

}

[Serializable]
public class TenantElement : ConfigurationElement
{
    [ConfigurationProperty("name", IsKey = true, IsRequired = true)]
    [StringValidator(MinLength = 2)]
    public String Name
    {
        get { return this["name"] as String; }
        set { this["name"] = value; }
    }

    [ConfigurationProperty("type", IsKey = true, IsRequired = true)]
    [TypeConverter(typeof(TypeTypeConverter))]
    public Type Type
    {
        get { return this["type"] as Type; }
        set { this["type"] = value; }
    }

    [ConfigurationProperty("default", IsKey = false, IsRequired = false,
        DefaultValue = false)]
    public Boolean Default
    {
        get { return (Boolean)(this["default"] ?? false); }
        set { this["default"] = value; }
    }
}

```

So, our tenant location strategy (**ITenantLocationStrategy**) implementation might resemble the following:

Code Sample 25

```

public sealed class XmlTenantLocationStrategy : ITenantLocationStrategy
{
    public static readonly ITenantLocationStrategy Instance = new
        XmlTenantLocationStrategy();

    public IDictionary<String, Type> GetTenants()
    {
        var tenants = TenantsSection.Section.Tenants
            .ToDictionary(x => x.Name, x => x.Type);
        foreach (var tenant in TenantsSection.Section.Tenants
            .OfType<TenantElement>())
        {
            if (tenant.Default)
            {
                if (String.IsNullOrEmpty

```

```

        (TenantsConfiguration.DefaultTenant))
    {
        TenantsConfiguration.DefaultTenant =
            tenant.Name;
    }
}
return tenants;
}
}

```

You might have noticed the **Instance** field; because there isn't much point in having several instances of this class, since all point to the same .config file, we can have a single static instance of it, and always use it when necessary. Now, all we have to do is set this strategy as the one to use in **TenantsConfiguration**. If we are to use the Common Service Locator strategy (see Unity and the Common Service Locator), we need to add the following line in our Unity registration method and **TenantsConfiguration** static class:

Code Sample 26

```

public static class TenantsConfiguration
{
    //rest goes here
    public static class Locations
    {
        public static XmlTenantLocationStrategy Xml()
        {
            return XmlTenantLocationStrategy.Instance;
        }
        //rest goes here
    }
}

container.RegisterInstance<ITenantLocationStrategy>(TenantsConfiguration.Locations.Xml());

```

By keeping a factory of all strategies in the **TenantsConfiguration** class, it is easier for developers to find the strategy they need, from the set of the ones provided out of the box.

Finding tenants automatically

As for finding tenants automatically, there are several alternatives, but I opted for using [Microsoft Extensibility Framework](#) (MEF). This is a framework included with .NET that offers mechanisms for automatically locating plug-ins from, among others, the file system. Its concepts include:

- **Contract Type:** The abstract base class or interface that describes the functionality to import (the plug-in API)
- **Contract Name:** A free-form name that is used to differentiate between multiple parts of the same contract type
- **Part:** A concrete class that exports a specific Contract Type and Name and can be found by a **Catalog**
- **Catalog:** An MEF class that implements a strategy for finding parts

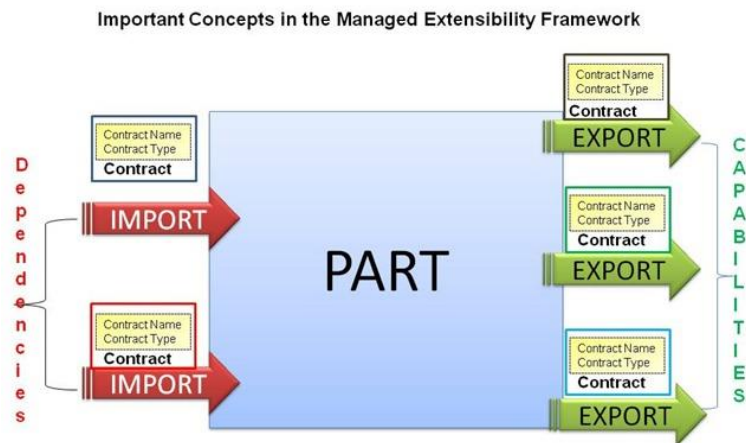


Figure 10: MEF architecture

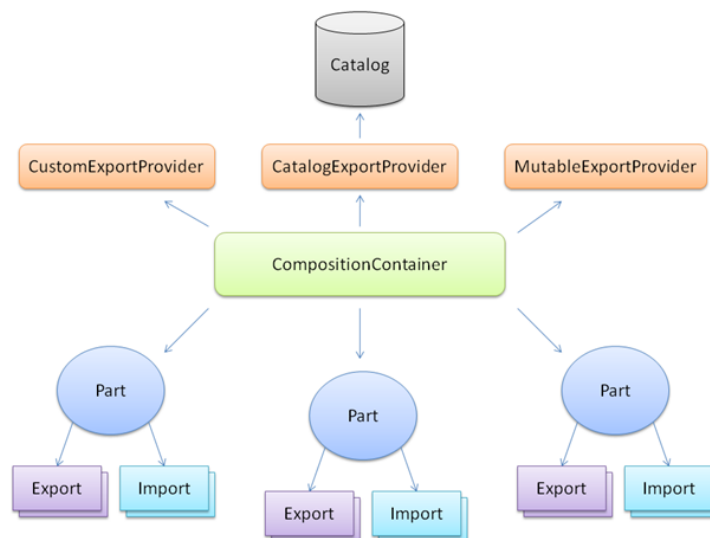


Figure 11: MEF parts

We won't go into MEF in depth; we'll just see how we can use it to find tenant configuration classes automatically in our code. We just need to decorate plug-in classes—in our case, tenant configurations—with a couple of attributes, choose a strategy to use, and MEF will find and optionally instantiate them for us. Let's see an example of using MEF attributes with a tenant configuration class:

Code Sample 27

```
[ExportMetadata("Default", true)]
[PartCreationPolicy(CreationPolicy.Shared)]
[Export("xyz.net", typeof(ITenantConfiguration))]
public sealed class XyzNetTenantConfiguration : ITenantConfiguration
{
    public XyzNetTenantConfiguration()
    {
        //do something productive
        this.Properties = new Dictionary<String, Object>();
        this.Counters = new List<String> { "C", "D", "E" };
    }

    public void Initialize()
    {
        //do something productive
    }

    public String Name { get { return "xyz.net"; } }
    public String MasterPage { get { return this.Name; } }
    public String Theme { get { return this.Name; } }
    public IServiceProvider ServiceProvider { get; private set; }
    public IDictionary<String, Object> Properties { get; private set; }
    public IEnumerable<String> Counters { get; private set; }
}
```

This class is basically identical to one presented in code sample 18, with the addition of the [ExportMetadataAttribute](#), [PartCreationPolicyAttribute](#), and [ExportAttribute](#) attributes. These are part of the MEF framework and their purpose is:

- [ExportMetadataAttribute](#): Allows adding custom attributes to a registration; you can add as many of these as you want. These attributes don't have any special meaning to MEF, and will only be meaningful to classes importing plug-ins.
- [PartCreationPolicyAttribute](#): How the plug-in is going to be created by MEF; currently, two options exist: [Shared](#) (for singletons) or [NonShared](#) (for transient instances, the default)
- [ExportAttribute](#): Marks a type for exporting as a part. This is the only required attribute.

Now, our implementation of a tenant location strategy using MEF goes like this:

Code Sample 28

```
public sealed class MefTenantLocationStrategy : ITenantLocationStrategy
{
    private readonly ComposablePartCatalog catalog;

    public MefTenantLocationStrategy(params String [] paths)
    {
        this.catalog = new AggregateCatalog(paths.Select(
            path => new DirectoryCatalog(path)));
    }
}
```

```

    }

    public MefTenantLocationStrategy(params Assembly [] assemblies)
    {
        this.catalog = new AggregateCatalog(assemblies
            .Select(asm => new AssemblyCatalog(asm)));
    }

    public IDictionary<String, Type> GetTenants()
    {
        //get the default tenant
        var tenants = this.catalog.GetExports(
            new ImportDefinition(a => true, null,
                ImportCardinality.ZeroOrMore, false, false))
            .ToList();

        var defaultTenant = tenants.SingleOrDefault(x => x.Item2.Metadata
            .ContainsKey("Default"));

        if (defaultTenant != null)
        {
            var isDefault = Convert.ToBoolean(defaultTenant.Item2
                .Metadata["Default"]);
            if (isDefault)
            {
                if (String.IsNullOrEmpty(
                    TenantsConfiguration.DefaultTenant))
                {
                    TenantsConfiguration.DefaultTenant =
                        defaultTenant.Item2.ContractName;
                }
            }
        }

        return this.catalog.GetExportedTypes<ITenantConfiguration>();
    }
}

```

This code will look up parts from either a set of assemblies or a set of paths. Then it will try to set a tenant as the default one, if no default is set. Let's add it to the **TenantsConfiguration** class:

Code Sample 29

```

public static class TenantsConfiguration
{
    //rest goes here
    public static class Locations
    {
        public static XmlTenantLocationStrategy Xml()
    }
}

```

```

        {
            return XmlTenantLocationStrategy.Instance;
        }
        public static MefTenantLocationStrategy Mef
            (params Assembly[] assemblies)
        {
            return new MefTenantLocationStrategy(assemblies);
        }

        public static MefTenantLocationStrategy Mef(params String [] paths)
        {
            return new MefTenantLocationStrategy(paths);
        }
        //rest goes here
    }
}

container.RegisterInstance<ITenantLocationStrategy>(TenantsConfiguration.Locations.
    Mef("some", "path"));

```

At the end, we register this implementation as the default tenant location strategy—remember, there can be only one. This will be the instance returned by the Common Service Locator for the `ITenantLocationStrategy` type.

Bootstrapping tenants

We need to run the bootstrapping code at the start of the web application. We have a couple of options:

- At the [Application Start](#) event of the [HttpApplication](#), normally defined in the **Global.asax.cs**
- Using the [PreApplicationStartMethodAttribute](#), to run code before [Application Start](#)

In any case, we need to set the strategies and get the tenants list:

Code Sample 30

```

TenantsConfiguration.DefaultTenant = "abc.com";
TenantsConfiguration.TenantIdentifier = TenantsConfiguration.Identifiers
    .HostHeader;
TenantsConfiguration.TenantLocation = TenantsConfiguration.Locations.Mef();
TenantsConfiguration.Initialize();

```

Here's an updated `TenantsConfiguration` class:

Code Sample 31

```

public static class TenantsConfiguration
{
    public static String DefaultTenant { get; set; }
}

```

```

public static ITenantIdentifierStrategy TenantIdentifierStrategy
    { get; set; }
public static ITenantLocationStrategy TenantLocationStrategy { get; set;
; }

public static void Initialize()
{
    var tenants = GetTenants();
    InitializeTenants(tenants);
    CreateLogFactories(tenants);
    CreatePerformanceCounters(tenants);
}

private static void InitializeTenants
    (IEnumerable<ITenantConfiguration> tenants)
{
    foreach (var tenant in tenants)
    {
        tenant.Initialize();
    }
}

private static void CreatePerformanceCounters(
    IEnumerable<ITenantConfiguration> tenants)
{
    if (PerformanceCounterCategory.Exists("Tenants") == false)
    {
        var col = new CounterCreationDataCollection(tenants
            .Select(t => new CounterCreationData(t.Name,
                String.Empty,
                PerformanceCounterType.NumberOfItems32))
            .ToArray());
        var category = PerformanceCounterCategory
            .Create("Tenants",
                "Tenants Performance Counters",
                PerformanceCounterCategoryType
                    .MultiInstance,
                col);

        foreach (var tenant in tenants)
        {
            foreach (var instanceName in tenant.Counters)
            {
                using (var pc = new PerformanceCounter(
                    category.CategoryName,
                    tenant.Name,
                    String.Concat(tenant.Name,
                        ":",
                        instanceName), false))
                {

```

```

    }
    }
    }
    }
    pc.RawValue = 0;
}

```

By leveraging the **ServiceProvider** property, we can add our own tenant-specific services. For instance, consider this implementation that registers a private Unity container and adds a couple of services to it:

Code Sample 32

```

public sealed class XyzNetTenantConfiguration : ITenantConfiguration
{
    private readonly IUnityContainer container = new UnityContainer();

    public void Initialize()
    {
        this.container.RegisterType<IMyService, MyServiceImpl>();
        this.container.RegisterInstance<IMyOtherService>(new
MyOtherServiceImpl());
    }

    public IServiceProvider ServiceProvider { get { return
this.container; } }
    //rest goes here
}

```

Then, we can get the actual services from the current tenant, if they are available:

Code Sample 33

```

var tenant = TenantsConfiguration.GetCurrentTenant();
var myService = tenant.ServiceProvider.GetService(typeof(IMyService)) as
IMyService;

```

There's a small gotcha here: Unity will throw an exception in case there is no registration by the given type. In order to get around this, we can use this nice extension method over [IServiceProvider](#):

Code Sample 34

```

public static class ServiceProviderExtensions
{
    public static T GetService<T>(this IServiceProvider serviceProvider)
    {
        var service = default(T);

        try
        {

```

```
        service = (T) serviceProvider.GetService(typeof (T));
    }
    catch
    {
    }

    return service;
}
}
```

As you can see, besides performing a cast, it also returns the default type of the generic parameter (likely **null**, in the case of interfaces) if no registration exists.

Chapter 4 ASP.NET Web Forms

Introduction

[ASP.NET Web Forms](#) is the original framework introduced with .NET for web development. Probably the reason for its success lies in how easy it is to implement simple scenarios including data binding, AJAX functionality, keeping control state between postbacks, and performing user authentication and authorization. Its detractors point out that all the magic comes with a cost in terms of performance, excessive complexity, error-proneness of the page, and control lifecycle; and have been pushing newer technologies such as MVC. The truth, in my opinion, is that some complex requirements, such as those addressed by SharePoint (which is based on Web Forms) are too difficult, if not impossible, to implement with MVC.

In this chapter we will see which mechanisms ASP.NET Web Forms has to offer when it comes to writing multitenant applications—namely, when it comes to branding.



Note: *Even though it may seem as though MVC is taking over, make no mistake—Web Forms will still be around for a long time. It is true that ASP.NET 5 will not support Web Forms, at least in its initial version, but the 4.x series will coexist with version 5 and will continue development of Web Forms. For pointers and discussions on MVC vs Web Forms, check out this [Microsoft Curah](#) and [Dino Esposito's view](#) on the subject.*

Table 5: Branding concepts

Concept	API
Branding	Master Pages
	Themes and Skins
Authentication	ASP.NET Membership / ASP.NET Identity
Workflow	Unity / Common Service Locator
Data Model	Entity Framework Code First or NHibernate

Branding

ASP.NET Web Forms offers primarily two techniques for branding:

- [Master Pages](#): A shared layout (and possibly functionality) that defines the global look and feel to which individual pages can add contents to; it works by defining areas (content placeholders), possibly with default contents, which pages can override, while keeping the global layout.
- [Themes and Skins](#): A named collection of control property settings and CSS stylesheet files that apply to all pages automatically. All **.css** files in a theme folder under **App_Themes** are automatically added to all pages; all control properties are applied to all controls declared in the markup, unless explicitly overridden there.

Master pages

In the master page, we can add the same markup—HTML and ASP.NET control declarations—as we would in a regular page, but we can also add **content placeholders**:

Code Sample 35

```
<%@ Master Language="C#" CodeBehind="Site.Master.cs" Inherits="WebForms.Site" %>
<!DOCTYPE html>
<html>
<head runat="server">
  <title>
    <asp:ContentPlaceholder ID="title" runat="server"/>
  </title>
  <asp:ContentPlaceholder ID="head" runat="server">
    Default Head content
  </asp:ContentPlaceholder>
</head>
<body>
  <form runat="server">
    <asp:ContentPlaceholder ID="header" runat="server" />
    <div>
      <asp:ContentPlaceholder ID="body" runat="server"/>
    </div>
    <asp:ContentPlaceholder ID="footer" runat="server" />
  </form>
</body>
</html>
```

[ContentPlaceholder](#) elements represents the “extensible” areas, or holes, in the master page. Pages may include [Content](#) elements that supply content, thus overriding what is defined in the master page. The master page is normally assigned in the markup of a page:

Code Sample 36

```
<%@ Page Language="C#" MasterPageFile="~/Site.Master" CodeBehind="Default.aspx.cs" Inherits="WebForms.Default" %>
```



```

<asp:Content ContentPlaceHolderID="title" runat="server">
    This is the title
</asp:Content>
<asp:Content ContentPlaceHolderID="head" runat="server">
    This is the overridden head
</asp:Content>
<asp:Content ContentPlaceHolderID="body" runat="server">
    Body content goes here
</asp:Content>

```

You can see that not all content placeholders defined in the master page are being used in the page. Also, the content for the **head** placeholder is overridden in the page, which is perfectly normal.

A content placeholder takes the space of its containing HTML element, so, for example, two master pages could be defined as:

Code Sample 37

```

<!-- master page 1 -->
<table>
    <tr>
        <td><asp:ContentPlaceHolder ID="first" runat="server" /></td>
        <td><asp:ContentPlaceHolder ID="second" runat="server" /></td>
    </tr>
</table>

```

And:

Code Sample 38

```

<!-- master page 2 -->
<table>
    <tr>
        <td><asp:ContentPlaceHolder ID="first" runat="server" /></td>
    </tr>
    <tr>
        <td><asp:ContentPlaceHolder ID="second" runat="server" /></td>
    </tr>
</table>

```

I'm not saying that you should use HTML tables for layout; this is just to make a point: content will appear differently whether master page 1 or 2 is used.

If you recall, the tenant configuration interface that we specified earlier, **ITenantConfiguration**, included a **MasterPage** property. We will leverage this property in order to set automatically the master page to use for our pages, depending on the current tenant.

There are two ways by which a master page can be set:

- Declaratively in the page's markup, using the [MasterPageFile](#) attribute of the [@Page](#) directive
- Programmatically, using the [Page.MasterPageFile](#) property

A master page can only be set programmatically before or during the [Page.PreInit](#) event of the page's lifecycle; after that, any attempt to change it will result in an exception being thrown. If we want to set master pages automatically, without imposing a base page class, we should write a custom module for that purpose:

Code Sample 39

