Table of Contents

Network Programming in the .NET Framework

7/29/2017 • 4 min to read • Edit Online

The Microsoft .NET Framework provides a layered, extensible, and managed implementation of Internet services that can be quickly and easily integrated into your applications. Your network applications can build on pluggable protocols to automatically take advantage of new Internet protocols, or they can use a managed implementation of the Windows socket interface to work with the network on the socket level.

In This Section

Introducing Pluggable Protocols
Describes how to access an Internet resource without regard to the access protocol that it requires.

Requesting Data
Explains how to use pluggable protocols to upload and download data from Internet resources.

Programming Pluggable Protocols
Explains how to derive protocol-specific classes to implement pluggable protocols.

Using Application Protocols
Describes programming applications that take advantage of network protocols such as TCP, UDP, and HTTP.

Internet Protocol Version 6
Describes the advantages of Internet Protocol version 6 (IPv6) over the current version of the Internet Protocol suite (IPv4), describes IPv6 addressing, routing and auto-configuration, and how to enable and disable IPv6.

Configuring Internet Applications
Explains how to use the .NET Framework configuration files to configure Internet applications.

Network Tracing in the .NET Framework
Explains how to use network tracing to get information about method invocations and network traffic generated by a managed application.

Cache Management for Network Applications
Describes how to use caching for applications that use the System.Net.WebClient, System.Net.WebRequest, and System.Net.HttpWebRequest classes.

Security in Network Programming
Describes how to use standard Internet security and authentication techniques.

Best Practices for System.Net Classes
Provides tips and tricks for getting the most out of your Internet applications.

Accessing the Internet Through a Proxy
Describes how to configure proxies.

NetworkInformation
Describes how to gather information about network events, changes, statistics, and properties and also explains how to determine whether a remote host is reachable by using the System.Net.NetworkInformation.Ping class.

Changes to the System.Uri namespace in Version 2.0
Describes several changes made to the System.Uri class in Version 2.0 to fixed incorrect behavior, enhance usability, and enhance security.

International Resource Identifier Support in System.Uri
Describes enhancements to the System.Uri class in Version 3.5, 3.0 SP1, and 2.0 SP1 for International Resource Identifier (IRI) and Internationalized Domain Name (IDN) support.

Socket Performance Enhancements in Version 3.5
Describes a set of enhancements to the System.Net.Sockets.Socket class in Version 3.5, 3.0 SP1, and 2.0 SP1 that provide an alternative asynchronous pattern that can be used by specialized high-performance socket applications.

Peer Name Resolution Protocol
Describes support added in Version 3.5 to support the Peer Name Resolution Protocol (PNRP), a serverless and dynamic name registration and name resolution protocol. These new features are supported by the System.Net.PeerToPeer namespace.

Peer-to-Peer Collaboration
Describes support added in Version 3.5 to support the Peer-to-Peer Collaboration that builds on PNRP. These new features are supported by the System.Net.PeerToPeer.Collaboration namespace.

Changes to NTLM authentication for HttpWebRequest in Version 3.5 SP1
Describes security changes made in Version 3.5 SP1 that affect how integrated Windows authentication is handled by the System.Net.HttpWebRequest, System.Net.HttpListener, System.Net.Security.NegotiateStream, and related classes in the System.Net namespace.

Integrated Windows Authentication with Extended Protection
Describes enhancements for extended protection that affect how integrated Windows authentication is handled by the System.Net.HttpWebRequest, System.Net.HttpListener, System.Net.Mail.SmtpClient, System.Net.Security.SslStream, System.Net.Security.NegotiateStream, and related classes in the System.Net and related namespaces.

NAT Traversal using IPv6 and Teredo
Describes enhancements added to the System.Net, System.Net.NetworkInformation, and System.Net.Sockets namespaces to support NAT traversal using IPv6 and Teredo.

Network Isolation for Windows Store Apps
Describes the impact of network isolation when classes in the System.Net, System.Net.Http, and System.Net.Http.Headers namespaces are used in Windows 8.x Store apps.

Network Programming Samples
Links to downloadable network programming samples that use classes in the System.Net, System.Net.Cache, System.Net.Configuration, System.Net.Mail, System.Net.Mime, System.Net.NetworkInformation, System.Net.PeerToPeer, System.Net.Security, System.Net.Sockets namespaces.