```
public sealed class MultiTenancyModule : IHttpModule
{
    public void Dispose() { }

    public void Init(HttpApplication context)
    {
        context.PreRequestHandlerExecute += OnPreRequestHandlerExecute;
    }

    public static String MasterPagesPath { get; set; }

    private void OnPreRequestHandlerExecute(Object sender, EventArgs e)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        var app = sender as HttpApplication;
        if ((app != null) && (app.Context != null))
        {
            var page = app.Context.CurrentHandler as Page;
            if (page != null)
            {
                page.PreInit += (s, args) =>
                {
                    var p = s as Page;
                    if (!String.IsNullOrEmpty(
                        tenant.MasterPage))
                    {
                        //set the master page
                        p.MasterPageFile =
                            String.Format("{0}/{1}.Master",
                                MasterPagesPath,
                                tenant.MasterPage);
                    }
                };
            }
        }
    }
}
```

The **MasterPagesPath** static property exists so that we can specify an alternative location for our master pages, in case we don't want them located at the site's root. It's safe to leave it blank if your master pages are located there.

That this module hooks up to the current page's [PreInit](#) event and, inside the event handler, checks if the **MasterPage** property for the current tenant is set, and, if so, sets it as the master page of the current page.

Themes and skins

For applying a theme or a skin, we need to create a folder under **App_Themes**:

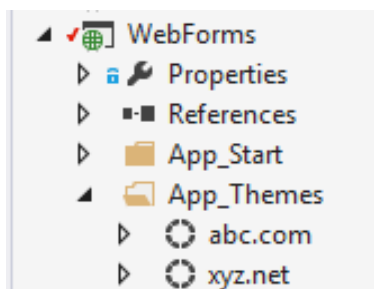


Figure 12: Theme folders

We can have any number of folders, but only one can be set as the current theme.

Themes

A theme consists of one or more **.css** files located inside the theme folder; even inside other folders, ASP.NET makes sure all of them are added to the pages. There are three ways to set a theme:

- Through the **StyleSheetTheme** attribute of the [pages](#) element of **Web.config**
- By setting the **StyleSheetTheme** attribute inside the [Page](#) directive of the **.aspx** file
- Assigning a value to the [StyleSheetTheme](#) property of the [Page](#) class

The first two don't really play well with dynamic contents, but the final one does. It also takes precedence over the other two:

- The **Web.config** setting is the first one that will be applied to all pages, unless specified otherwise in other location.
- Then comes the **.aspx** attribute, which only applies to that specific page.
- Finally, the **page** property, if set until the [Page.PreInit](#) event, is the one to use.

Skins

A skin consists of one or more **.skin** files, also under a folder beneath **App_Themes**. Each file contains multiple control declarations, such as those we would find in an **.aspx**, **.ascx**, or **.master** markup file, with values for some of the control's properties:

```
<asp:TextBox runat="server" SkinID="Email" placeholder="Email address"/>
<asp:TextBox runat="server" Text="<enter value>"/>
<asp:Button runat="server" SkinID="Dark" BackColor="DarkGray" ForeColor="Black"/>
<asp:Button runat="server" SkinID="Light" BackColor="Cyan" ForeColor="Green"/>
<asp:Image runat="server" onerror="this.src = 'missing.png'" />
```

Only properties having the [ThemeableAttribute](#) with a value of **true** (default if no attribute is set) and located in a control also having [ThemeableAttribute](#) set to **true** (or no attribute at all), or plain attributes that do not have a corresponding property (like **placeholder** in the first **TextBox** or **onerror** in the **Image** declaration in Code Sample 37) can be set through a **.skin** file.

Here we can see a few different options:

- Any [TextBox](#) with the [SkinID](#) property set to **Email** will get an HTML5 [placeholder](#) attribute, which is basically a watermark; notice that this is not a property of the [TextBox](#) control, or any of its base classes; instead, it will be translated to an identically named attribute in the generated HTML tag.
- All [TextBox](#) controls without a [SkinID](#) set will have text “<enter value>”
- [Button](#) controls with a [SkinID](#) of “**Dark**” will get a darker aspect than those with a [SkinID](#) of “**Light**”
- Any [Image](#) control that raises a JavaScript [onerror](#) event will have an alternative image set through a JavaScript event handler.



Note: *I chose these examples so that it is clear that themed properties don't necessarily mean only user interface settings.*

The [SkinID](#) property is optional; if set, ASP.NET will try to find any declarations on the current theme's **.skin** files that match its value. Otherwise, it will fall back to control declarations that do not contain a [SkinID](#) attribute.

Like with themes, there are three ways to set a skin folder:

- Through the **Theme** attribute of the [Pages](#) element of **Web.config**
- By setting the **Theme** attribute inside the [Page](#) directive of the .ASPX file
- Assigning a value to the [Theme](#) property of the [Page](#) class.

Skins and stylesheet themes, although different things, are closely related, so it is a good idea to use the same theme folder for both by setting just the [Theme](#) property. If you do, ASP.NET will parse all the **.skin** files and also include any **.css** files found inside the theme folder.

Setting themes automatically

Knowing this, let's change our **MultiTenancyModule** so that, besides setting the master page, it also sets the theme for the current page. Let's create a theme for each tenant, under its name, and set it automatically:

```

public sealed class MultiTenancyModule : IHttpModule
{
    //rest goes here
    private void OnPreRequestHandlerExecute(Object sender, EventArgs e)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        var app = sender as HttpApplication;
        if ((app != null) && (app.Context != null))
        {
            var page = app.Context.CurrentHandler as Page;
            if (page != null)
            {
                page.PreInit += (s, args) =>
                {
                    var p = s as Page;
                    if (!String.IsNullOrEmpty(tenant))
                    {
                        //set the theme
                        p.Theme = tenant.Theme;
                        p.MasterPageFile =
                            String.Format("{0}/{1}.Master",
                                MasterPagesPath, p.MasterPage);
                    }
                };
            }
        }
    }
}

```

Tenant-specific images

You can also keep other contents, such as images, inside the theme folder. This is nice if we want to have different images for each tenant with the same name. The problem is that you cannot reference those images directly in your pages, because you don't know beforehand which theme—meaning, which tenant—will be accessing the page. For example, shall you point the image to “~/App_Themes/abc.com/logo.png” or “~/App_Themes/xyz.net/logo.png”?

Fortunately, ASP.NET Web Forms offers an extensibility mechanism by the name of [Expression Builders](#), which comes in handy here. If you have used resources in Web Forms pages, you have already used the [Resource](#) expression builder. In a nutshell, an expression builder picks up a string passed as a parameter, tries to make sense of it, and then returns some content to the property to which it is bound (an expression builder always runs in the context of a property of a server-side control). How you parse the parameter and what you do with it is up to you.

Let's write an expression builder that takes a partial URL and makes it relative to the current theme. Please consider the `ThemeFileUrl` expression builder:

```

[ExpressionPrefix("ThemeFileUrl")]
public class ThemeFileUrlExpressionBuilder : ExpressionBuilder
{
    public override Object EvaluateExpression(Object target,
        BoundPropertyEntry entry, Object parsedData,
        ExpressionBuilderContext context)
    {
        if (String.IsNullOrEmpty(entry.Expression))
        {
            return base.EvaluateExpression(target, entry, parsedData,
                context);
        }
        else
        {
            return GetThemeUrl(entry.Expression);
        }
    }

    public override Boolean SupportsEvaluate { get { return true; } }

    public override CodeExpression GetCodeExpression(BoundPropertyEntry ent
ry,
        Object parsedData, ExpressionBuilderContext context)
    {
        if (String.IsNullOrEmpty(entry.Expression))
        {
            return new CodePrimitiveExpression(String.Empty);
        }
        else
        {
            return new CodeMethodInvokeExpression(
                new CodeMethodReferenceExpression(
                    new CodeTypeReferenceExpression(this.GetType()),
                    "GetThemeUrl"),
                new CodePrimitiveExpression(entry.Expression));
        }
    }

    public static String GetThemeUrl(String fileName)
    {
        var page = HttpContext.Current.Handler as Page;
        //we can use the Page.Theme property because, at this point, the
MultiTenancyModule has already run, and has set it properly
        var theme = page.Theme;
        var path = (page != null) ? String.Concat("/App_Themes/",
            theme, "/", fileName) : String.Empty;
        return path;
    }
}

```

Before we can use an expression builder, we need to register in the **Web.config** file in section [system.web/compilation/expressionBuilders](#) (do replace “**MyNamespace**” and “**MyAssembly**” with the proper values):

Code Sample 43

```
<system.web>
  <compilation debug="true" targetFramework="4.5">
    <expressionBuilders>
      <add expressionPrefix="ThemeFileUrl" type="MyNamespace
.ThemeFileUrlExpressionBuilder, MyAssembly"/>
    </expressionBuilders>
  </compilation>
</system.web>
```

Now, we can use it in our pages as:

Code Sample 44

```
<asp:Image runat="server" ImageUrl="<%$ ThemeFileUrl:/logo.jpg %%" />
```

The **ThemeFileUrl** expression builder will make sure that the right theme-specific path is used.

Showing or hiding tenant-specific contents

Another use of an expression builder is to show or hide contents directed to a specific tenant without writing code. We create the **MatchTenant** expression builder:

Code Sample 45

```
[ExpressionPrefix("MatchTenant")]
public sealed class MatchTenantExpressionBuilder : ExpressionBuilder
{
    public override Object EvaluateExpression(Object target,
        BoundPropertyEntry entry, Object parsedData,
        ExpressionBuilderContext context)
    {
        if (String.IsNullOrEmpty(entry.Expression))
        {
            return base.EvaluateExpression(target, entry, parsedData,
                context);
        }
        else
        {
            return MatchTenant(entry.Expression);
        }
    }
}
```

```

        public override Boolean SupportsEvaluate { get { return true; } }

        public override CodeExpression GetCodeExpression(BoundPropertyEntry ent
ry,
            Object parsedData, ExpressionBuilderContext context)
        {
            if (String.IsNullOrEmpty(entry.Expression))
            {
                return new CodePrimitiveExpression(String.Empty);
            }
            else
            {
                return new CodeMethodInvokeExpression(
                    new CodeMethodReferenceExpression(
                        new CodeTypeReferenceExpression(
                            this.GetType()), "MatchTenant"),
                    new CodePrimitiveExpression(entry.Expression));
            }
        }

        public static Boolean MatchTenant(String tenant)
        {
            var currentTenant = TenantsConfiguration.GetCurrentTenant();

            if (tenant == currentTenant.Name)
            {
                return true;
            }

            if (tenant.StartsWith("!"))
            {
                if (tenant.Substring(1) != currentTenant.Name)
                {
                    return false;
                }
            }

            return false;
        }
    }
}

```

And we register it in the **Web.config** file:

Code Sample 46

```

<system.web>
  <compilation debug="true" targetFramework="4.5">
    <expressionBuilders>
      <add expressionPrefix="ThemeFileUrl" type="MyNamespace
.ThemeFileUrlExpressionBuilder, MyAssembly"/>
      <add expressionPrefix="MatchTenant" type="MyNamespace

```



```
.MatchTenantExpressionBuilder, MyAssembly"/>
    </expressionBuilders>
</compilation>
</system.web>
```

Two sample usages:

Code Sample 47

```
<asp:Label runat="server" Text="abc.com only" Visible="<%$ MatchTenant:abc.com %>" />

<asp:Label runat="server" Text="Anything but
abc.com" Visible="<%$ MatchTenant:!abc.com %>" />
```

By adding the “!” symbol to the tenant name, we negate the rule.

Security

Authorization

[Authorization in ASP.NET Web Forms](#) is dealt with by the [FileAuthorizationModule](#) and the [UrlAuthorizationModule](#) modules. These modules weren’t really designed with extensibility in mind, so it isn’t exactly easy to make them do what we want.

It is based around two concepts: [authenticated or unauthenticated users](#) and [user roles](#). The built-in authorization mechanisms allow us to define, per path, a page or a filesystem folder under our web application root, if the resource shall be accessed by:

- Anonymous (not authenticated) users
- Specific user names (e.g., “**peter.jackson**”, “**johndoe**”, etc.)
- One of a specific set of roles (e.g., “**Admins**”, “**Developers**”, etc.)
- Everyone (the default)

As you can see, there is no obvious mapping between these concepts and that of tenants, but we can use roles as tenants’ names to restrict access to certain resources. We do so in the **Web.config** file, in the [location](#) sections:

Code Sample 48

```
<location path="AbcComFolder">
    <system.webServer>
        <security>
            <authorization>
                <add accessType="Deny" users="?" />
                <add accessType="Allow" roles="abc.com" />
            </authorization>
        </security>
    </system.webServer>
</location>
```

```
</security>
</system.webServer>
</location>
```

This example uses a role of “**abc.com**”, which is also a tenant’s name. Things get a little bit tricky, though, if we need to use both roles and tenants in access rules. Another option is to restrict calling certain methods by the current tenant. An example using [.NET’s built-in declarative security](#) might be:

Code Sample 49

```
[Serializable]
public sealed class TenantPermission : IPermission
{
    public TenantPermission(params String [] tenants)
    {
        this.Tenants = tenants;
    }

    public IEnumerable<String> Tenants { get; private set; }

    public IPermission Copy()
    {
        return new TenantPermission(this.Tenants.ToArray());
    }

    public void Demand()
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();

        if (!this.Tenants.Any(t => tenant.Name() == t))
        {
            throw new SecurityException
                ("Current tenant is not allowed to access this resource.");
        }
    }

    public IPermission Intersect(IPermission target)
    {
        var other = target as TenantPermission;

        if (other == null)
        {
            throw new ArgumentException
                ("Invalid permission.", "target");
        }

        return new TenantPermission
            (this.Tenants.Intersect(other.Tenants).ToArray());
    }
}
```

```

public Boolean IsSubsetOf(IPermission target)
{
    var other = target as TenantPermission;

    if (other == null)
    {
        throw new ArgumentException
            ("Invalid permission.", "target");
    }

    return this.Tenants.All(t => other.Tenants.Contains(t));
}

public IPermission Union(IPermission target)
{
    var other = target as TenantPermission;

    if (other == null)
    {
        throw new ArgumentException
            ("Invalid permission.", "target");
    }

    return new TenantPermission
        (this.Tenants.Concat(other.Tenants).Distinct()
        .ToArray());
}

public void FromXml(SecurityElement e)
{
    if (e == null)
    {
        throw new ArgumentNullException("e");
    }

    var tenantTag = e.SearchForChildByTag("Tenants");

    if (tenantTag == null)
    {
        throw new ArgumentException
            ("Element does not contain any tenants.", "e");
    }

    var tenants = tenantTag.Text.Split(',').Select(t => t.Trim());

    this.Tenants = tenants;
}

public SecurityElement ToXml()

```

```

    {
        var xml = String
            .Concat("<IPermission class=\"",
                this.GetType().AssemblyQualifiedName,
                "\" version=\"1\" unrestricted=\"false\"><Tenants>",
                String.Join(", ", this.Tenants),
                "</Tenants></IPermission>");
        return SecurityElement.FromString(xml);
    }
}

[Serializable]
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
    AllowMultiple = true, Inherited = false)]
public sealed class RequiredTenantPermissionAttribute : CodeAccessSecurityAttribute
{
    public RequiredTenantPermissionAttribute(SecurityAction action) :
        base(action) {}

    public override IPermission CreatePermission()
    {
        return new TenantPermission(this.Tenants.Split(','))
            .Select(t => t.Trim()).ToArray();
    }

    public String Tenants { get; set; }
}

```

The **RequiredTenantPermissionAttribute** attribute, when applied to a method, does a runtime check, which in this case checks the current tenant against a list of allowed tenants (the **Tenants** property). If there's no match, then an exception is thrown. An example:

Code Sample 50

```

[RequiredTenantPermission(SecurityAction.Demand, Tenants = "abc.com")]
public override void OnLoad(EventArgs e)
{
    //nobody other than abc.com can access this method
    base.OnLoad(e);
}

```

In a later chapter, we will look at another technique that can also be used for restricting accesses.

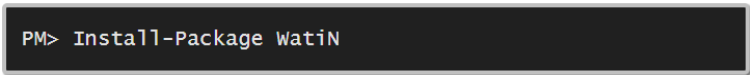
Unit testing

Unit testing with Web Forms is very difficult because it's hard to reproduce the event model that pages and controls use. It isn't impossible, though, and tools such as [WatiN](#) make it possible. WatiN allows instrumenting web sites using our browser of choice. I won't cover it in detail here, but have a look at this sample code, and I think you'll get the picture:

Code Sample 51

```
using (var browser = new IE("http://abc.com"))
{
    Assert.IsTrue(browser.ContainsText("ABC"));
    browser.Button(Find.ByName("btnGo")).Click();
}
```

In order to use WatiN, just add a reference to it using NuGet:



```
PM> Install-Package WatiN
```

Figure 13: Adding the WatiN NuGet package

Chapter 5 ASP.NET MVC

Introduction

[ASP.NET MVC](#) introduces a very different development model. Most people would agree that it is more testable, promotes a separation of responsibilities - views for user interface, controllers for business logic – and is much closer to the HTTP protocol, avoiding some magic that Web Forms employs, which, although useful, can result in increased complexity and decrease of performance.



Note: Nick Harrison wrote [ASP.NET MVC Succinctly](#) for the Succinctly series; be sure to read it for a good introduction to MVC.

Branding

Here we will explore three branding mechanisms offered by MVC:

- Page layouts: The equivalent to Web Forms' master pages, where we shall have a different one for each tenant.
- View locations: Each tenant will have its views stored in a specific folder.
- CSS bundles: Each tenant will have its own CSS bundle (a collection of tenant-specific .css files).

Page layouts

MVC's views have a mechanism similar to Web Forms' master pages, by the name of [Page Layouts](#). A page layout specifies the global structure of the HTML content, leaving “holes” to be filled by pages that use it. By default, ASP.NET MVC uses the “~/Views/Shared/_Layout.cshtml” layout for C# views, and “_Layout.vbhtml” for Visual Basic views.

The only way to use a page layout is to explicitly set it in a view, like in this example, using [Razor](#):

Code Sample 52

```
@{  
    Layout = "~/Views/Shared/Layout.cshtml";  
}
```

In order to avoid repeating code over and over just to set the page layout, we can use one of MVC's extensibility points, the [View Engine](#). Because we will be using [Razor](#) as the view engine of choice, we need to subclass [RazorViewEngine](#), in order to inject our tenant-specific page layout.

Our implementation will look like this:

Code Sample 53

```
public sealed class MultitenantRazorViewEngine : RazorViewEngine
{
    public override ViewEngineResult FindView(ControllerContext ctx,
String viewName, String masterName, Boolean useCache)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        //the third parameter to FindView is the page layout
        return base.FindView(controllerContext, viewName,
            tenant.MasterPage, useCache);
    }
}
```

In order to use our new view engine, we need to replace the existing one in the [ViewEngines.Engines](#) collection; this should be done in some method spawned from [Application_Start](#):

Code Sample 54

```
ViewEngines.Engines.Clear();

ViewEngines.Engines.Add(new MultitenantRazorViewEngine());
```

Finally, we must make sure our views set the page layout ([Layout](#) property) from **ViewBag.Layout**, which is mapped to whatever is returned in the third parameter of [FindView](#):

Code Sample 55

```
@{
    Layout = ViewBag.Layout;
}
```

View locations

By default, the view engine will look for views (**.cshtml** or **.vbhtml**) in a pre-defined collection of virtual paths. What we would like to do is look for a view in a tenant-specific path first, and if it's not found there, fall back to the default locations. Why would we want that? Well, this way, we can design our views to have a more tenant-specific look, more than we could do just by changing page layouts.

We will make use of the same view engine introduced in the previous page, and will adapt it for this purpose:

Code Sample 56

```
public sealed class MultitenantRazorViewEngine : RazorViewEngine
{
    private Boolean pathsSet = false;

    public MultitenantRazorViewEngine() : this(false) { }

    public MultitenantRazorViewEngine(Boolean usePhysicalPathsPerTenant)
    {
        this.UsePhysicalPathsPerTenant = usePhysicalPathsPerTenant;
    }

    public Boolean UsePhysicalPathsPerTenant { get; private set; }

    private void SetPaths(ITenantConfiguration tenant)
    {
        if (this.UsePhysicalPathsPerTenant)
        {
            if (!this.pathsSet)
            {
                this.ViewLocationFormats = new String[]
                {
                    String.Concat("~/Views/",
                        tenant.Name,("/{1}/{0}.cshtml"),
                    String.Concat("~/Views/",
                        tenant.Name,("/{1}/{0}.vbhtml")
                }
                .Concat(this.ViewLocationFormats).ToArray();
                this.pathsSet = true;
            }
        }
    }

    public override ViewEngineResult FindView(ControllerContext ctx,
        String viewName, String masterName, Boolean useCache)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        //the third parameter to FindView is the page layout
        return base.FindView(controllerContext, viewName,
            tenant.MasterPage, useCache);
    }
}
```

This requires that the view engine is configured with the **usePhysicalPathsPerTenant** set:


```
ViewEngines.Engines.Add(new MultitenantRazorViewEngine(true));
```

CSS Bundles

[CSS Bundling and Minification](#) is integrated in the two major flavors of ASP.NET: Web Forms and MVC. In Web Forms, because it has the themes mechanism (see Themes and), it is probably not much used for branding, but MVC does not have an analogous mechanism.

A CSS bundle consists of a name and a set of **.css** files located in any number of folders. We will create, for each of the registered tenants, an identically-named bundle containing all of the files in a folder named from the tenant configuration's **Theme** property under the folder **~/Content**. Here's how:

Code Sample 58

```
public static void RegisterBundles(BundleCollection bundles)
{
    foreach (var tenant in TenantsConfiguration.GetTenants())
    {
        var virtualPath = String.Format("~/{0}", tenant.Name);
        var physicalPath = String.Format("~/Content/{0}",
            tenant.Theme);
        if (!BundleTable.Bundles.Any(b => b.Path == virtualPath))
        {
            var bundle = new StyleBundle(virtualPath)
                .IncludeDirectory(physicalPath, "*.css");
            BundleTable.Bundles.Add(bundle);
        }
    }
}
```

This method needs to be called after the tenants are registered:

Code Sample 59

```
RegisterBundles(BundleTable.Bundles);
```

Tenant **“abc.com”** will get a CSS bundle called **“~/abc.com”** containing all **.css** files under **“~/Content/abc.com”**.

This is not all, however; in order to actually add the CSS bundle to a view, we need to add an explicit call on the view, like this one:

Code Sample 60

```
@Styles.Render("~/abc.com")
```

However, if we are to hardcode the tenant's name, this won't scale. This would be okay for tenant-specific layout pages, but not for generic views. What we need is a mechanism to automatically supply, on each request, the right bundle name. Fortunately, we can achieve this with an [action filter](#):

Code Sample 61

```
public sealed class MultitenantActionFilter : IActionFilter
{
    void IActionFilter.OnActionExecuted(ActionExecutedContext ctx) { }

    void IActionFilter.OnActionExecuting(ActionExecutingContext ctx)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        ctx.Controller.ViewBag.Tenant = tenant.Name;
        ctx.Controller.ViewBag.TenantCSS = String.Concat("~/",
            filterContext.Controller.ViewBag.Tenant);
    }
}
```

This action filter implements the [IActionFilter](#) interface, and all it does is inject two values in the [ViewBag](#) collection, the tenant name (**Tenant**), and the CSS bundle's name (**TenantCSS**). Action filters can be registered in a number of ways, one of which is the [GlobalFilters.Filters](#) collection:

Code Sample 62

```
GlobalFilters.Filters.Add(new MultitenantActionFilter());
```

With this on, all we need to add custom tenant-specific bundles on a view is:

Code Sample 63

```
@Styles.Render(ViewBag.TenantCSS)
```

Security

We might want to restrict some controller actions for some tenant. MVC offers a hook called an [authorization filter](#) that can be used to allow or deny access to a given controller or action method. There's a base implementation, [AuthorizeAttribute](#), which grants access only if the current user is authenticated. The implementation allows extensibility, and that's what we are going to do:

Code Sample 64

```
[Serializable]
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    AllowMultiple = false, Inherited = true)]
public sealed class AllowedTenantsAttribute : AuthorizeAttribute
{
    public AllowedTenantsAttribute(params String [] tenants)
    {
        this.Tenants = tenants;
    }

    public IEnumerable<String> Tenants { get; private set; }

    protected override Boolean AuthorizeCore(HttpContextBase ctx)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        return this.Tenants.Any(x => x == tenant.Name);
    }
}
```

When applied to a controller method or class with one or more tenants as parameters, it will only grant access to the specified tenants:

Code Sample 65

```
public class SomeController : Controller
{
    [AllowedTenants("abc.com")]
    public ActionResult Something()
    {
        //this can only be accessed by abc.com
        //return something
    }

    [AllowedTenants("xyz.net", "abc.com")]
    public ActionResult OtherThing()
    {
        //this can be accessed by abc.com and xyz.net
        //return something
    }
}
```

The next chapter describes another technique for restricting access to resources without the need for code.

Unit Testing

Testing MVC controllers and actions is straightforward. In the context of multitenant applications, as we have been talking in this book, we just need to set up our test bench according to the right tenant identification strategy. For example, if you are using [NUnit](#), you would do it before each test, in a method decorated with the [SetUpFixtureAttribute](#):

Code Sample 66

```
[SetUpFixture]
public static class MultitenantSetup
{
    [SetUp]
    public static void Setup()
    {
        var req = new HttpRequest(
            filename: String.Empty,
            url: "http://abc.com/",
            queryString: String.Empty);
        req.Headers["HTTP_HOST"] = "abc.com";
        //add others as needed

        var response = new StringBuilder();
        var res = new HttpResponse(new StringWriter(response));

        var ctx = new HttpContext(req, res);

        var principal = new GenericPrincipal(
            new GenericIdentity("Username"), new [] { "Role" });
        var culture = new CultureInfo("pt-PT");

        Thread.CurrentThread.CurrentCulture = culture;
        Thread.CurrentThread.CurrentUICulture = culture;

        HttpContext.Current = ctx;
        HttpContext.Current.User = principal;
    }
}
```

Chapter 6 Web Services

Introduction

Web services, of course, are also capable of handling multitenant requests. In the .NET world, there are basically two APIs for implementing web services: [Windows Communication Foundation](#) (WCF) and the new [Web API](#). Although they share some overlapping features, their architecture and basic concepts are quite different. While a thorough discussion of these APIs is outside the scope of this book, let's see some of the key points that are relevant to multitenancy.

WCF

WCF, by default, does not make use of the ASP.NET pipeline, which means that [HttpContext.Current](#) is not available. This is so that WCF has a more streamlined, focused model where the classic pipeline is not needed (but it is possible to enable it).

If we do not want to use the ASP.NET pipeline, we need to change our implementation of the tenant identification strategy. Inside a WCF method call, it is always possible to access the current request context through the [OperationContext.Current](#) or [WebOperationContext.Current](#) (for REST web services) properties. So, we need to write one implementation of the Tenant Identification strategy that knows how to fetch this information:

Code Sample 67

```
public class WcfHostHeaderTenantIdentification : ITenantIdentifierStrategy
{
    public String GetCurrentTenant(RequestContext context)
    {
        var request = WebOperationContext.Current.IncomingRequest;
        var host = request.Headers["Host"];

        return host.Split(':').First().ToLower();
    }
}
```

On the other side, if ASP.NET pipeline is an option, we just need to enable it through XML configuration, through the [aspNetCompatibilityEnabled](#) attribute:

Code Sample 68

```
<system.serviceModel>
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
</system.serviceModel>
```

Or, we can enable it through an attribute in code:

```
[AspNetCompatibilityRequirements(  
    RequirementsMode = AspNetCompatibilityRequirementsMode.Required)]  
public class DateService : IDateService  
{  
    //rest goes here  
}
```



***Note:** For a good discussion of the ASP.NET compatibility mode, check out [this article](#).*

Web API

[ASP.NET Web API](#) is the new API for writing REST web services with the .NET framework. It closely follows the ASP.NET MVC model, and is thus based on controllers and actions.

Currently, it can be hosted in either IIS (the traditional way) or in an OWIN-supported host. In the first case, all of the classic ASP.NET APIs (**System.Web.DLL**) are available, including [HttpContext.Current](#), so you can use the same strategy implementations that were described for MVC and Web Forms. For OWIN, read Chapter 8 for some tips.

Unit Testing

Whenever the ASP.NET classic APIs are available, we can follow the prescription described in the Unit Testing section of Chapter 5. For OWIN, jump to Chapter 8.

Chapter 7 Routing

Introduction

Routing is a powerful mechanism for managing the URLs of your web applications. It allows us to have nice, friendly URLs that can provide a useful insight on what they are meant for. Other than that, it also offers security and extensibility features that shouldn't be discarded.

Routing

Routing is an essential part of MVC, but was actually introduced to Web Forms in ASP.NET 3.5 SP1. It depends on an ASP.NET module ([UrlRoutingModule](#)) that is configured in the global **Web.config** file, so we normally don't need to worry about it, and also on a routing table, which is configured through code. We won't be looking extensively at ASP.NET routing, but we will explore two of its extensibility points: route handlers and route constraints.

Route handlers

A route handler is a class that is actually responsible for processing the request, and consists of an implementation of the [IRouteHandler](#) interface. ASP.NET includes one implementation for MVC ([MvcRouteHandler](#)), but you can easily roll out your own for Web Forms. So, what is it useful for? Well, you can add your own code logic to it and rest assured that it will only apply to the route specified. Let's look at an example:

Code Sample 70

```
var customRoute = new Route("{controller}/{action}/{id}",
    new RouteValueDictionary(new { controller = "Home", action = "Index",
        id = UrlParameter.Optional })), new MyRouteHandler());
RouteTable.Routes.Add(customRoute);
```

We are registering an ordinary route, with the sole difference that we are passing a custom route handler, **MyRouteHandler**, a sample implementation of which is presented next:

Code Sample 71

```
class MyRouteHandler : MvcRouteHandler
{
    protected override IHttpHandler GetHttpHandler(RequestContext
        requestContext)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        //do something with the current tenant
        return base.GetHttpHandler(requestContext);
    }
}
```

```
}
```

This handler will apply to all requests matching the given URL template of “{controller}/{action}/{id}”, regardless of the current tenant. By calling its base implementation, everything will work as expected. We can, however, restrict our routes based on some condition—enter route constraints.

Before we go to that, it is important to notice that these examples are for MVC, but we can do something similar for Web Forms; the only difference is the base class, [PageRouteHandler](#), in this case:

Code Sample 72

```
class PageRouteHandler : PageRouteHandler
{
    public WebFormsRouteHandler(String virtualPath) : base(virtualPath) {
    }

    public override IHttpHandler GetHttpHandler(RequestContext
        requestContext)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        //do something with the current tenant
        return base.GetHttpHandler(requestContext);
    }
}
```

Also, registration is slightly different:

Code Sample 73

```
var customRoute = new Route(String.Empty, new WebFormsRouteHandler
    ("~/Default.aspx"));
RouteTable.Routes.Add(customRoute);
```

Route constraints

A route constraint needs to implement the framework interface [IRouteConstraint](#). When such an implementation is applied to a route, the [UrlRoutingModule](#) will check first with it to see if the current request can be matched to a route. Here’s how we can specify one (or more) route constraints:

Code Sample 74

```
var customRoute = new Route("{controller}/{action}/{id}",
    new RouteValueDictionary(new { controller = "Home", action = "Index",
        id = UrlParameter.Optional }), new RouteValueDictionary(
        new Dictionary<String, Object> { { "my", new MyRouteConstraint() } } ),
    new MyRouteHandler());
RouteTable.Routes.Add(customRoute);
```


And, finally, a constraint example that takes into account the current tenant:

Code Sample 75

```
class MyRouteConstraint : IRouteConstraint
{
    public Boolean Match(HttpContextBase httpContext, Route route,
        String parameterName, RouteValueDictionary values,
        RouteDirection routeDirection)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        //do some check with the current tenant and return true or
false
        return true;
    }
}
```

IIS Rewrite Module

Another option for restricting access based on the current tenant lies in the [IIS Rewrite Module](#). This module is a free download from Microsoft, and it allows us to specify rules for controlling access and redirecting requests, all from the **Web.config** file. Essentially, a rewrite rule consists of:

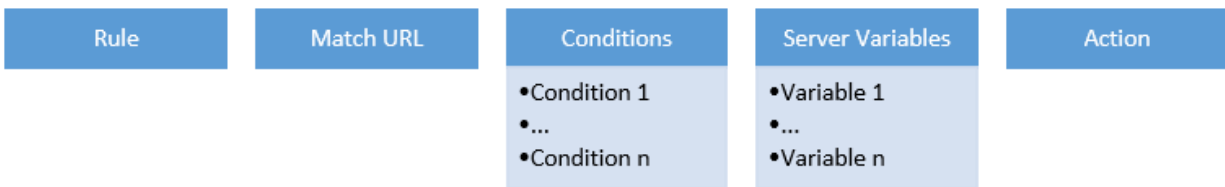


Figure 14: IIS Rewrite Module rule components

The **rule** element defines the **name**, if the rule should be **enabled** or not, and the pattern matching syntax used in all fuzzy expression attributes (**patternSyntax**), which can be:

- **Wildcard**: No regular expression, but just wildcard (*) matching
- **ECMAScript**: ECMAScript standard regular expressions
- **ExactMatch**: An exact match

Code Sample 76

```
<rewrite>
  <rules>
    <rule enabled="true" patternSyntax="ECMAScript"
      name="ABC.COM only">
      <!-- rest goes here -->
    </rule>
  </rules>
</rewrite>
```

Then there's the **match** element. In it we specify a URL pattern (**url**), which can be a specific page, if the match should be done in a case-insensitive way (**ignoreCase**), and even if the matching should be reverted (**negate**), that is, accept URLs that don't match the pattern:

Code Sample 77

```
<rewrite>
  <rules>
    <rule enabled="true" patternSyntax="ECMAScript"
          name="ABC.COM only">
      <match url="/abc\.com\/.*" ignoreCase="true" />
      <!-- rest goes here -->
    </rule>
  </rules>
</rewrite>
```

We can then add any number of **conditions**; these are normally constructed by checking [Server Variables](#) (**input**) against one of the following match types (**matchType**):

- **Pattern**: patterns, the default
- **IsFile**: checking if a file exists
- **IsDirectory**: checking if a directory exists

Conditions can be evaluated differently (**logicalGrouping**):

- **MatchAll**: All rules need to be evaluated positively (the default).
- **MatchAny**: At least one rule is positive.

Conditions can also be **negated**, as show in this example:

Code Sample 78

```
<rewrite>
  <rules>
    <rule enabled="true" patternSyntax="ECMAScript"
          name="ABC.COM only">
      <match url="/abc\.com\/.*" ignoreCase="true" />
      <conditions>
        <add input="{HTTP_HOST}" matchType="Pattern"
              pattern="abc.com" negate="true" />
      </conditions>
      <!-- rest goes here -->
    </rule>
  </rules>
</rewrite>
```

Server variables are more or less standard across web servers; they are defined in [RFC 3875](#), and include:

- **HTTP_HOST**: the server host name, as sent in the request
- **HTTP_REFERER**: the URL that the browser came from
- **HTTP_USER_AGENT**: the browser being used to access the page
- **HTTPS**: whether the request is using HTTPS or not
- **PATH_INFO**: a path following the actual server resource
- **QUERY_STRING**: the query string of the request
- **REMOTE_ADDR**: the client's address
- **REMOTE_HOST**: the client's host name
- **REMOTE_USER**: the authenticated user

Finally, we specify an **action** to be triggered if the **match** and **conditions** are successfully evaluated. An action can be one of:

- **AbortRequest**: stops the current request; useful for denying access
- **None**: does not do anything
- **Rewrite**: internally rewrites the request to something different
- **Redirect**: redirects the browser to a different resource
- **CustomResponse**: sends custom HTTP headers, a status code, and subcode

This example denies access to everyone except the **abc.com** tenant to everything under **/abc.com**:

Code Sample 79

```
<rewrite>
  <rules>
    <rule enabled="true" patternSyntax="ECMAScript"
      name="ABC.COM only">
      <match url="/abc\.com\/.*" ignoreCase="true" />
      <conditions>
        <add input="{HTTP_HOST}" matchType="Pattern"
          pattern="abc.com" negate="true"/>
      </conditions>
      <action type="AbortRequest" statusCode="401"
        statusDescription="Denied for others than
ABC.COM" />
    </rule>
  </rules>
</rewrite>
```

Finally, we can specify one or more server variables (**serverVariables**). These can be used either for being referenced in conditions, or to pass some kind of information to the handlers that will be processing the request. For example, if we were to capture the tenant using the Host header strategy, we could have:

Code Sample 80

```
<rewrite>
  <rules>
    <rule enabled="true" patternSyntax="ECMAScript">
```

```

        name="ABC.COM only">
        <serverVariables>
            <set name="TENANT" value="{HTTP_HOST}" />
        </serverVariables>
        <match url="\abc\.com\/.*" ignoreCase="true" />
        <conditions>
            <add input="{TENANT}" matchType="Pattern"
                pattern="abc.com" negate="true"/>
        </conditions>
        <action type="AbortRequest" statusCode="401"
            statusDescription="Denied for others than
ABC.COM" />
    </rule>
</rules>
</rewrite>

```



This code is meant for demo purposes only. It does not have any kind of validation and just returns the requested host name in lowercase. Real-life code should be slightly more complex.

The IIS Rewrite Module also features a capture mechanism that allows us to retrieve parts of matching regular expressions. For example, if we were to use the Query string strategy for determining the tenant, we could use:

Code Sample 81

```

<conditions>
    <add input="{QUERY_STRING}" pattern="&?tenant=(.+) " />
    <!-- the tenant passed in the query string will be in {C:1} -->
</conditions>
<action type="Rewrite" url="/tenant/{C:1}"/>

```

We would use **{R:n}** for references captured in the **match** element and **{C:n}** for those captured in **conditions**, where 0 would return the whole matching string and 1, ..., n, each of the captured elements, inside “()”. This simple example redirects all requests for “tenant=something” to “/tenant/something”. It can be useful, for example, if you want to redirect all requests from a certain tenant from one resource to another, namely, for resources that shouldn’t be accessed by some customer.


Chapter 8 OWIN

Introduction

The [Open Web Interface for .NET](#) (OWIN) is a Microsoft specification that defines a contract for .NET applications to interact with web servers. This includes the pipeline for execution of HTTP requests and contracts for services to sit in between (middleware). It also happens to be the basis for the upcoming ASP.NET 5.

OWIN allows you to decouple your service application from any particular web server, such as IIS. Although it can use IIS, it can also be self-hosted (like WCF) through OWIN's web server implementation, [OwinHttpListener](#).

You will need to add a reference to the [Microsoft.Owin.Host.HttpListener](#) NuGet package:



```
PM> Install-Package Microsoft.Owin.Host.HttpListener
```

Figure 15: Installing the *HttpListener* NuGet package

Why should we care with OWIN? Well, for once, it gives you a consistent pipeline that is more aligned with the way ASP.NET will be in the near future: no more Web Forms—MVC will be configured through OWIN!



Note: For a good introduction to OWIN, check out the book [OWIN Succinctly](#), by Ugo Lattanzi and Simone Chiazza, also in the Succinctly collection.

Registering services

The OWIN pipeline starts with a call to a method called **Configuration** of a class called **Startup**. This is by convention; there is no base class or interface that dictates this, and both the class and method can even be static. If we want to change this bootstrap method, we can do so by adding an [OwinStartupAttribute](#) at the assembly level or a “[owin:appStartup](#)” entry to the **Web.config**'s [appSettings](#) section. Here we register a middleware component that in return, will register all our services (like tenant identification and registration):

Code Sample 82

```
public static class Startup
{
    public static void Configuration(IAppBuilder builder)
    {
        builder.UseStaticFiles();
        builder.UseDefaultFiles();
    }
}
```

```

        //rest goes here
        builder.Use<MultitenancyMiddleware>();
        //rest goes here
    }
}

```

The middleware class inherits from [OwinMiddleware](#)—there are other options, but this is my favorite one. A middleware class receives in its constructor a pointer to the next middleware in the pipeline, and should return the result of its invocation in its [Invoke](#) method.

An example:

Code Sample 83