Reference

System.Net
Provides a simple programming interface for many of the protocols used on networks today. The System.Net.WebRequest and System.Net.WebResponse classes in this namespace are the basis for pluggable protocols.

System.Net.Cache
Defines the types and enumerations used to define cache policies for resources obtained using the System.Net.WebRequest and System.Net.HttpWebRequest classes.

System.Net.Configuration
Classes that applications use to programmatically access and update configuration settings for the System.Net namespaces.

System.Net.Http
Classes that provides a programming interface for modern HTTP applications.

System.Net.Http.Headers
Provides support for collections of HTTP headers used by the System.Net.Http namespace

System.Net.Mail
Classes to compose and send mail using the SMTP protocol.

System.Net.Mime
Defines types that are used to represent Multipurpose Internet Mail Exchange (MIME) headers used by classes in the System.Net.Mail namespace.

System.Net.NetworkInformation
Classes to programmatically gather information about network events, changes, statistics, and properties.

System.Net.PeerToPeer
Provides a managed implementation of the Peer Name Resolution Protocol (PNRP) for developers.

System.Net.PeerToPeer.Collaboration
Provides a managed implementation of the Peer-to-Peer Collaboration interface for developers.

System.Net.Security
Classes to provide network streams for secure communications between hosts.

System.Net.Sockets
Provides a managed implementation of the Windows Sockets (Winsock) interface for developers who need to help control access to the network.

System.Net.WebSockets
Provides a managed implementation of the WebSocket interface for developers.

System.Uri
Provides an object representation of a uniform resource identifier (URI) and easy access to the parts of the URI.

System.Security.Authentication.ExtendedProtection
Provides support for authentication using extended protection for applications.

System.Security.Authentication.ExtendedProtection.Configuration
Provides support for configuration of authentication using extended protection for applications.

See Also

Network Programming How-to Topics
Network Programming Samples
Networking Samples for .NET on MSDN Code Gallery
HttpClient Sample

Network Programming How-to Topics

7/29/2017 • 1 min to read • Edit Online

The following list includes links to the How-to topics found in the conceptual documentation for network programming.

Requesting Data:

- How to: Request a Web Page and Retrieve the Results as a Stream
- How to: Request Data Using the WebRequest Class
- How to: Send Data Using the WebRequest Class
- How to: Retrieve a Protocol-Specific WebResponse that Matches a WebRequest

Pluggable and Application Protocols:

- How to: Register a Custom Protocol Using WebRequest
- How to: Typecast a WebRequest to Access Protocol Specific Properties
- How to: Access HTTP-Specific Properties
- How to: Assign User Information to Group Connections
- How to: Create a Socket
- How to: Download Files with FTP
- How to: Upload Files with FTP
- How to: List Directory Contents with FTP

Internet Protocol Version 6:

- How to: Modify the Computer Configuration File to Enable IPv6 Support

Network Tracing:

- How to: Configure Network Tracing

Configuring Caching:

- How to: Set a Location-Based Cache Policy for an Application
- How to: Set the Default Time-Based Cache Policy for an Application
- How to: Customize a Time-Based Cache Policy
- How to: Set Cache Policy for a Request

Using Proxies:

- How to: Enable a WebRequest to Use a Proxy to Communicate With the Internet
- How to: Override a Global Proxy Selection

Network Information:

- How to: Detect Network Availability and Address Changes
- How to: Get Interface and Protocol Information
- How to: Ping a Host

See Also

Network Programming in the .NET Framework
Network Programming Samples
Networking Samples for .NET on MSDN Code Gallery

Introducing Pluggable Protocols

7/29/2017 • 5 min to read • Edit Online

The Microsoft .NET Framework provides a layered, extensible, and managed implementation of Internet services that can be integrated quickly and easily into your applications. The Internet access classes in the System.Net and System.Net.Sockets namespaces can be used to implement both Web-based and Internet-based applications.

Internet Applications

Internet applications can be classified broadly into two kinds: client applications that request information and server applications that respond to information requests from clients. The classic Internet client-server application is the World Wide Web, where people use browsers to access documents and other data stored on Web servers worldwide.

Applications are not limited to just one of these roles; for instance, the familiar middle-tier application server responds to requests from clients by requesting data from another server, in which case it is acting as both a server and a client.

The client application makes a request by identifying the requested Internet resource and the communication protocol to use for the request and response. If necessary, the client also provides any additional data required to complete the request, such as proxy location or authentication information (user name, password, and so on). Once the request is formed, the request can be sent to the server.