```

public class MultitenancyMiddleware : OwinMiddleware
{
    public MultitenancyMiddleware(OwinMiddleware next) : base(next) {
    }

    public override Task Invoke(IOwinContext context)
    {
        //services registration
        context.Set<ITenantLocationStrategy>(<
            typeof(ITenantLocationStrategy).FullName,
            new MefTenantLocationStrategy(
                typeof(Common.ContextRepository).Assembly));
        context.Set<ITenantIdentifierStrategy>(<
            typeof(ITenantIdentifierStrategy).FullName,
            new HostHeaderTenantIdentifierStrategy());
        context.Set<IConfiguration>(<
            typeof(IConfiguration).FullName,
            new AppSettingsConfiguration());

        //rest goes here

        return this.Next.Invoke(context);
    }
}

```



Note: Right now, you cannot use OWIN with ASP.NET MVC or Web Forms, because these depend on the *System.Web.DLL*, the core of the ASP.NET framework, but ASP.NET MVC 6 will work on top of OWIN.

So, if you are to use OWIN, and you need to resolve one of our bootstrap services, you can either stick with your choice of IoC container and Common Service Locator, or you can use OWIN's internal implementation:

```
public override Task Invoke(IOwinContext context)
{
    var tls = context.Get<ITenantLocationStrategy>(
        typeof(ITenantLocationStrategy).FullName);
    return this.Next.Invoke(context);
}
```

OWIN also supports [ASP.NET Identity](#) for authentication. Just add NuGet package [Microsoft.AspNet.Identity.Owin](#) and make any changes to the added classes:

```
PM> Install-Package Microsoft.AspNet.Identity.Owin
```

Figure 16: ASP.NET Identity package for OWIN

Web API

Unlike MVC, Web API can be used with OWIN. You will need to register an implementation, such as the one provided by the Microsoft NuGet package [Microsoft.AspNet.WebApi.Owin](#):

```
PM> Install-Package Microsoft.AspNet.WebApi.Owin
```

Figure 17: Web API OWIN package

Then you will need to register the Web API service in the aforementioned **Configuration** method, using the **UseWebApi** extension method:

```
public static class Startup
{
    public static void Configuration(IAppBuilder builder)
    {
        builder.UseWebApi(new HttpConfiguration());
        //rest goes here
    }
}
```

Replacing System.Web

OWIN came into being because of the desire to replace the **System.Web** architecture. One key class in **System.Web** is [HttpContext](#), which no longer exists in OWIN. This poses a problem: our interfaces that were introduced earlier, namely **ITenantIdentifierStrategy**, rely on classes that are part of **System.Web**, so we need to figure out how we can achieve the same results without it. The OWIN pipeline is significantly simpler than ASP.NET; it follows a pattern known as [Chain of Responsibility](#), where each stage (called middleware) in the pipeline calls the next one, eventually doing something before and after that. So, if we want to make available some services to the other, we need to do it in the first stage:

Code Sample 86

```
public class OwinHostHeaderTenantIdentifierStrategy : OwinMiddleware
{
    public OwinHostHeaderTenantIdentifierStrategy(OwinMiddleware next):
        base(next) { }

    public override Task Invoke(IOwinContext context)
    {
        context.Request.Environment["Tenant"] =
            TenantsConfiguration.GetTenants().Single(x =>
                x.Name == context.Request.Host.Value.Split(':')[0]
                    .First().ToLower());

        return this.Next.Invoke(context);
    }
}
```

The equivalent of the [HttpContext](#) in OWIN is the implementation of the [IOwinContext](#) interface: it has [Request](#) and [Response](#) objects, plus a couple of other useful methods and properties. In this example, we are storing the current tenant, as identified by the host header, in an environment variable (the [Environment](#) collection). OWIN does not dictate how to do it, so please feel free to do it any way you prefer. The important thing is that this middleware needs to be inserted on the pipeline before anything else that might need it:

Code Sample 87

```
public static class Startup
{
    public static void Configuration(IAppBuilder builder)
    {
        builder.UseWebApi(new HttpConfiguration());
        builder.UseStaticFiles();
        builder.UseDefaultFiles();
        //rest goes here
        builder.Use<MultitenancyMiddleware>();
        builder.Use<OwinHostHeaderTenantIdentifierStrategy>();
        //rest goes here
    }
}
```



```
}
```

Unit testing

In order to unit test an OWIN setup, we may need to inject a couple of properties, such as the current host, so that it can be caught by our implementation of the Host header strategy:

Code Sample 88

```
public static void Setup(this IOwinContext context)
{
    context.Request.Host = new HostString("abc.com");
}
```

Other strategies don't need any particular setup.

Chapter 9 Application Services

Introduction

Any application that does something at least slightly elaborate depends on some services. Call them common concerns, application services, middleware, or whatever. The challenge here is that we cannot just use these services without providing them with some context, namely, the current tenant. For example, consider cache: you probably wouldn't want something cached for a specific tenant to be accessible by other tenants. Here we will see some techniques for preventing this by leveraging what ASP.NET and the .NET framework already offer.

Inversion of Control

Throughout the code, I have been using the Common Service Locator to retrieve services regardless of the Inversion of Control container used to actually register them, but in the end, we are using Unity. When we do actually register them, we have a couple of options:

- Register static instances
- Register class implementations
- Register injection factories

For those services that need context (knowing the current tenant), injection factories are our friends, because this information can only be obtained when the service is actually requested. Using Unity, we register them using code such as this:

Code Sample 89

```
var container = UnityConfig.GetConfiguredContainer();
container.RegisterType<IContextfulService>(new PerRequestLifetimeManager(),
    new InjectionFactory(x =>
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        return new ContextfulServiceImpl(tenant.Name);
    }));
```

Noteworthy:

- The [PerRequestLifetimeManager](#) is a Unity lifetime manager implementation that provides a per-request lifetime for registered components, meaning components will only be created in the scope of the current HTTP request if they weren't already created. Otherwise, the same instance will always be returned. Disposable components are dealt with appropriately at the end of the request.
- [InjectionFactory](#) takes a delegate that just returns a pre-created instance. In this example, we are building a bogus service implementation, `ContextfulServiceImpl`, which takes a constructor parameter that is the current tenant name.



Tip: *It doesn't make much sense using `InjectionFactory` with lifetime managers other than `PerRequestLifetimeManager`.*



Note: *Like I said previously, you are not tied to Unity—you can use whatever IoC container you like, provided that (for the purpose of the examples in this book) it offers an adapter for the Common Service Locator.*

Remember the interface that represents a tenant's configuration, `ITenantConfiguration`? If you do, you know that it features a general-purpose, indexed collection, `Properties`. We can implement a custom lifetime manager that stores items in the current tenant's `Properties` collection in a transparent way:

Code Sample 90

```
public sealed class PerTenantLifetimeManager : LifetimeManager
{
    private readonly Guid id = Guid.NewGuid();
    private static readonly ConcurrentDictionary<String,
        ConcurrentDictionary<Guid, Object>> items =
        new ConcurrentDictionary<String,
            ConcurrentDictionary<Guid, Object>>();

    public override Object GetValue()
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        ConcurrentDictionary<Guid, Object> registrations = null;
        Object value = null;

        if (items.TryGetValue(tenant.Name, out registrations))
        {
            registrations.TryGetValue(this.id, out value);
        }

        return value;
    }

    public override void RemoveValue()
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        ConcurrentDictionary<Guid, Object> registrations = null;
    }
}
```

```

        if (items.TryGetValue(tenant.Name, out registrations))
        {
            Object value;
            registrations.TryRemove(this.id, out value);
        }

    }

    public override void SetValue(Object newValue)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        var registrations = items.GetOrAdd(tenant.Name,
            new ConcurrentDictionary<Guid, Object>());
        registrations[this.id] = newValue;
    }
}

```

Caching

Caching is one of those techniques that can dramatically improve the performance of an application, because it keeps in memory data that is potentially costly to acquire. We need to be able to store data against a key, where several tenants can use the same key. The trick here is to, behind the scenes, join the supplied key with a tenant-specific identifier. For instance, a typical caching service interface might be:

Code Sample 91

```

public interface ICache
{
    void Remove(String key, String regionName = null);
    Object Get(String key, String regionName = null);
    void Add(String key, Object value, DateTime absoluteExpiration,
        String regionName = null);
    void Add(String key, Object value, TimeSpan slidingExpiration,
        String regionName = null);
}

```

And an implementation using the ASP.NET built-in cache:

Code Sample 92

```

public sealed class AspNetMultitenantCache : ICache, ITenantAwareService
{
    public static readonly ICache Instance = new AspNetMultitenantCache();

    private AspNetMultitenantCache()
    {
        //private constructor since this is meant to be used as a singleton
    }
}

```

```

    }

    private String GetTenantKey(String key, String regionName)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant().Name;
        key = String.Concat(tenant, ":", regionName, ":", key);
        return key;
    }

    public void Remove(String key, String regionName = null)
    {
        HttpContext.Current.Cache.Remove(this.GetTenantKey(key,
            regionName));
    }

    public Object Get(String key, String regionName = null)
    {
        return HttpContext.Current.Cache.Get(this.GetTenantKey(key,
            regionName));
    }

    public void Add(String key, Object value, DateTime absoluteExpiration,
        String regionName = null)
    {
        HttpContext.Current.Cache.Add(this.GetTenantKey(key, regionName),
            value, null, absoluteExpiration, Cache.NoSlidingExpiration,
            CacheItemPriority.Default, null);
    }

    public void Add(String key, Object value, TimeSpan slidingExpiration,
        String regionName = null)
    {
        HttpContext.Current.Cache.Add(this.GetTenantKey(key, regionName),
            value, null, Cache.NoAbsoluteExpiration, slidingExpiration,
            CacheItemPriority.Default, null);
    }
}

```



Note: Do not worry about *ITenantAwareService*; it is just a marker interface I use for those services that know about the current tenant.

Nothing too fancy here—we just wrap the user-supplied key with the current tenant's ID and region name. This is implemented as a singleton, because there's only one ASP.NET cache, and there's no point in having multiple instances of `AspNetMultitenantCache`, without any state and all pointing to the same cache. That's what the `ITenantAwareService` interface states: it knows who we are addressing, so there's no need to have different instances per tenant.



Note: Notice the *AspNet* prefix on the class name. It is an indication that this class requires **ASP.NET** to work.

Registering the cache service is easy, and we don't need to use [InjectionFactory](#), since we will be registering a singleton:

Code Sample 93

```
container.RegisterInstance<ICache>(AspNetMultitenantCache.Instance);
```

Configuration

I am not aware of any mid-sized application that does not require some configuration of any kind. The problem here is the same: two tenants might share the same configuration keys, while at the same time they would expect different values. Let's define a common interface for the basic configuration features:

Code Sample 94

```
public interface IConfiguration
{
    Object GetValue(String key);
    void SetValue(String key, Object value);
    Object RemoveValue(String key);
}
```

Let's also define a multitenant implementation that uses [appSettings](#) in a per-tenant configuration file as the backing store:

Code Sample 95

```
public class AppSettingsConfiguration : IConfiguration, ITenantAwareService
{
    public static readonly IConfiguration Instance =
        new AppSettingsConfiguration();

    private AppSettingsConfiguration()
    {
    }

    private void Persist(Configuration configuration)
    {
        configuration.Save();
    }

    private Configuration Configuration
```

```

{
    get
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        var configMap = new ExeConfigurationFileMap();
        configMap.ExeConfigFilename = String.Format(
            AppDomain.CurrentDomain.BaseDirectory,
            tenant.Name, ".config");

        var configuration = ConfigurationManager
            .OpenMappedExeConfiguration(configMap,
                ConfigurationUserLevel.None);

        return configuration;
    }
}

public Object GetValue(String key)
{
    var entry = this.Configuration.AppSettings.Settings[key];
    return (entry != null) ? entry.Value : null;
}

public void SetValue(String key, Object value)
{
    if (value == null)
    {
        this.RemoveValue(key);
    }
    else
    {
        var configuration = this.Configuration;
        configuration.AppSettings.Settings
            .Add(key, value.ToString());
        this.Persist(configuration);
    }
}

public Object RemoveValue(String key)
{
    var configuration = this.Configuration;
    var entry = configuration.AppSettings.Settings[key];

    if (entry != null)
    {
        configuration.AppSettings.Settings.Remove(key);
        this.Persist(configuration);
        return entry.Value;
    }
}

```

```
        return null;
    }
}
```



Note: This class is totally agnostic as to web or non-web; it can be used in any kind of project. It also implements *ITenantAwareService*, for the same reason as before.

With this implementation, we can have one file per tenant, say, **abc.com.config**, or **xyz.net.config**, and the syntax is going to be identical to that of the ordinary .NET configuration files:

Code Sample 96

```
<configuration>
  <appSettings>
    <add key="Key" value="Value"/>
  </appSettings>
</configuration>
```

This approach is nice because we can even change the files at runtime, and we won't cause the ASP.NET application to restart, which would happen if we were to change the **Web.config** file.

We'll use the same pattern for registration:

Code Sample 97

```
container.RegisterInstance<IConfiguration>(AppSettingsConfiguration.Instance);
```

Logging

It would be cumbersome and rather silly to implement another logging framework, when there are so many good ones to choose from. For this book, I have chosen the [Enterprise Library Logging Application Block](#) (ELLAB). You will probably want to add support for it through NuGet:

```
PM> Install-Package EnterpriseLibrary.Logging
```

Figure 18: Installing the Enterprise Library Logging Application Block

ELLAB offers a single API that you can use to send logs to a number of sources, as outlined in the following picture:

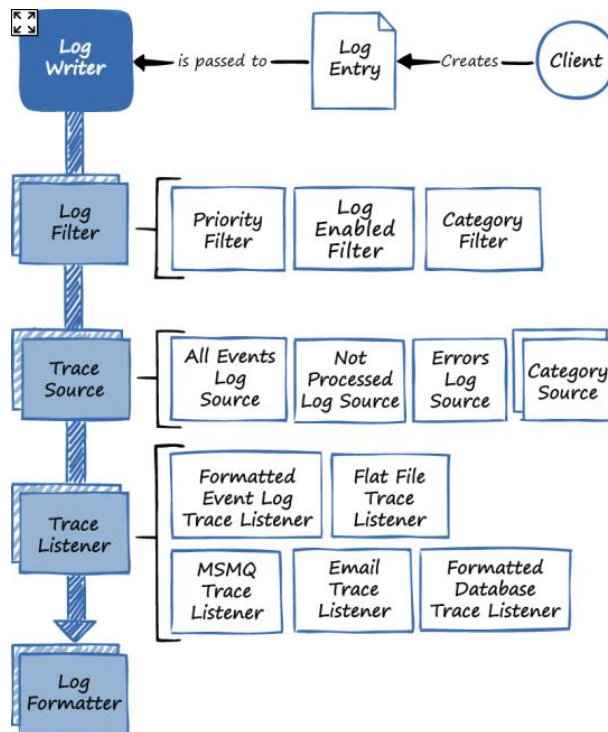


Figure 19: Enterprise Library Logging Application Block architecture

These sources include:

- Text files ([RollingFlatFileTraceListener](#), [FlatFileTraceListener](#), [XmlTraceListener](#))
- Windows Event Log ([FormattedEventLogTraceListener](#))
- MSMQ ([MsmqEventTraceListener](#))
- Email ([EmailTraceListener](#))
- Database ([FormattedDatabaseTraceListener](#))
- WMI ([WmiTraceListener](#))

Now, our requirement is to send the output for each tenant to its own file. For example, output for tenant “**abc.com**” will go to “**abc.com.log**”, and so on. But first things first—here’s our basic contract for logging:

Code Sample 98

```

public interface ILog
{
    void Write(Object message, Int32 priority, TraceEventType severity,
               Int32 eventId = 0);
}

```

I kept it simple, but, by all means, do add any auxiliary methods you find appropriate.

The ELLAB can log:

- An arbitrary **message**
- A string **category**: we will use it for the tenant name, so we will not expose it in the API
- An integer **priority**
- The event **severity**
- An event identifier, which is optional (**eventId**)

The implementation of it on top of the ELLAB could be:

Code Sample 99

```
public sealed class EntLibLog : ILog, ITenantAwareService
{
    public static readonly ILog Instance = new EntLibLog();

    private EntLibLog()
    {
        //private constructor since this is meant to be used as a singleton
    }

    public void Write(Object message, Int32 priority, TraceEventType severity,
        Int32 eventId = 0)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        Logger.Write(message, tenant.Name, priority, eventId, severity);
    }
}
```

The registration follows the same pattern as before:

Code Sample 100

```
container.RegisterInstance<ILog>(EntLibLog.Instance);
```

This time, we also need to account for the ELLAB configuration. Because we need to setup each tenant individually, we have to run the following code upon startup:

Code Sample 101

```
private void CreateLogFactories(IEnumerable<ITenantConfiguration> tenants)
{
    foreach (var tenant in tenants)
    {
        try
        {

```

```

        var configurationSource =
            new FileConfigurationSource(tenant.Name +
                ".config");
        var logWriterFactory = new LogWriterFactory(
            configurationSource);
        Logger.SetLogWriter(logWriterFactory.Create());
    }
    catch {}
}

var tenants = TenantsConfiguration.GetTenants();
CreateLogFactories(tenants);

```



Tip: The *try...catch* block surrounding the initialization code is there so that if we do not want to have a configuration file for a specific tenant, it doesn't throw.



Note: There are lots of other options for providing this kind of information, such as with attributes.

This code, specifically the `FileConfigurationSource` class, requires that all tenants have their configuration in an individual file, named after the tenant (`<tenant>.config`). Next is an example of a logging configuration that uses a rolling flat file that rotates every week:

Code Sample 102

```

<configuration>
  <configSections>
    <section name="loggingConfiguration"
      type="Microsoft.Practices.EnterpriseLibrary.Logging.
Configuration.LoggingSettings, Microsoft.Practices.EnterpriseLibrary.Logging"
    />
  </configSections>
  <loggingConfiguration name="loggingConfiguration" tracingEnabled="true"
    defaultCategory="" logWarningsWhenNoCategoriesMatch="true">
    <listeners>
      <add name="Rolling Flat File Trace Listener"
        type="Microsoft.Practices.EnterpriseLibrary
.Logging.TraceListeners.RollingFlatFileTraceListener,
Microsoft.Practices.EnterpriseLibrary.Logging"
        listenerDataType="Microsoft.Practices.EnterpriseLibrary
.Logging.Configuration.RollingFlatFileTraceListenerData,
Microsoft.Practices.EnterpriseLibrary.Logging"
        fileName="abc.com.log"
        footer="-----"
      />
    </listeners>
  </loggingConfiguration>
</configuration>

```

```

        formatter="Text Formatter"
        header="-----"
        rollFileExistsBehavior="Increment"
        rollInterval="Week"
        timeStampPattern="yyyy-MM-dd hh:mm:ss"
        traceOutputOptions="LogicalOperationStack, DateTime,
Timestamp, ProcessId, ThreadId, Callstack"
        filter="All" />
</listeners>
<formatters>
    <add type="Microsoft.Practices.EnterpriseLibrary.Logging.
Formatters.TextFormatter,
Microsoft.Practices.EnterpriseLibrary.Logging"
        template="Timestamp: {timestamp}&#xA;
Message: {message}&#xA;Category: {category}&#xA;
Priority: {priority}&#xA;EventId: {eventid}&#xA;
Severity: {severity}&#xA;Title:{title}&#xA;
Machine: {machine}&#xA;Process Id: {processId}&#xA;
Process Name: {processName}&#xA;"
        name="Text Formatter" />
</formatters>
<categorySources>
    <add switchValue="All" name="General">
        <listeners>
            <add name=
                "Rolling Flat File Trace Listener" />
        </listeners>
    </add>
</categorySources>
<specialSources>
    <allEvents switchValue="All" name="All Events">
        <listeners>
            <add name=
                "Rolling Flat File Trace Listener" />
        </listeners>
    </allEvents>
    <notProcessed switchValue="All"
        name="Unprocessed Category">
        <listeners>
            <add name=
                "Rolling Flat File Trace Listener" />
        </listeners>
    </notProcessed>
    <errors switchValue="All"
        name="Logging Errors & Warnings">
        <listeners>
            <add name=
                "Rolling Flat File Trace Listener" />
        </listeners>
    </errors>

```

```
</specialSources>
</loggingConfiguration>
</configuration>
```



Note: For more information on the Enterprise Library Logging Application Block, check out the [Enterprise Library site](#). For logging to a database, you need to install the [Logging Application Block Database Provider](#) NuGet package.

Monitoring

The classic APIs offered by ASP.NET and .NET applications for diagnostics and monitoring do not play well with multitenancy, namely:

- [Tracing](#)
- [Performance counters](#)
- Web events ([Health Monitoring Provider](#))

In this chapter, we will look at some techniques for making these APIs multitenant. Before that, let's define a unifying contract for all these different APIs:

Code Sample 103

```
public interface IDiagnostics
{
    Int64 IncrementCounterInstance(String instanceName, Int32 value = 1);
    Guid RaiseWebEvent(Int32 eventCode, String message, Object data,
        Int32 eventDetailCode = 0);
    void Trace(Object value, String category);
}
```

An explanation of each of these methods is required:

- **Trace:** writes to the registered trace listeners
- **IncrementCounterInstance:** increments a performance counter specific to the current tenant
- **RaiseWebEvent:** raises a Web Event in the ASP.NET Health Monitoring infrastructure

In the following sections, we'll see each in more detail.

Tracing

ASP.NET Tracing

The [ASP.NET tracing](#) service is very useful for profiling your ASP.NET Web Forms pages, and it can even help in sorting out some problems.

Before you can use tracing, it needs to be globally enabled, which is done on the **Web.config** file, through a [trace](#) element:

Code Sample 104

```
<system.web>

    <trace enabled="true" localOnly="true" writeToDiagnosticsTrace="true"
        pageOutput="true" traceMode="SortByTime" requestLimit="20"/>

</system.web>
```

What this declaration says is:

- Tracing is **enabled**.
- Trace output is only present to local users (**localOnly**).
- Standard diagnostics tracing listeners are notified (**writeToDiagnosticsTrace**).
- Output is sent to the bottom of each page instead of being displayed on the trace handler URL (**pageOutput**).
- Trace events are sorted by their timestamp (**traceMode**).
- Up to 20 requests are stored (**requestLimit**).

You also need to have it enabled on a page-by-page basis (the default is to be disabled):

Code Sample 105

```
<%@ Page Language="C#" CodeBehind="Default.aspx.cs" Inherits="WebForms.Default"
    Trace="true" %>
```

We won't go into details on ASP.NET tracing. Instead, let's see a sample trace, for ASP.NET Web Forms:

Request Details			
Session Id:	yrtoloyropobwamhag15zev	Request Type:	GET
Time of Request:	04/06/2015 13:02:44	Status Code:	200
Request Encoding:	Unicode (UTF-8)	Response Encoding:	Unicode (UTF-8)
Trace Information			
Category	Message	From First(s)	From Last(s)
aspx.page	abc.com: Begin PreInit		
aspx.page	abc.com: End PreInit	0,001671	0,001671
aspx.page	abc.com: Begin Init	0,002118	0,000447
aspx.page	abc.com: End Init	0,002519	0,000401
aspx.page	abc.com: Begin InitComplete	0,002805	0,000366
aspx.page	abc.com: End InitComplete	0,003252	0,000367
aspx.page	abc.com: Begin PreLoad	0,003603	0,000351
aspx.page	abc.com: End PreLoad	0,003957	0,000354
aspx.page	abc.com: Begin Load	0,004308	0,000351
aspx.page	abc.com: End Load	0,028871	0,024564
aspx.page	abc.com: Begin LoadComplete	0,029529	0,000658
aspx.page	abc.com: End LoadComplete	0,029895	0,000366
aspx.page	abc.com: Begin PreRender	0,030250	0,000355
aspx.page	abc.com: End PreRender	0,030618	0,000368
aspx.page	abc.com: Begin PreRenderComplete	0,030998	0,000379
aspx.page	abc.com: End PreRenderComplete	0,031341	0,000343
aspx.page	abc.com: Begin SaveState	0,032079	0,000738
aspx.page	abc.com: End SaveState	0,032714	0,000635
aspx.page	abc.com: Begin SaveStateComplete	0,033090	0,000376
aspx.page	abc.com: End SaveStateComplete	0,062371	0,029281
aspx.page	abc.com: Begin Render	0,063178	0,000807
aspx.page	abc.com: End Render	0,065420	0,002242
Control Tree			
Control UniqueID	Type	Render Size Bytes (including children)	ViewState Size Bytes (excluding children)
Page	ASP.default_aspx	2481	0
ct000	ASP.abc_com_master	2481	0
ct000sc005	System.Web.UI.LiteralControl	27	0
ct000sc010	System.Web.UI.HtmlControls.HtmlHead	1027	0
ct000sc012	System.Web.UI.HtmlControls.HtmlTitle	19	0
ct000sc013	System.Web.UI.WebControls.ContentPlaceholder	0	0
ct000sc013	WebForms.GoogleAnalytics	435	0
ct000sc014	System.Web.UI.WebControls.ContentPlaceholder	483	0
ct000sc014	System.Web.UI.LiteralControl	483	0
ct000sc014	System.Web.UI.HtmlControls.HtmlLink	77	0
ct000sc015	System.Web.UI.LiteralControl	11	0
aspnetForm	System.Web.UI.HtmlControls.HtmlForm	0	0
ct000sc017	System.Web.UI.LiteralControl	24	0
ct000sc018	System.Web.UI.WebControls.ContentPlaceholder	0	0
ct000sc018	System.Web.UI.LiteralControl	14	0
ct000sc019	System.Web.UI.WebControls.ContentPlaceholder	641	0
ct000sc019	System.Web.UI.LiteralControl	30	0
ct000sc019	System.Web.UI.WebControls.Label	25	0
ct000sc019	System.Web.UI.LiteralControl	5	0
ct000sc019	System.Web.UI.WebControls.TextBox	85	0
ct000sc019	System.Web.UI.LiteralControl	5	0

Figure 20: Page trace for Web Forms

In MVC, the only difference is that the **Control Tree** table is empty, which is obvious, once you think of it.

This trace is being displayed on the page itself, as dictated by `pageOutput`; the other option is to have traces only showing on the trace handler page, `Trace.axd`, and leave the page alone. Either way, the output is the same.

A tracing entry corresponds to a request handled by the server. We can add our own trace messages to the entry by calling one of the trace methods in the [TraceContext](#) class, conveniently available as [Page.Trace](#) in Web Forms:

Code Sample 106

```
protected override void OnLoad(EventArgs e)
{
    this.Trace.Write("On Page.Load");
    base.OnLoad(e);
}
```

Or as static methods of the [Trace](#) class (for MVC or in general), which even has an overload for passing a condition:

Code Sample 107

```
public ActionResult Index()
{
    Trace.WriteLine("Before presenting a view");
}
```

```

var tenant = TenantsConfiguration.GetCurrentTenant();
Trace.WriteLineIf(tenant.Name != "abc.com", "Not abc.com");
return this.View();
}

```

Now, the tracing provider keeps a number of traces, up to the value specified by **requestLimit**—the default being 10, and the maximum, 10,000. This means that requests for all our tenants will be handled the same way, so if we go to the **Trace.axd** URL, we have no way of knowing which tenant for the request was for. But if you look closely to the **Message** column of the **Trace Information** table in Figure 20, you will notice a prefix that corresponds to the tenant for which the request was issued. In order to have that, we need to register a custom diagnostics listener on the **Web.config** file, in a [system.diagnostics](#) section:

Code Sample 108

```

<system.diagnostics>
  <trace autoflush="true">
    <listeners>
      <add name="MultitenantTrace"
           type="WebForms.MultitenantTraceListener,
WebForms" />
    </listeners>
  </trace>
</system.diagnostics>

```

The code for **MultitenantTraceListener** is presented below:

Code Sample 109

```

public sealed class MultitenantTraceListener : WebPageTraceListener
{
    private static readonly MethodInfo GetDataMethod = typeof(TraceContext)
        .GetMethod("GetData", BindingFlags.NonPublic | BindingFlags.Instance);

    public override void WriteLine(String message, String category)
    {
        var ds = GetDataMethod.Invoke(HttpContext.Current.Trace, null)
            as DataSet;
        var dt = ds.Tables["Trace_Trace_Information"];
        var dr = dt.Rows[dt.Rows.Count - 1];
        var tenant = TenantsConfiguration.GetCurrentTenant();
        dr["Trace_Message"] = String.Concat(tenant.Name, ": ",
            dr["Trace_Message"]);

        base.WriteLine(message, category);
    }
}

```


What this does is, with a bit of reflection magic, gets a reference to the current dataset containing the last traces, and, on the last of them—the one for the current request—adds a prefix that is the current tenant's name (for example, **abc.com**).

Web.config is not the only way by which a diagnostics listener can be registered; there's also code: [Trace.Listeners](#). Using this mechanism, you can add custom listeners that will do all kinds of things when a trace call is issued:

Code Sample 110

```
protected void Application_Start()
{
    //unconditionally adding a listener
    Trace.Listeners.Add(new CustomTraceListener());
}

protected void Application_BeginRequest()
{
    //conditionally adding a listener
    var tenant = TenantsConfiguration.GetCurrentTenant();
    if (tenant.Name == "abc.com")
    {
        Trace.Listeners.Add(new AbcComTraceListener());
    }
}
```

Tracing Providers

Other tracing providers exist in the .NET base class library:

- [EventLogTraceListener](#): writes to the Windows Event Log
- [WebPageTraceListener](#): ASP.NET tracing
- [IisTraceListener](#): IIS trace
- [EventProviderTraceListener](#): [Event Tracing for Windows](#) (ETW)
- [ConsoleTraceListener](#): console
- [DefaultTraceListener](#): output window in Visual Studio
- [TextWriterTraceListener](#): text file
- [DelimitedListTraceListener](#): text file with custom field delimiter
- [EventSchemaTraceListener](#): XML text file
- [XmlWriterTraceListener](#): XML text file
- [FileLogTraceListener](#): text file

All of these can be registered in [system.diagnostics](#) or [Trace.Listeners](#). We will have a wrapper class for the [Trace](#) static methods, in which we implement the **IDiagnostics** interface's **Trace** method:

Code Sample 111

```
public sealed class MultitenantDiagnostics : IDiagnostics, ITenantAwareService
{
```

```

public void Trace(Object value, String category)
{
    var tenant = TenantsConfiguration.GetCurrentTenant();
    System.Diagnostics.Trace.AutoFlush = true;
    System.Diagnostics.Trace.WriteLine(String.Concat(tenant.Name,
        ": ", value), category);
}
}

```

Performance Counters

[Performance Counters](#) are a Windows feature that can be used to provide insights on running applications and services. It is even possible to have Windows react automatically to the situation where a performance counter's value exceeds a given threshold. If we so desire, we can use performance counters to communicate to interested parties aspects of the state of our applications, where this state consists of integer values.

Performance Counters are organized in:

- Categories: a name
- Counters: a name and a type (let's forget about the type for now)
- Instances: a name and a value of a given type (we'll assume a long integer)

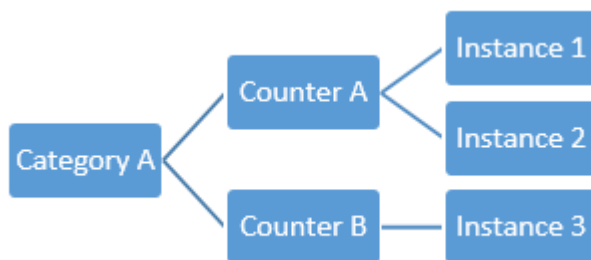


Figure 21: Performance counters basic concepts

In the **ITenantConfiguration** interface, we added a **Counters** property, which, when implemented, is used to automatically create performance counter instances. We will follow this approach:

Table 6: Mapping performance counter concepts for multitenancy

Concept	Content
Category	"Tenants"
Counter	The tenant name (eg, abc.com or xyz.net)
Instance	A tenant-specific name, coming from ITenantConfiguration.Counters

The code for automatically creating each performance counter and instance in the **TenantsConfiguration** class is as follows:

Code Sample 112

```
public sealed class TenantsConfiguration : IDiagnostics, ITenantAwareService
{
    private static void CreatePerformanceCounters(
        IEnumerable<ITenantConfiguration> tenants)
    {
        if (PerformanceCounterCategory.Exists("Tenants"))
        {
            PerformanceCounterCategory.Delete("Tenants");
        }

        var counterCreationDataCollection =
            new CounterCreationDataCollection(
                tenants.Select(tenant =>
                    new CounterCreationData(
                        tenant.Name,
                        String.Empty,
                        PerformanceCounterType.NumberOfItems32))
                    .ToArray());

        var category = PerformanceCounterCategory.Create("Tenants",
            "Tenants performance counters",
            PerformanceCounterCategoryType.MultiInstance,
            counterCreationDataCollection);

        foreach (var tenant in tenants)
        {
            foreach (var instance in tenant.Counters)
            {
                var counter = new PerformanceCounter(
                    category.CategoryName,
                    tenant.Name,
                    String.Concat(tenant.Name,
```

```

        ":" ,
        instance.InstanceName), false);
    }
}
}
}

```

On the other side, the code for incrementing a performance counter instance is defined in the **IDiagnostics** interface (**IncrementCounterInstance**), and can we can implement it as:

Code Sample 113

```

public sealed class MultitenantDiagnostics : IDiagnostics, ITenantAwareService
{
    public Int64 IncrementCounterInstance(String instanceName,
        Int32 value = 1)
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        using (var pc = new PerformanceCounter("Tenants", tenant.Name,
            String.Concat(tenant.Name, ":", instanceName), false))
        {
            pc.RawValue += value;
            return pc.RawValue;
        }
    }
}

```

When the application is running, we can observe in real-time the values of counter instances through the [Performance Monitor](#) application:

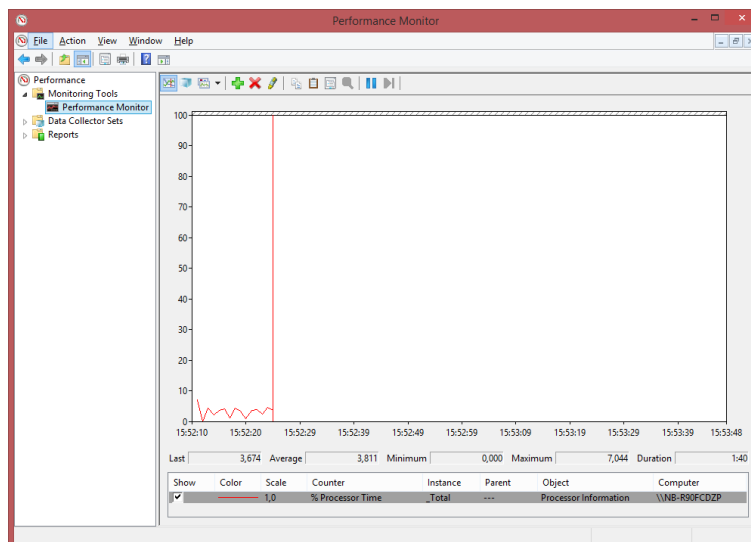


Figure 22: Performance Monitor application displaying performance counters

We just need to add some counters to the display; ours will be available under **Tenants - <tenant name> - <instance name>**:

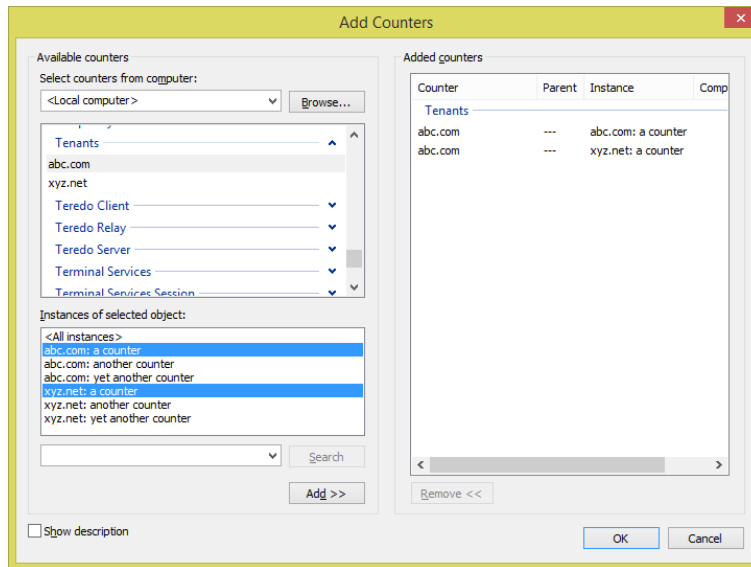


Figure 23: Adding performance counters

In this example, it is perceivable that both tenants, **abc.com** and **xyz.net**, have identically named counter instances, but it doesn't have to be the case.

There are other built-in ASP.NET-related performance counters that can be used to monitor your applications. For example, in **Performance Monitor**, add a new counter and select **ASP.NET Applications**:

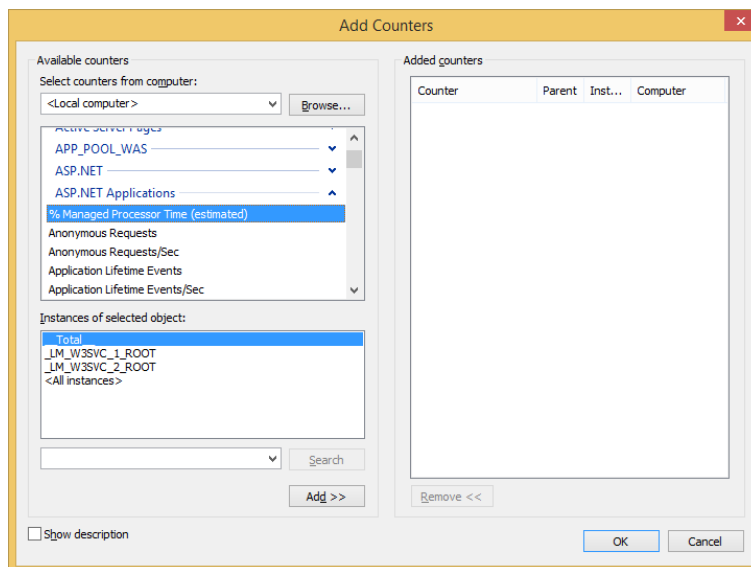


Figure 24: ASP.NET Applications performance counters

You will notice that you have several instances, one for each site that is running. These instances are automatically named, but each number can be traced to an application. For instance, if you are using IIS Express, open the **ApplicationHost.config** file (**C:\Users\<username>\Documents\IISExpress\Config**) and go to the **sites** section, where you will find something like this (in case, of course, you are serving tenants **abc.com** and **xyz.net**):

Code Sample 114