Identifying Resources

The .NET Framework uses a Uniform Resource Identifier (URI) to identify the requested Internet resource and communication protocol. The URI consists of at least three, and possibly four, fragments: the scheme identifier, which identifies the communications protocol for the request and response; the server identifier, which consists of either a Domain Name System (DNS) host name or a TCP address that uniquely identifies the server on the Internet; the path identifier, which locates the requested information on the server; and an optional query string, which passes information from the client to the server. For example, the URI "http://www.contoso.com/whatsnew.aspx?date=today" consists of the scheme identifier "http", the server identifier "www.contoso.com", the path "/whatsnew.aspx", and the query string "?date=today".

After the server has received the request and processed the response, it returns the response to the client application. The response includes supplemental information, such as the type of the content (raw text or XML data, for example).

Requests and Responses in the .NET Framework

The .NET Framework uses specific classes to provide the three pieces of information required to access Internet resources through a request/response model: the Uri class, which contains the URI of the Internet resource you are seeking; the WebRequest class, which contains a request for the resource; and the WebResponse class, which provides a container for the incoming response.

Client applications create `WebRequest` instances by passing the URI of the network resource to the Create method. This static method creates a `WebRequest` for a specific protocol, such as HTTP. The `WebRequest` that is returned provides access to properties that control both the request to the server and access to the data stream that is sent when the request is made. The GetResponse method on the `WebRequest` sends the request from the client application to the server identified in the URI. In cases in which the response might be delayed, the request can be made asynchronously using the BeginGetResponse method on the **WebRequest**, and the response can be

returned at a later time using the EndGetResponse method.

The **GetResponse** and **EndGetResponse** methods return a **WebResponse** that provides access to the data returned by the server. Because this data is provided to the requesting application as a stream by the GetResponseStream method, it can be used in an application anywhere data streams are used.

The **WebRequest** and **WebResponse** classes are the basis of pluggable protocols — an implementation of network services that enables you to develop applications that use Internet resources without worrying about the specific details of the protocol that each resource uses. Descendant classes of **WebRequest** are registered with the **WebRequest** class to manage the details of making the actual connections to Internet resources.

As an example, the HttpWebRequest class manages the details of connecting to an Internet resource using HTTP. By default, when the **WebRequest.Create** method encounters a URI that begins with "http:" or "https:" (the protocol identifiers for HTTP and secure HTTP), the **WebRequest** that is returned can be used as is, or it can be typecast to **HttpWebRequest** to access protocol-specific properties. In most cases, the **WebRequest** provides all the necessary information for making a request.

Any protocol that can be represented as a request/response transaction can be used in a **WebRequest**. You can derive protocol-specific classes from **WebRequest** and **WebResponse** and then register them for use by the application with the static System.Net.WebRequest.RegisterPrefix method.

When client authorization for Internet requests is required, the Credentials property of the **WebRequest** supplies the necessary credentials. These credentials can be a simple name/password pair for basic HTTP or digest authentication, or a name/password/domain set for NTLM or Kerberos authentication. One set of credentials can be stored in a NetworkCredential instance, or multiple sets can be stored simultaneously in a CredentialCache instance. The **CredentialCache** uses the URI of the request and the authentication scheme that the server supports to determine which credentials to send to the server.

Simple Requests with WebClient

For applications that need to make simple requests for Internet resources, the WebClient class provides common methods for uploading data to or downloading data from an Internet server. **WebClient** relies on the **WebRequest** class to provide access to Internet resources; therefore, the **WebClient** class can use any registered pluggable protocol.

For applications that cannot use the request/response model, or for applications that need to listen on the network as well as send requests, the **System.Net.Sockets** namespace provides the TcpClient, TcpListener, and UdpClient classes. These classes handle the details of making connections using different transport protocols and expose the network connection to the application as a stream.

Developers familiar with the Windows Sockets interface or those who need the control provided by programming at the socket level will find that the **System.Net.Sockets** classes meet their needs. The **System.Net.Sockets** classes are a transition point from managed to native code within the **System.Net** classes. In most cases, **System.Net.Sockets** classes marshal data into their Windows 32-bit counterparts, as well as handling any necessary security checks.