```
<sites>
  <site name="abc.com" id="1">
    <application path="/" applicationPool="Clr4IntegratedAppPool">
      <virtualDirectory path="/"
        physicalPath="C:\InetPub\Multitenant" />
    </application>
    <bindings>
      <binding protocol="http" bindingInformation="*:80:abc.com" />
    </bindings>
  </site>
  <site name="xyz.net" id="2">
    <application path="/" applicationPool="Clr4IntegratedAppPool">
      <virtualDirectory path="/"
        physicalPath="C:\InetPub\Multitenant" />
    </application>
    <bindings>
      <binding protocol="http" bindingInformation="*:80:xyz.net" />
    </bindings>
  </site>
</sites>
```

Or using IIS:

Add Website

Site name: Application pool:

Content Directory

Physical path:

Pass-through authentication

Binding

Type: IP address: Port:

Host name:

Example: www.contoso.com or marketing.contoso.com

☒ Start Website immediately

Figure 25: Creating a separate site for abc.com

Add Website

Site name: Application pool:

Content Directory

Physical path:

Pass-through authentication

Binding

Type: IP address: Port:

Host name:

Example: www.contoso.com or marketing.contoso.com

☒ Start Website immediately

Figure 26: Creating a separate site for xyz.net

In order to have more control, we separated the two sites; this is required for more accurately controlling each. The code base remains the same, though.

In the name of each instance (`_LM_W3SVC_<n>_ROOT`), `<n>` will match one of these numbers.

If, on the other hand, you want to use the full IIS, the [appcmd](#) command will give you this information:

Code Sample 115

```
C:\Windows\System32\inetsrv>appcmd list site

SITE "abc.com" (id:1,bindings:http/abc.com:80:,state:Started)
SITE "xyz.net" (id:2,bindings:http/xyz.net:80:,state:Started)


C:\Windows\System32\inetsrv>appcmd list apppool

APPPool "DefaultAppPool" (MgdVersion:v4.0,MgdMode:Integrated,state:Started)
APPPool "Classic .NET AppPool" (MgdVersion:v2.0,MgdMode:Classic,state:Started)
APPPool ".NET v2.0 Classic" (MgdVersion:v2.0,MgdMode:Classic,state:Started)
APPPool ".NET v2.0" (MgdVersion:v2.0,MgdMode:Integrated,state:Started)
APPPool ".NET v4.5 Classic" (MgdVersion:v4.0,MgdMode:Classic,state:Started)
APPPool ".NET v4.5" (MgdVersion:v4.0,MgdMode:Integrated,state:Started)
APPPool "abc.com" (MgdVersion:v4.0,MgdMode:Integrated,state:Started)
APPPool "xyz.net" (MgdVersion:v4.0,MgdMode:Integrated,state:Started)
```

I am listing all sites (first list) and then all application pools.

Health Monitoring

The [Health Monitoring](#) feature of ASP.NET enables the configuration of rules to respond to certain events that take place in an ASP.NET application. It uses a provided model to decide what to do when the rule conditions are met. An example might be sending out a notification mail when the number of failed login attempts reaches three in a minute's time. So, very simply, the Health Monitoring feature allows us to:

- Register a number of providers that will perform actions
- Add named rules bound to specific providers
- Map event codes to the created rules

There are a number of out-of-the-box events that are raised by the ASP.NET APIs internally, but we can define our own as well:

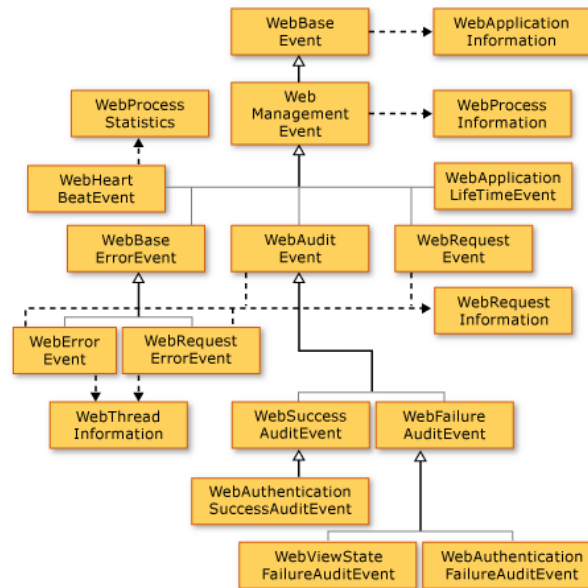


Figure 27: Included Health Monitoring events

Events are grouped in classes and subclasses. There are specific classes for dealing with authentication events ([WebAuditEvent](#), [WebFailureAuditEvent](#), [WebSuccessAuditEvent](#)), request ([WebRequestEvent](#), [WebRequestErrorEvent](#)) and application lifetime events ([WebApplicationLifeTimeEvent](#)), view state failure events ([WebViewStateFailureEvent](#)), and others. Each of these events (and classes) is assigned a unique numeric identifier, which is listed in [WebEventCodes](#) fields. Custom events should start with the value in [WebEventCodes.WebExtendedBase](#) + 1.

A number of providers—for executing actions when rules are met—are also included, of course. We can implement our own too, by inheriting from [WebEventProvider](#) or one of its subclasses:

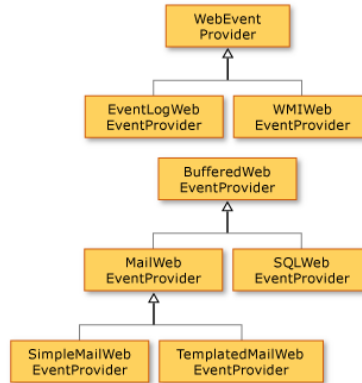


Figure 28: Included Health Monitoring providers

Included providers cover a number of typical scenarios:

- Writing to a database ([SQLWebEventProvider](#))
- Writing to WMI ([WMIWebEventProvider](#))
- Writing to the Event Log ([EventLogWebEventProvider](#))
- Sending mail ([SimpleMailWebEventProvider](#))

Before we go into writing some rules, let's look at our custom provider, **MultitenantEventProvider**, and its related classes **MultitenantWebEvent** and **MultitenantEventArgs**:

Code Sample 116

```

public sealed class MultitenantEventProvider : WebEventProvider
{
    private static readonly IDictionary<String, MultiTenantEventProvider>
        providers =
            new ConcurrentDictionary<String, MultiTenantEventProvider>();

    private const String TenantKey = "tenant";

    public String Tenant { get; private set; }

    public event EventHandler<MultiTenantEventArgs> Event;

    public static void RegisterEvent(String tenantId,
        EventHandler<MultiTenantEventArgs> handler)
    {
        var provider = FindProvider(tenantId);
        if (provider != null)
        {
            provider.Event += handler;
        }
    }

    public static MultiTenantEventProvider FindProvider(String tenantId)

```

```

    {
        var provider = null as MultiTenantEventProvider;
        providers.TryGetValue(tenantId, out provider);

        return provider;
    }

    public override void Initialize(String name, NameValueCollection config
)
    {
        var tenant = config.Get(TenantKey);

        if (String.IsNullOrEmpty(tenant))
        {
            throw new InvalidOperationException(
                "Missing tenant name.");
        }

        config.Remove(TenantKey);

        this.Tenant = tenant;
        providers[tenant] = this;

        base.Initialize(name, config);
    }

    public override void Flush()
    {
    }

    public override void ProcessEvent(WebBaseEvent raisedEvent)
    {
        var evt = raisedEvent as MultitenantWebEvent;
        if (evt != null)
        {
            var tenant = TenantsConfiguration.GetCurrentTenant();

            if (tenant.Name == evt.Tenant)
            {
                var handler = this.Event;
                if (handler != null)
                {
                    handler(this,
                                new MultitenantEventArgs(
                                    this, evt));
                }
            }
        }
    }
}

```

```

        public override void Shutdown()
        {
        }
    }

    [Serializable]
    public sealed class MultitenantEventArgs : EventArgs
    {
        public MultitenantEventArgs(MultitenantEventProvider provider,
            MultitenantWebEvent evt)
        {
            this.Provider = provider;
            this.Event = evt;
        }

        public MultitenantEventProvider Provider { get; private set; }
        public MultitenantWebEvent Event { get; private set; }
    }

    public class MultitenantWebEvent : WebBaseEvent
    {
        public MultitenantWebEvent(String message, Object eventSource,
            Int32 eventCode, Object data) :
            this(message, eventSource, eventCode, data, 0) {}

        public MultitenantWebEvent(String message, Object eventSource,
            Int32 eventCode, Object data, Int32 eventDetailCode) :
            base(message, eventSource, eventCode, eventDetailCode)
        {
            var tenant = TenantsConfiguration.GetCurrentTenant();
            this.Tenant = tenant.Name;
            this.Data = data;
        }

        public String Tenant { get; private set; }
        public Object Data { get; private set; }
    }
}

```

So, we have a provider class (**MultitenantEventProvider**), a provider event argument (**MultitenantEventArgs**), and a provider event (**MultitenantWebEvent**). We can always find the provider that was registered for the current tenant by calling the static method **FindProvider**, and from this provider we can register event handlers to the **Event** event. In a moment, we'll see how we can wire this, but first, here is a possible implementation of the **IDiagnostics** interface **RaiseWebEvent** method:

Code Sample 117

```

public sealed class MultitenantDiagnostics : IDiagnostics, ITenantAwareService
{

```

```

public Guid RaiseWebEvent(Int32 eventCode, String message, Object data,
    Int32 eventDetailCode = 0)
{
    var tenant = TenantsConfiguration.GetCurrentTenant();
    var evt = new MultitenantWebEvent(message, tenant, eventCode,
        data, eventDetailCode);
    evt.Raise();
    return evt.EventID;
}
}

```

Now, let's add some rules and see things moving. First, we need to add a couple of elements to the **Web.config** file's [healthMonitoring](#) section:

Code Sample 118

```

<configuration>
  <system.web>
    <healthMonitoring enabled="true" heartbeatInterval="0">
      <providers>
        <add name="abc.com"
            type="MultitenantEventProvider,
MyAssembly"
            tenant="abc.com" />
        <add name="xyz.net"
            type="MultiTenantEventProvider, MyAssembly"
            tenant="xyz.net" />
      </providers>
      <rules>
        <add name="abc.com Custom Event"
            eventName="abc.com Custom Event"
            provider="abc.com"
            minInterval="00:01:00"
            minInstances="1" maxLimit="1" />
        <add name="xyz.net Custom Event"
            eventName="xyz.net Custom Event"
            provider="xyz.net"
            minInterval="00:01:00"
            minInstances="2" maxLimit="2" />
      </rules>
      <eventMappings>
        <add name="abc.com Custom Event"
            startEventCode="100001"
            endEventCode="100001"
            type="MultiTenantWebEvent, MyAssembly" />
        <add name="xyz.net Custom Event"
            startEventCode="200001"
            endEventCode="200001"
            type="MultiTenantWebEvent, MyAssembly" />
      </eventMappings>
    </healthMonitoring>
  </system.web>
</configuration>

```

```
        </eventMappings>
    </healthMonitoring>
</system.web>
</configuration>
```

So, what we have here is:

- Two **providers** registered of the same class (**MultitenantEventProvider**), one for each tenant, with an attribute **tenant** that states it
- Two **rules**, each which raises a custom event when a named event (**eventName**) is raised a number of times (**minInstances**, **maxLimit**) in a certain period of time (**minInterval**), for a given **provider**;
- Two event mappings (**eventMappings**) that translate event id intervals (**startEventCode**, **endEventCode**) to a certain event class (**type**).

We'll also need to add an event handler for the **MultitenantEventProvider** class of the right tenant, maybe in [Application_Start](#):

Code Sample 119

```
protected void Application_Start()
{
    MultiTenantEventProvider.RegisterEvent("abc.com", (s, e) =>
    {
        //do something when the event is raised
    });
}
```

So, in the end, if a number of web events is raised in the period specified in any of the rules, an event is raised and hopefully something happens.

Analytics

Hands down, [Google Analytics](#) is the *de facto* standard when it comes to analyzing the traffic on your web site. No other service that I am aware of offers the same amount of information and features, so it makes sense to use it, also for multitenant sites.

If you don't have a Google Analytics account, go create one, the process is pretty straightforward (yes, you do need a Google account for that).

After you have one, go to the **Admin** page and create as many properties as your tenants (**Property** drop down list):

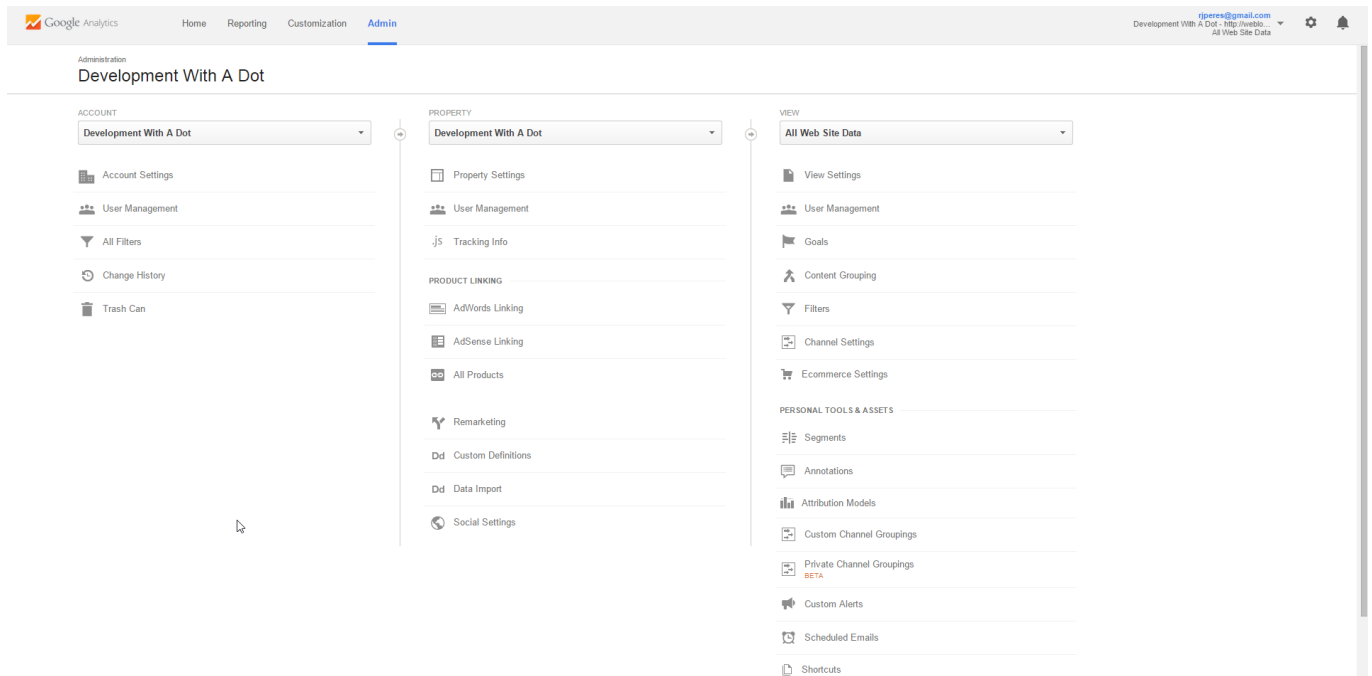


Figure 29: Creating Google Analytics properties

Each tenant will be assigned a unique key in the form **UA-*nnnnnnnnn-n***, where *n* are numbers. If you followed the previous topic on the Configuration service, you will have a configuration file per tenant (**abc.com.config**) at the root of your web site, and this is where you will store this key:

Code Sample 120

```
<configuration>
  <appSettings>
    <add key="GoogleAnalyticsKey" value="UA-nnnnnnnnn-n" />
  </appSettings>
</configuration>
```

Of course, do replace **UA-*nnnnnnnnn-n*** with the right key!

Now, we need to add some JavaScript to the page where we mention this key and tenant name. This script is where the actual call to the Google Analytics API is done, and it looks like this:

Code Sample 121

```
<script type="text/javascript"> // <![CDATA[
  (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
    (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
```

```

        m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
    })(window,document,'script','//www.google-
analytics.com/analytics.js','ga');
    ga('create', '{0}', 'auto');
    ga('send', 'pageview');
// ]]></script>

```

If you are curious about where this came from, Google Analytics' **Admin** page has this information ready for you to pick up—just click on **Tracking Info** for your property (tenant) of choice.

You might have noticed the **{0}** token—this is a standard .NET string formatting placeholder. What it means is, it needs to be replaced with the actual tenant key (**UA-*nnnnnnnnn-n***). Each tenant will have this key stored in its configuration, say, under the key **GoogleAnalyticsKey**. This way we can always retrieve it through the **IConfiguration** interface presented a while ago.

If we will be using ASP.NET Web Forms, a nice wrapper for this might be a control:

Code Sample 122

```

public sealed class GoogleAnalytics : Control
{
    private const String Script = "<script type=\"text/javascript\">// <![CDATA[" +
        "(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){\" +
        \" (i[r].q=i[r].q||[]).push(arguments)}};i[r].l=1*new Date();a=s.createElement(o,\" +
        \" m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)\" +
        \" })}(window,document,'script','//www.google-\" +
        analytics.com/analytics.js','ga');\" +
        \" ga('create', '{0}', 'auto');\" +
        \" ga('send', 'pageview');\" +
        \"// ]]></script>\";

    protected override void Render(HtmlTextWriter writer)
    {
        var config = ServiceLocator.Current
            .GetInstance<IConfiguration>();
        var key = config.GetValue("GoogleAnalyticsKey");

        writer.Write(Script, key);
    }
}

```


Here is a sample declaration on a page or master page:

Code Sample 123

```
<%@ Register Assembly="Multitenancy.WebForms" Namespace="Multitenancy.WebForms"
TagPrefix="mt" %>

<mt:GoogleAnalytics runat="server" />
```

Otherwise, for MVC, the right place would be an extension method over [HtmlHelper](#):

Code Sample 124

```
public static class HtmlHelperExtensions
{
    private const String Script = "<script type=\"text/javascript\">/// <![C
DATA[" +
    "(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||func
tion(){{" +
    " (i[r].q=i[r].q||[]).push(arguments)}};i[r].l=1*new Date();a=s.create
Element(o)," +
    " m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insert
Before(a,m)" +
    " }})(window,document,'script','//www.google-
analytics.com/analytics.js','ga');" +
    " ga('create', '{0}', 'auto');" +
    " ga('send', 'pageview');" +
    "/// ]]></script>";

    public static void GoogleAnalytics(this HtmlHelper html)
    {
        var config = ServiceLocator.Current
            .GetInstance<IConfiguration>();
        var key = config.GetValue("GoogleAnalyticsKey");
        html.Raw(String.Format(Script, key));
    }
}
```

It would then be invoked as this, on a local or shared view:

Code Sample 125