See Also

Programming Pluggable Protocols
Network Programming in the .NET Framework
Network Programming Samples
Networking Samples for .NET on MSDN Code Gallery

Requesting Data

7/29/2017 • 2 min to read • Edit Online

Developing applications that run in the distributed operating environment of today's Internet requires an efficient, easy-to-use method for retrieving data from resources of all types. Pluggable protocols let you develop applications that use a single interface to retrieve data from multiple Internet protocols.

Uploading and Downloading Data from an Internet Server

For simple request and response transactions, the WebClient class provides the easiest method for uploading data to or downloading data from an Internet server. **WebClient** provides methods for uploading and downloading files, sending and receiving streams, and sending a data buffer to the server and receiving a response. **WebClient** uses the WebRequest and WebResponse classes to make the actual connections to the Internet resource, so any registered pluggable protocol is available for use.

Client applications that need to make more complex transactions request data from servers using the **WebRequest** class and its descendants. **WebRequest** encapsulates the details of connecting to the server, sending the request, and receiving the response. **WebRequest** is an abstract class that defines a set of properties and methods that are available to all applications that use pluggable protocols. Descendants of **WebRequest**, such as HttpWebRequest, implement the properties and methods defined by **WebRequest** in a way that is consistent with the underlying protocol.

The **WebRequest** class creates protocol-specific instances of **WebRequest** descendants, using the value of the URI passed to its Create method to determine the specific derived-class instance to create. Applications indicate which **WebRequest** descendant should be used to handle a request by registering the descendant's constructor with the System.Net.WebRequest.RegisterPrefix method.

A request is made to the Internet resource by calling the GetResponse method on the **WebRequest**. The **GetResponse** method constructs the protocol-specific request from the properties of the **WebRequest**, makes the TCP or UDP socket connection to the server, and sends the request. For requests that send data to the server, such as HTTP **Post** or FTP **Put** requests, the System.Net.WebRequest.GetRequestStream method provides a network stream in which to send the data.

The **GetResponse** method returns a protocol-specific **WebResponse** that matches the **WebRequest.**

The **WebResponse** class is also an abstract class that defines properties and methods that are available to all applications that use pluggable protocols. **WebResponse** descendants implement these properties and methods for the underlying protocol. The HttpWebResponse class, for example, implements the **WebResponse** class for HTTP.

The data returned by the server is presented to the application in the stream returned by the System.Net.WebResponse.GetResponseStream method. You can use this stream like any other, as shown in the following example.

```
StreamReader sr =
   new StreamReader(resp.GetResponseStream(), Encoding.ASCII);
```

```
Dim sr As StreamReader
sr = New StreamReader(resp.GetResponseStream(), Encoding.ASCII)
```

See Also

Network Programming in the .NET Framework
How to: Request a Web Page and Retrieve the Results as a Stream
How to: Retrieve a Protocol-Specific WebResponse that Matches a WebRequest

Creating Internet Requests

7/29/2017 • 1 min to read • Edit Online

Applications create WebRequest instances through the System.Net.WebRequest.Create method. This is a static method that creates a class derived from **WebRequest** based on the URI scheme passed to it.

Web, File and FTP Requests

The .NET Framework provides the HttpWebRequest class, which is derived from **WebRequest**, to handle HTTP and HTTPS requests. In most cases, the **WebRequest** class provides all the properties you need to make a request; however, if necessary, you can cast **WebRequest** objects created by the **WebRequest.Create** method to the **HttpWebRequest** type to access the HTTP-specific properties of the request. Similarly, the **HttpWebResponse** object handles the responses from HTTP and HTTPS requests. To access the HTTP-specific properties of the **HttpWebResponse** object, you need to cast **WebResponse** objects to the **HttpWebResponse** type.

The .NET Framework also provides the FileWebRequest and FileWebResponse classes to handle requests for resources that use the "file:" URI scheme. Likewise, the FtpWebRequest and FtpWebResponse classes are provided to handle requests for resources that use the "ftp:" scheme. If your request is for a resource that uses any of these schemes, you can use the **WebRequest.Create** method to obtain an object with which to make your request.

To handle requests that use other application-level protocols, you need to implement protocol-specific classes derived from **WebRequest** and **WebResponse**. For more information, see Programming Pluggable Protocols.

See Also

How to: Request Data Using the WebRequest Class
Requesting Data

How to: Request a Web Page and Retrieve the Results as a Stream

7/29/2017 • 1 min to read • Edit Online

This example shows how to request a Web page and retrieve the results in a stream.