```
@Html.GoogleAnalytics()
```

And that's it! Each tenant will get its own property page on Google Analytics, and you can start monitoring them.

Chapter 10 Security

Introduction

All applications need security enforcing. Multitenant applications have additional requirements, in the sense that something that is allowed for some tenant maybe shouldn't be for other tenants. This chapter talks about some of the security mechanisms to consider when writing multitenant applications.

Authentication

The authentication mechanism that will be used should be capable of:

- Logging in users associated with the current tenant
- Denying login to users coming from a tenant with which they are not associated

Let's see how we can implement these requirements.

ASP.NET Membership and Role Providers

The venerable [ASP.NET Membership and Role providers](#) are part of the extensible [provider model introduced in ASP.NET 2.0](#). The Membership provider takes care of authenticating and managing users, and the Role provider associates these users to groups. This framework has aged over the years, but because it might be necessary to support it (brownfield development), I decided it was worth mentioning.

The framework includes two implementations of these providers: one for using Windows facilities for authentication and group membership, and one for using SQL Server-based objects.

You can imagine that the SQL Server versions are more flexible, in that they do not force us to use Windows (or Active Directory) accounts for our users and groups. The schema they use is this:

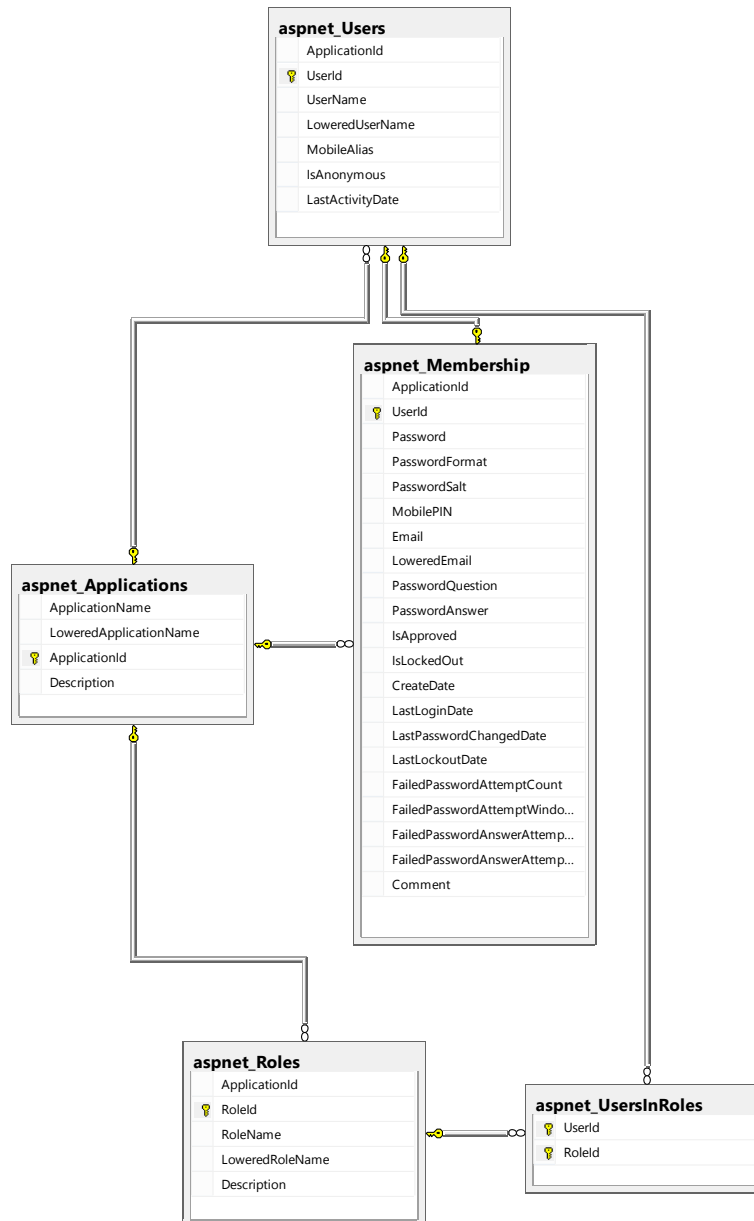


Figure 30: ASP.NET Membership and Role provider schema

Key to understanding this schema are these tables:

Table 7: ASP.NET Membership and Role provider tables and their purposes

Table	Purpose
aspnet_Applications	Stores all the applications that the Membership and Role providers know of
aspnet_Users	Stores all the registered users, associated with an application
aspnet_Roles	Stores all of the roles
aspnet_Membership	Additional information about registered users
aspnet_UsersInRoles	Association of users to roles

Both the users tables and the roles tables are dependant of the application table, meaning different applications can have different users and roles. A single database can hold several registered users and roles, maybe for different web sites, each acting as a separate application; in this case, each web site will be identified by a different application name. The configuration is normally defined statically in the global ASP.NET configuration file, **Machine.config**, the related part of which is shown here, a bit simplified:

Code Sample 126

```
<membership defaultProvider="AspNetSqlMembershipProvider">
  <providers>
    <add name="AspNetSqlMembershipProvider" applicationName="/"
        type="System.Web.Security.SqlMembershipProvider"
        connectionStringName="LocalSqlServer" />
  </providers>
</membership>
<roleManager enabled="true" defaultProvider="AspNetSqlRoleProvider">
  <providers>
    <add name="AspNetSqlRoleProvider" connectionStringName="LocalSqlServer"
        applicationName="/" type="System.Web.Security.SqlRoleProvider" />
    <add name="AspNetWindowsTokenRoleProvider" applicationName="/"
        type="System.Web.Security.WindowsTokenRoleProvider" />
  </providers>
</roleManager>
```

It's easy to see that the “application” concept of the Membership and Role providers matches closely that of “tenants.” The problem is that the application ([applicationName](#) attribute) is hardcoded in the **Web.config** file. However, because its associated property ([ApplicationName](#)) in the provider implementation class is virtual, we can have classes that inherit from the included ones ([SqlMembershipProvider](#) and [SqlRoleProvider](#)) and override this property so that it returns the proper value each time it is called:

Code Sample 127

```
//Membership provider
public class MultitenantSqlMembershipProvider : SqlMembershipProvider
{
    public override String ApplicationName
    {
        get { return TenantsConfiguration.GetCurrentTenant().Name; }
        set { /*nothing to do here*/ }
    }
}

//Role provider
public class MultitenantSqlRoleProvider : SqlRoleProvider
{
    public override String ApplicationName
    {
        get { return TenantsConfiguration.GetCurrentTenant().Name; }
        set { /*nothing to do here*/ }
    }
}
```



Note: There are provider implementations for other RDBMs, such as Oracle and MySQL. Because the implementation would be exactly the same, I only show the SQL Server version.

Now, we need to register our new implementations to either the **Machine.config** (for machine-wide availability) or **Web.config** files:

Code Sample 128

```
<membership defaultProvider="MultitenantSqlMembershipProvider">
  <providers>
    <add name="MultitenantSqlMembershipProvider"
        type="MyNamespace.MultitenantSqlMembershipProvider,
MyAssembly"
        connectionStringName="LocalSqlServer" />
  </providers>
</membership>
<roleManager enabled="true" defaultProvider="MultitenantSqlRoleProvider">
  <providers>
    <add name="MultitenantSqlRoleProvider"
        connectionStringName="LocalSqlServer"
        type="MyNamespace.MultitenantSqlRoleProvider,
MyAssembly" />
  </providers>
</roleManager>
```

```
        type="MyNamespace.MultitenantSqlRoleProvider, MyAssembly" />
    </providers>
</roleManager>
```



Tip: Even if I opted to talk about the Membership provider, by no means do I think that you should be using it in new (Greenfield) developments. The future according to Microsoft is [ASP.NET Identity](#) (at least, for the moment), and I strongly urge you to get into it. There's a good post [here](#) on how to migrate from the Membership provider to ASP.NET Identity.

ASP.NET Identity

A much better way to provide authentication to your ASP.NET applications is the [ASP.NET Identity](#) framework. Microsoft designed this framework to complement the shortcomings and problems of the old provider model, and to offer support for new authentication standards, such as [OAuth](#). Unfortunately, the Microsoft architects that designed it left out built-in support for multitenancy. All is not lost, however, because it has already been implemented for us by the community, in the person of [James Skimming](#) and the **AspNet.Identity.EntityFramework.Multitenant** package, among others. ASP.NET Identity is provider-agnostic, which means it can use, for example, Entity Framework or NHibernate as its persistence engine, but James' implementation relies on the Entity Framework.



*Note: There's a package called **NHibernate.AspNet.Identity**, by [Antonio Bastos](#), that provides an implementation of ASP.NET Identity with NHibernate. You can get the NuGet package [here](#), and the source code [here](#).*

So, after you add the ASP.NET Identity package using NuGet, you will need to add the **AspNet.Identity.EntityFramework.Multitenant** package:

```
PM> Install-Package
AspNet.Identity.EntityFramework.Multitenant
```

Figure 31: ASP.NET Identity EntityFramework with multitenancy support

After you do, we need to make a couple of changes to the ASP.NET Identity classes in our project. The class that interests us is **ApplicationUser**, which should be changed so as to inherit from **MultitenantIdentityUser** instead of **IdentityUser**:

Code Sample 129

```
public class ApplicationUser : MultitenantIdentityUser
{
    //rest goes here
}
```

The class **MultitenantIdentityUser** already provides a property called **TenantId**, which is used to distinguish between different tenants. This property is mapped to the **TenantId** column of the **AspNetUserLogins** table. The model looks like this:



Figure 32: ASP.NET Identity model

Other than that, we need to change our [DbContext](#)-derived class to make it inherit from **MultitenantIdentityDbContext<ApplicationUser>** as well:

Code Sample 130

```

public class ApplicationDbContext : MultitenantIdentityDbContext<ApplicationUser>
{
    //rest goes here
}
  
```


Finally, we need to teach ASP.NET Identity how to get the current tenant. In order to do that, we need to change the **Create** method of the **ApplicationUserManager** class like this:

Code Sample 131

```
public static ApplicationUserManager Create(
    IdentityFactoryOptions<ApplicationUserManager> options, IOwinContext context)
{
    var tenant = TenantsConfiguration.GetCurrentTenant();
    var manager = new ApplicationUserManager(
        new MultitenantUserStore<ApplicationUser>(
            context.Get<ApplicationDbContext>())
        { TenantId = tenant.Name });
    //rest goes here
}
```

The **ApplicationUserManager.Create** method is registered as the factory method for our user store. Whenever it is called, it will use the current tenant name. That's all there is to it—ASP.NET Identity can be used in exactly the same way as it would be without the multitenant support.



Note: James Skimming made available the source code for his *AspNet.Identity.EntityFramework.Multitenant* package in GitHub [here](#). The actual commit where these changes are outlined is [here](#).

HTTPS

If we are to follow the host header approach, and we would like to use HTTPS, we will need to supply a certificate for each of the named tenants. Now, we have two options:

- Acquire a certificate from a root authority
- Generate our own (dummy) certificate in IIS Manager, if we don't care about getting a message about an invalid root authority

For development purposes, we can go with a dummy certificate. We need to tell IIS to use it:

Add Website

Site name: MultiTenant Application pool: MultiTenant Select...

Content Directory

Physical path: c:\inetpub\MultiTenant ...

Pass-through authentication

Connect as... Test Settings...

Binding

Type: https IP address: All Unassigned Port: 443

Host name: abc.com

☐ Require Server Name Indication

SSL certificate: IIS Express Development Certificate Select... View...

☒ Start Website immediately

OK Cancel

Figure 33: Creating a new site

Add Site Binding

Type: https IP address: All Unassigned Port: 443

Host name: xyz.net

☐ Require Server Name Indication

SSL certificate: IIS Express Development Certificate Select... View...

OK Cancel

Figure 34: Adding a binding to a site



Note: *In the unlikely case that all our tenants will share at least part of the domain, we can use a wildcard certificate; otherwise, we will need one for each tenant, which might prove costly.*

Application pools

As mentioned earlier, having different application pools per tenant allows us to have the application running under different accounts. This can prove helpful, for example, if each user has different default databases on the server, and you don't specify the initial database on the connection string. Each tenant will land on its own database, according to the database server configuration. It is wiser, however, to not trust the default database assigned to a user, and to use a database partitioning or selection strategy, as will be outlined in the next chapter.

Cookies

Cookies are stored on the client side, in the browser's own store. By default, they are bound to the domain from which they originated, which is fine if we are to use the Host header strategy, but not so much if we want to use another tenant identification strategy. For example, if we don't do anything about it, a cookie sent in the context of a tenant will also be available if the user decides to browse another tenant's site. Cookies don't really offer many options, other than the path (relative part of the URL), the domain, and the availability for non-HTTPS sites. The only thing that we can play with are the names of the keys: we can add, for example, a prefix that identifies the tenant to which the cookie applies, something like:

Code Sample 132

```
var tenantCookies = HttpContext.Current.Request.Cookies.OfType<HttpCookie>()
    .Where(x => x.Name.StartsWith(String.Concat(tenant, ":")));

var cookie = new HttpCookie();
cookie.Name = String.Concat(tenant, ":", key);
cookie.Value = value;
HttpContext.Current.Response.Cookies.Add(cookie);
```

Chapter 11 Data Access

Introduction

Problem: each tenant needs to have, at least partially, separate data and even schema.

When it comes to accessing data in a multitenant application, there are three major techniques:

- **Separate database:** Each tenant's data is kept in a separate database instance, with a different connection string for each; the multitenant system should pick automatically the one appropriate for the current tenant



Figure 35: Separate databases

- **Separate schema:** The same database instance is used for all the tenants' data, but each tenant will have a separate schema; not all RDBMSes support this properly, for example, SQL Server doesn't, but Oracle does. When I say SQL Server doesn't support this, I don't mean to say that it doesn't have schemas, it's just that they do not offer an isolation mechanism as Oracle schemas do, and it isn't possible to specify, per query or per connection, the schema to use by default.

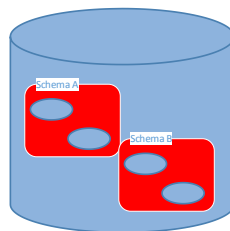


Figure 36: Separate schemas

- **Partitioned data:** The data for all tenants is kept in the same physical instance and schema, and a partitioning column is used to differentiate tenants; it is up to the framework to issue proper SQL queries that filter data appropriately.

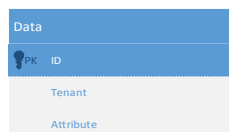


Figure 37: Partitioned data



Note: If you are interested in learning more, [this article](#) presents some of the differences between Oracle and SQL Server regarding schemas.

As for actually retrieving and updating data in relational databases, two main approaches exist:

- Using ADO.NET, or some thin wrapper around it, like the [Enterprise Library Data Access Block](#)
- Using an Object-Relational Mapper (ORM), such as [NHibernate](#) or [Entity Framework \(Code First\)](#)

A discussion as to which of these approaches is better is outside the scope of this book, and personally, it feels pointless to me: both have pros and cons. ORMs, however, offer some configuration mechanisms that allow data to be filtered automatically based on some condition, such as the tenant's name. Therefore, we will discuss how to use ORMs NHibernate and Entity Framework Code First for multitenant data access, and we will leave out SQL-based solutions.

NHibernate

Different Databases

In the NHibernate architecture, a connection string is tied to a **session factory**. It's the session factory that builds **sessions**, which in turn encapsulate ADO.NET **connections**. In general, it is a good practice to have a single session factory; in our case, we will need one per tenant (and connection string).

To make session factories discoverable, we will use the same mechanism we used earlier in this book, and that is the Common Service Locator. Each tenant will have its own registration of **ISessionFactory** under its name, and each session factory will point to a likewise-named connection string. Consider the following bootstrap code that, again, uses Unity as the Inversion of Control (IoC) framework:

Code Sample 133

```
protected void Application_BeginRequest()
{
    var tenant = TenantsConfiguration.GetCurrentTenant().Name;
    var sessionFactory = ServiceLocator.Current
        .TryResolve<ISessionFactory>(tenant);
    if (sessionFactory == null)
    {
        this.SetupSessionFactory(tenant);
    }
}

private void SetupSessionFactory(String tenant)
{
    var cfg = new Configuration().DataBaseIntegration(x =>
    {
```

```

        x.ConnectionStringName = tenant;
        //rest goes here
    });
    //etc
    //get the Unity instance from the Common Service Locator - another
    option would be to encapsulate this in some class or to use a custom
    connection provider
    var unity = ServiceLocator.Current.GetInstance<IUnityContainer>();
    unity.RegisterInstance<ISessionFactory>(tenant, cfg.BuildSessionFactory(
));
}

```

Here's what it does: when a request is received, it gets the tenant name from it, and checks to see if there is already a registered session factory under that name in the Common Service Locator. If not, it starts the process of building and registering the session factory for the current tenant.

This code can either go in the **Global.asax.cs** file, as an instance method of the [HttpApplication](#)-derived class, or in a module, a class that implements [IHttpModule](#). The latter is recommended to allow for code reuse and better maintainability, but if you are to follow this path, you will need to register the module yourself in the **Web.config** file, section [system.webServer/modules](#):

Code Sample 134

```

<system.webServer>
  <modules>
    <add name="MyModule" type="MyNamespace.MyModule, MyAssembly"/>
  </modules>
</system.webServer>

```

So, whenever you wish to open a session, you first need to retrieve the appropriate session factory for the current tenant:

Code Sample 135

```

var tenant = TenantsConfiguration.GetCurrentTenant().Name;
//lookup the session factory for the current tenant
var sessionFactory = ServiceLocator.Current.GetInstance<ISessionFactory>(tenant);
using (var session = sessionFactory.OpenSession())
{
    //...
}

```



Note: If you want to learn more about the NHibernate architecture, I suggest reading [NHibernate Succinctly](#), also in the Succinctly series.

Different Schemas

The problem with NHibernate and mappings is that normally we only have one session factory and a configuration instance from which it originated. Because it's the configuration instance that holds the mappings, which then passes to the session factory and in the end, to the sessions spawned from it, we need to have different configuration instances, one for each tenant.

Here's how we can configure the schema for the current tenant:

Code Sample 136

```
public class MyMultitenantEntityClassMapping : ClassMapping<MyMultiTenanEntity>
{
    public MyMultitenantEntityClassMapping()
    {
        var tenant = TenantsConfiguration.GetCurrentTenant().Name;
        this.Schema(tenant);
        //rest goes here
    }
}
```

Or, we can do it through conventions:

Code Sample 137

```
var mapper = new ConventionModelMapper();
var tenant = TenantsConfiguration.GetCurrentTenant().Name;
mapper.BeforeMapClass += (modelInspector, type, classCustomizer) =>
{
    classCustomizer.Schema(tenant);
};
```



Tip: Don't forget that the schema name cannot take all characters, normally—only alphanumeric characters are allowed—so you may need to do some transformation on the tenant name.

Data partitioning

NHibernate has a nice feature by the name of **filter** which can be used to define arbitrary SQL restrictions at the entity level. A filter can be enabled or disabled and can take runtime parameters, something that plays very nicely with tenant names. For example, say our table has a **tenant** column that keeps the name of the tenant that it refers to. We would add a SQL restriction of “**tenant = 'abc.com'**”, except that we can't hardcode the tenant name; we use parameters instead. A filter is defined in the entity's mapping. Here's an example using mapping by code:

Code Sample 138

```
public class MyMultitenantEntityClassMapping : ClassMapping<MyMultitenantEntity>
{
    public MyMultitenantEntityClassMapping()
    {
        this.Filter("tenant", filter =>
        {
            filter.Condition("tenant = :tenant");
        });
        //rest goes here
    }
}
```

Notice the “**tenant = :tenant**” part; this is a SQL restriction in disguise, where **tenant** is the name of a column and **:tenant** is a named parameter, which happens to have the same name. I omitted the most part of the mapping, because only the filtering part is relevant for our discussion. Similar code should be repeated in all the mappings of all the tenant-aware entities, and, of course, the proper column name should be specified.

Here's another example using the conventional mapper:

Code Sample 139

```
var mapper = new ConventionModelMapper();
var tenant = TenantsConfiguration.GetCurrentTenant().Name;
mapper.BeforeMapClass += (modelInspector, type, classCustomizer) =>
{
    classCustomizer.Filter("tenant", filter =>
    {
        filter.Condition("tenant = :tenant");
    });
};
```


Now, whenever we open a new session, we need to enable the **tenant** filter and assign a value to its **tenant** parameter:

Code Sample 140