Example

```
WebClient myClient = new WebClient();
Stream response = myClient.OpenRead("http://www.contoso.com/index.htm");
// The stream data is used here.
response.Close();
```

```
Dim myClient As WebClient = New WebClient()
Dim response As Stream = myClient.OpenRead("http://www.contoso.com/index.htm")
' The stream data is used here.
response.Close()
```

Compiling the Code

This example requires:

- References to the System.IO and System.Net namespaces.

See Also

Requesting Data

How to: Request Data Using the WebRequest Class

7/29/2017 • 3 min to read • Edit Online

The following procedure describes the steps used to request a resource from a server, for example, a Web page or file. The resource must be identified by a URI.

To request data from a host server

1. Create a WebRequest instance by calling Create with the URI of the resource.

```
WebRequest request = WebRequest.Create("http://www.contoso.com/");
```

```
Dim request as WebRequest = WebRequest.Create("http://www.contoso.com/")
```

> **NOTE**
> The .NET Framework provides protocol-specific classes derived from **WebRequest** and **WebResponse** for URIs that begin with "http:", "https:'', "ftp:", and "file:". To access resources using other protocols, you must implement protocol-specific classes that derive from **WebRequest** and **WebResponse**. For more information, see Programming Pluggable Protocols .

2. Set any property values that you need in the **WebRequest**. For example, to enable authentication, set the **Credentials** property to an instance of the NetworkCredential class.

```
request.Credentials = CredentialCache.DefaultCredentials;
```

```
request.Credentials = CredentialCache.DefaultCredentials
```

In most cases, the **WebRequest** class is sufficient to receive data. However, if you need to set protocol-specific properties, you must cast the **WebRequest** to the protocol-specific type. For example, to access the HTTP-specific properties of HttpWebRequest, cast the **WebRequest** to an **HttpWebRequest** reference. The following code example shows how to set the HTTP-specific UserAgent property.

```
((HttpWebRequest)request).UserAgent = ".NET Framework Example Client";
```

```
Ctype(request,HttpWebRequest).UserAgent = ".NET Framework Example Client"
```

3. To send the request to the server, call GetResponse. The actual type of the returned **WebResponse** object is determined by the scheme of the requested URI.

```
WebResponse response = request.GetResponse();
```

```
Dim response As WebResponse = request.GetResponse()
```

> **NOTE**
>
> After you are finished with a WebResponse object, you must close it by calling the Close method. Alternatively, if you have gotten the response stream from the response object, you can close the stream by calling the System.IO.Stream.Close method. If you do not close either the response or the stream, your application can run out of connections to the server and become unable to process additional requests.

4. You can access the properties of the **WebResponse** or cast the **WebResponse** to a protocol-specific instance to read protocol-specific properties. For example, to access the HTTP-specific properties of HttpWebResponse, cast the **WebResponse** to a **HttpWebResponse** reference. The following code example shows how to display the status information sent with a response.

```
Console.WriteLine (((HttpWebResponse)response).StatusDescription);
```

```
Console.WriteLine(CType(response,HttpWebResponse).StatusDescription)
```

5. To get the stream containing response data sent by the server, use the GetResponseStream method of the **WebResponse**.

```
Stream dataStream = response.GetResponseStream ();
```

```
Dim dataStream As Stream = response.GetResponseStream()
```

6. After reading the data from the response, you must either close the response stream using the **Stream.Close** method or close the response using the **WebResponse.Close** method. It is not necessary to call the **Close** method on both the response stream and the **WebResponse**, but doing so is not harmful. **WebResponse.Close** calls **Stream.Close** when closing the response.

```
response.Close();
```

```
response.Close()
```

Example

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

namespace Examples.System.Net
{
    public class WebRequestGetExample
    {
        public static void Main ()
        {
            // Create a request for the URL.
            WebRequest request = WebRequest.Create (
              "http://www.contoso.com/default.html");
            // If required by the server, set the credentials.
            request.Credentials = CredentialCache.DefaultCredentials;
            // Get the response.
            WebResponse response = request.GetResponse ();
            // Display the status.
            Console.WriteLine (((HttpWebResponse)response).StatusDescription);
            // Get the stream containing content returned by the server.
            Stream dataStream = response.GetResponseStream ();
            // Open the stream using a StreamReader for easy access.
            StreamReader reader = new StreamReader (dataStream);
            // Read the content.
            string responseFromServer = reader.ReadToEnd ();
            // Display the content.
            Console.WriteLine (responseFromServer);
            // Clean up the streams and the response.
            reader.Close ();
            response.Close ();
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Net
Imports System.Text
Namespace Examples.System.Net
    Public Class WebRequestGetExample