```
var tenant = TenantsConfiguration.GetCurrentTenant().Name;
session
    .EnableFilter("tenant")
    .SetParameter("tenant", tenant);
```

The restriction and parameter value will last for the whole lifetime of the session, unless explicitly changed. You can see I am resorting to the static auxiliary method **GetCurrentTenant** that was defined earlier. Now, whenever you query for the **MyMultitenantEntity** class, the filter SQL will be appended to the generated SQL, with the parameter properly replaced by its actual value.

Generic Repository

To make life easier for the developer, we can encapsulate the creation of the sessions and the configuration of the filter behind a [Repository Pattern](#) (or **Generic Repository**) façade. This pattern dictates that the data access be hidden behind methods and collections that abstract the database operations and make them look just like in-memory.



Note: I won't go into a discussion of whether the Repository/Generic Repository Pattern is a good thing or not. I personally understand its drawbacks, but I consider it useful in certain scenarios, such as this one.

A possible definition for a Generic Repository interface is:

Code Sample 141

```
public interface IRepository : IDisposable
{
    T Find<T>(Object id);
    IQueryable<T> All<T>(params Expression<Func<T, Object>> [] expansions);
    void Delete(Object item);
    void Save(Object item);
    void Update(Object item);
    void Refresh(Object item);
    void SaveChanges();
}
```

Most methods should be familiar to readers. We will not cover this interface in depth; instead, let's move on to a possible implementation for NHibernate:

Code Sample 142

```
public sealed class SessionRepository : IRepository
{
    private ISession session;
```

```

public SessionRepository()
{
    var tenant = TenantsConfiguration.GetCurrentTenant().Name;
    //lookup the one and only session factory
    var sessionFactory = ServiceLocator.Current
        .GetInstance<ISessionFactory>();
    this.session = sessionFactory.OpenSession();
    //enable the filter with the current tenant
    this.session.EnableFilter("tenant")
        .SetParameter("tenant", tenant);
    this.session.BeginTransaction();
}

public T Find<T>(params Object[] ids) where T : class
{
    return this.session.Get<T>(ids.Single());
}

public IQueryable<T> Query<T>(params
    Expression<Func<T, Object>>[] expansions) where T : class
{
    var all = this.session.Query<T>() as IQueryable<T>;
    foreach (var expansion in expansions)
    {
        all = all.Include(expansion);
    }
    return all;
}

public void Delete<T>(T item) where T : class
{
    this.session.Delete(item);
}

public void Save<T>(T item) where T : class
{
    this.session.Save(item);
}

public void Update<T>(T item) where T : class
{
    this.session.Update(item);
}

public void Refresh<T>(T item) where T : class
{
    this.session.Refresh(item);
}

```

```

public void Detach<T>(T item) where T : class
{
    this.session.Evict(item);
}

public void SaveChanges()
{
    this.session.Flush();
    try
    {
        this.session.Transaction.Commit();
    }
    catch
    {
        this.session.Transaction.Rollback();
    }
    this.session.BeginTransaction();
}

public void Dispose()
{
    if (this.context != null)
    {
        this.session.Dispose();
        this.session = null;
    }
}
}

```

Remember that now there is only a single session factory, because there is also a single database. Now, all we have to do is register the **IRepository** interface with our IoC framework and always access it through the Common Service Locator:

Code Sample 143

```

//register our implementation under the IRepository interface
unity.RegisterType<IRepository, SessionRepository>(
    new PerRequestLifetimeManager());
//get a reference to a new instance
using (var repository = ServiceLocator.Current.GetInstance<IRepository>())
{
    //query some entity
    var items = repository.All<MyEntity>().ToList();
}

```



Tip: The process of enabling the filter and setting the tenant parameter must always be done, so either make sure you use a repository or perform the initialization yourself, perhaps in some infrastructure code if possible.



Note: I used for the registration the *PerRequestLifetimeManager* presented in Chapter 9 *Application Services* chapter.

Entity Framework Code First

Different databases

The architecture of Entity Framework Code First is quite different from that of NHibernate. For one, there is no builder method that we can hook up to that can return a tenant-specific context with its own connection string. What we can do is build our own factory method that returns a proper connection string, for the current tenant:

Code Sample 144

```
public class MultitenantContext : DbContext
{
    public MultitenantContext(): base(GetTenantConnection()) { }

    private static String GetTenantConnection()
    {
        var tenant = TenantsConfiguration.GetCurrentTenant();
        return String.Format("Name={0}", tenant.Name);
    }

    //rest goes here
}
```

It is important that you do not allow the creation of a context with any arbitrary connection string, because this defeats the purpose of transparent multitenancy through separate databases.

Different schemas

With Code First, it is very easy to apply our own conventions to a model:

Code Sample 145

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    var tenant = TenantsConfiguration.GetCurrentTenant();
    //repeat for all multitenant entities
    modelBuilder.Types().Configure(x => x.ToTable(x.ClrType.Name,
tenant.Name);
    //rest goes here
    base.OnModelCreating(modelBuilder);
}
```



Tip: Make sure the schema name is valid.

Data partitioning

Even if not quite as powerful, Entity Framework Code First allows us to map an entity with a discriminator column and value. This basically serves the purpose of allowing the [Table Per Class Hierarchy](#) / [Single Table Inheritance](#) strategy, where this column is used to tell to which entity a record maps to, in a table that is shared by a hierarchy of classes. The only out-of-the-box way to achieve this configuration is by overriding the [OnModelCreating](#) method and specifying a discriminator column and value:

Code Sample 146

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    var tenant = TenantsConfiguration.GetCurrentTenant();

    //repeat for all multitenant entities
    modelBuilder.Entity<MyMultitenantEntity>().Map(m => m.Requires("Tenant"
)
        .HasValue(tenant.Name));

    //rest goes here

    base.OnModelCreating(modelBuilder);
}
```

This tells Entity Framework that, whenever a [DbContext](#) is created, some entities should be mapped so that whenever they are inserted or retrieved, Entity Framework will always consider a **Tenant** discriminator column with a value identical to the current tenant, automatically.



Note: Again, for a more in-depth understanding of Entity Framework Code First, I suggest reading [Entity Framework Code First Succinctly](#), also in the *Succinctly* series.

Generic Repository

For a more secure solution, here's what an implementation of the Generic Repository Pattern for an Entity Framework Code First data context might be:

Code Sample 147

```
public sealed class MultitenantContextRepository : IRepository
{
    private MultitenantContext context;

    public MultitenantContextRepository()
    {
        //if you use the code from the previous example, this is not
        //necessary, it is done there
        var tenant = TenantsConfiguration.GetCurrentTenant();
        //set the connection string name from the current tenant
        this.context = new
            MultitenantContext(String.Format("Name={0}",
                tenant.Name));
    }

    public T Find<T>(params Object[] ids) where T : class
    {
        return this.context.Set<T>().Find(ids);
    }

    public IQueryable<T> Query<T>(params
        Expression<Func<T, Object>>[] expansions) where T : class
    {
        var all = this.context.Set<T>() as IQueryable<T>;
        foreach (var expansion in expansions)
        {
            all = all.Include(expansion);
        }
        return all;
    }

    public void Delete<T>(T item) where T : class
    {
        this.context.Set<T>().Remove(item);
    }

    public void Save<T>(T item) where T : class
    {
        this.context.Set<T>().Add(item);
    }

    public void Update<T>(T item) where T : class
    {

```

```

        this.context.Entry(item).State = EntityState.Modified;
    }

    public void Refresh<T>(T item) where T : class
    {
        this.context.Entry(item).Reload();
    }

    public void Detach<T>(T item) where T : class
    {
        this.context.Entry(item).State = EntityState.Detached;
    }

    public void SaveChanges()
    {
        this.context.SaveChanges();
    }

    public void Dispose()
    {
        if (this.context != null)
        {
            this.context.Dispose();
            this.context = null;
        }
    }
}

```

Pretty straightforward, I think. Just store it in Unity (or any IoC container of your choice) and you're done:

Code Sample 148

```

//register our implementation with Unity under the IRepository interface
unity.RegisterType<IRepository, MultitenantContextRepository>(
    new PerRequestLifetimeManager());
//get a reference to a new instance using Common Service Locator
using (var repository = ServiceLocator.Current.GetInstance<IRepository>())
{
    //query some entity
    var items = repository.All<MyEntity>().ToList();
}

```



Note: Notice again the *PerRequestLifetimeManager* lifetime manager.

Chapter 12 Putting It All Together

Class model

The framework classes that we used in the examples are:

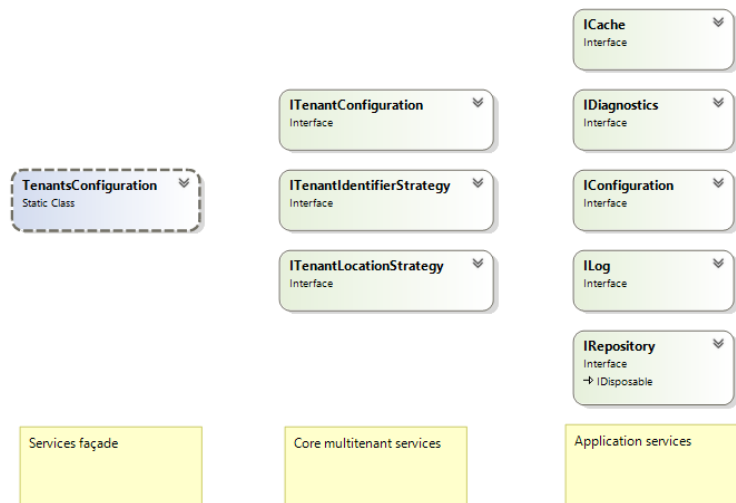


Figure 38: High level interfaces

And their common implementations:

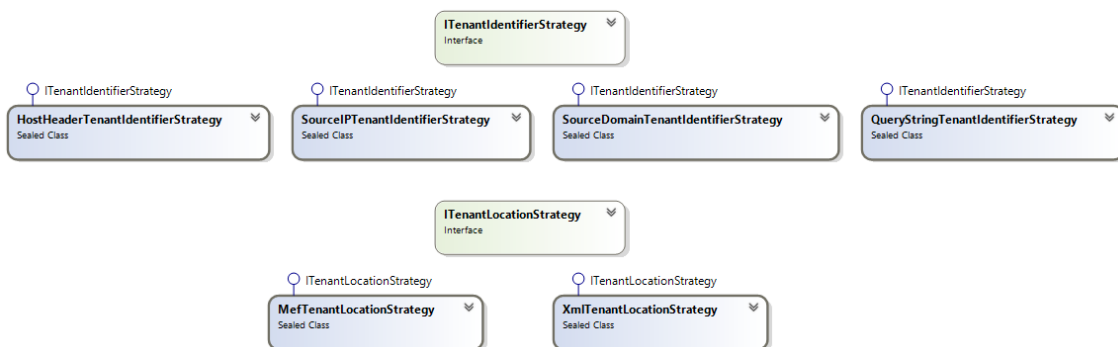


Figure 39: Typical implementations of high level interfaces

Choosing the right tenant identification strategy

All of the tenant identification strategies presented in Chapter 3 have some pros and cons; some require some additional configuration, others only really apply to simple scenarios. Some of the criteria include:

- Ease of testing one or the other tenant
- The need to bind a tenant to a specific source (IP range or domain)
- Ease of setting it up

You can use the following decision flow:

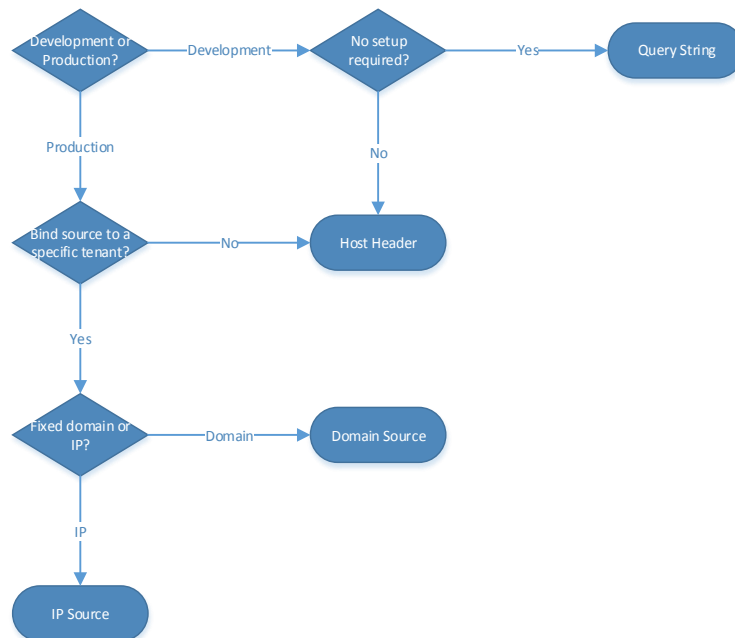


Figure 40: Simple decision flow for choosing the tenant identification strategy

Choosing the right tenant location strategy

When choosing a tenant location (and registration strategy), you need to take into account these questions:

- Do you need to register new tenants dynamically?
- How much effort is acceptable for registering new tenants (e.g., changing some configuration file, recompiling the application, restarting it)?
- Are some aspects of tenants only configurable through configuration?

You need to weigh all of these constraints before you make a decision. The following picture summarizes this:

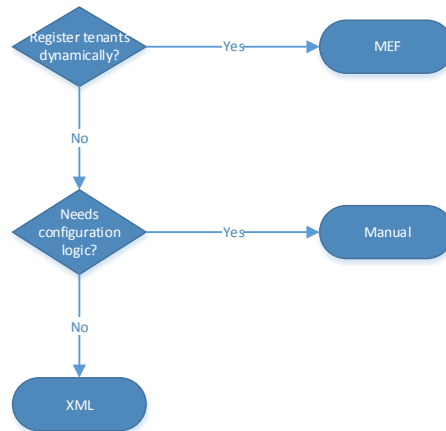


Figure 41: Simple decision flow for choosing the tenant location strategy

Choosing the right data access atrategy

The two APIs presented, NHibernate and Entity Framework, support all of the data access strategies. In the end, I think the decisive factor is the level of isolation that we want: separate databases offer the highest level, and data partitioning through discriminator columns offers the lowest. Criteria might include:

- Contractual requirements
- SLAs
- Regulatory demands
- Technical reasons
- Need to charge per used space

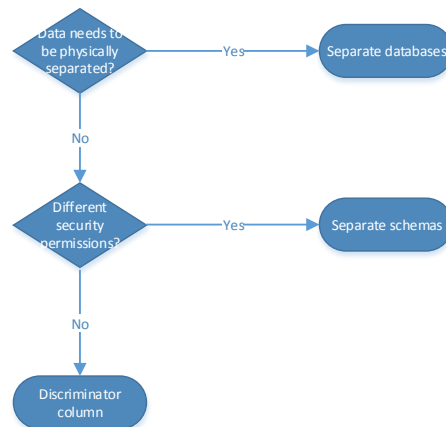


Figure 42: Decision flow for selecting data separation strategy

As for the APIs themselves, there will be probably some benefit in terms of ease of use with Entity Framework, but it all goes down to what the developer knows best, although it's true that having separate schemas is slightly trickier with NHibernate. In any case, using an ORM seems a wiser decision than sticking with plain ADO.NET, because of all of the additional services that ORMs provide. Using the Generic Repository pattern to abstract away the data access API might be a good idea only if you do not need any of the API-specific features, and just stick with the least common denominator.

Monitoring

Even if you do not use custom performance counters, the built-in ones offer great tools when it comes to monitoring—and perhaps, billing—individual tenants. Some performance counters offer good insights on what's going on, tenant-wise (you can select the sites to monitor independently):

- **ASP.NET Applications**
- **ASP.NET Apps v4.0.30319**
- **APP_POOL_WAS** (application pool information)

You can view reports on some of these metrics, which can help you make decisions or serve as a billing basis:

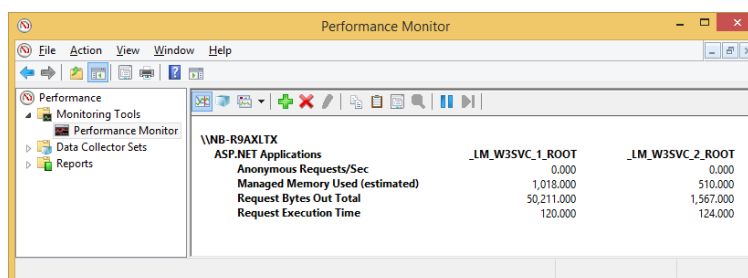


Figure 43: Reporting on performance counters

It is even possible to add alerts to be triggered when certain values exceed their normal thresholds:

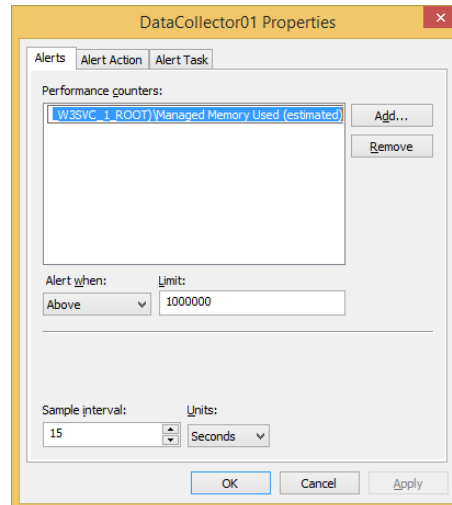


Figure 44: Adding a performance counter alert

This example shows a rule for generating an alert for when the **Managed Memory Used (estimated)** metric of the **ASP.NET Applications** counter exceeds 1,000,000, for the **_LM_W3SVC_1_ROOT** site (**abc.com**) which is evaluated every 15 seconds.

Layout

If you need to support different layouts per tenant, make sure you follow modern guidelines for defining the markup of your pages; namely, enforce a strict separation between content (HTML) and layout (CSS) and use the proper HTML elements for organizing your contents. For example, do not use the [Table](#) element for layout or grouping contents, because it does not scale well and imposes limitations on what we can change.



Note: The [CSS Zen Garden site](#) offers fantastic examples on the power of CSS.

References

- [As Easy As Falling Off a Log: Using the Logging Application Block](#)
- [ASP.NET 2.0 Provider Model](#)
- [ASP.NET 2.0 Provider Model: Introduction to the Provider Model](#)
- [Common Service Locator](#)
- [Concept mapping between SQL Server and Oracle](#)
- [Configuring/Mapping Properties and Types with the Fluent API](#)
- [CSS Zen Garden](#)
- [Developing Multi-tenant Applications for the Cloud, 3rd Edition](#)
- [Difference Between CName and A Record](#)
- [Enterprise Library - Data Access Block](#)
- [Enterprise Library - Logging Application Block](#)
- [Enterprise Library - Logging Application Block Database Provider](#)
- [Entity Framework](#)
- [Entity Framework Code First Inheritance](#)
- [Hands-On Labs for Microsoft Enterprise Library 6](#)
- [How to view the ASP.NET Performance Counters Available on Your Computer](#)
- [Katana Project](#)
- [Monitoring ASP.NET Application Performance](#)
- [Much ADO about Data Access: Using the Data Access Application Block](#)
- [Multitenancy](#)
- [NHibernate](#)
- [OWIN](#)
- [OWIN Middleware in the IIS integrated pipeline](#)
- [Performance Counters for ASP.NET](#)
- [PerfView](#)
- [Single Table Inheritance](#)
- [Understanding multi-tenancy in SharePoint Server 2013](#)
- [WatiN](#)