        Public Shared Sub Main()
            ' Create a request for the URL.
            Dim request As WebRequest = _
              WebRequest.Create("http://www.contoso.com/default.html")
            ' If required by the server, set the credentials.
            request.Credentials = CredentialCache.DefaultCredentials
            ' Get the response.
            Dim response As WebResponse = request.GetResponse()
            ' Display the status.
            Console.WriteLine(CType(response,HttpWebResponse).StatusDescription)
            ' Get the stream containing content returned by the server.
            Dim dataStream As Stream = response.GetResponseStream()
            ' Open the stream using a StreamReader for easy access.
            Dim reader As New StreamReader(dataStream)
            ' Read the content.
            Dim responseFromServer As String = reader.ReadToEnd()
            ' Display the content.
            Console.WriteLine(responseFromServer)
            ' Clean up the streams and the response.
            reader.Close()
            response.Close()
        End Sub
    End Class
End Namespace
```

See Also

Creating Internet Requests
Using Streams on the Network
Accessing the Internet Through a Proxy
Requesting Data
How to: Send Data Using the WebRequest Class

How to: Send Data Using the WebRequest Class

7/29/2017 • 4 min to read • Edit Online

The following procedure describes the steps used to send data to a server. This procedure is commonly used to post data to a Web page.

To send data to a host server

1. Create a WebRequest instance by calling Create with the URI of the resource that accepts data, for example, a script or ASP.NET page.

```
WebRequest request = WebRequest.Create("http://www.contoso.com/");
```

```
Dim request as WebRequest = WebRequest.Create("http://www.contoso.com/")
```

> **NOTE**
> The .NET Framework provides protocol-specific classes derived from **WebRequest** and **WebResponse** for URIs that begin with "http:", "https:", "ftp:", and "file:". To access resources using other protocols, you must implement protocol-specific classes that derive from **WebRequest** and **WebResponse**. For more information, see Programming Pluggable Protocols .

2. Set any property values that you need in the **WebRequest**. For example, to enable authentication, set the **Credentials** property to an instance of the NetworkCredential class.

```
request.Credentials = CredentialCache.DefaultCredentials;
```

```
request.Credentials = CredentialCache.DefaultCredentials
```

   In most cases, the **WebRequest** instance itself is sufficient to send data. However, if you need to set protocol-specific properties, you must cast the **WebRequest** to the protocol-specific type. For example, to access the HTTP-specific properties of HttpWebRequest, cast the **WebRequest** to an **HttpWebRequest** reference. The following code example shows how to set the HTTP-specific UserAgent property.

```
((HttpWebRequest)request).UserAgent = ".NET Framework Example Client";
```

```
Ctype(request,HttpWebRequest).UserAgent = ".NET Framework Example Client"
```

3. Specify a protocol method that permits data to be sent with a request, such as the HTTP **POST** method.

```
request.Method = "POST";
```

```
request.Method = "POST"
```

4. Set the **ContentLength** property.

```
request.ContentLength = byteArray.Length;
```

```
request.ContentLength = byteArray.Length
```

5. Set the **ContentType** property to an appropriate value.

```
request.ContentType = "application/x-www-form-urlencoded";
```

```
request.ContentType = "application/x-www-form-urlencoded"
```

6. Get the stream that holds request data by calling the GetRequestStream method.

```
Stream dataStream = request.GetRequestStream ();
```

```
Stream dataStream = request.GetRequestStream ()
```

7. Write the data to the Stream object returned by this method.

```
dataStream.Write (byteArray, 0, byteArray.Length);
```

```
dataStream.Write (byteArray, 0, byteArray.Length)
```

8. Close the request stream by calling the **Stream.Close** method.

```
dataStream.Close ();
```

```
dataStream.Close ()
```

9. Send the request to the server by calling GetResponse. This method returns an object containing the server's response. The returned WebResponse object's type is determined by the scheme of the request's URI.

```
WebResponse response = request.GetResponse();
```

```
Dim response As WebResponse = request.GetResponse()
```

> **NOTE**
>
> After you are finished with a WebResponse object, you must close it by calling the Close method. Alternatively, if you have gotten the response stream from the response object, you can close the stream by calling the System.IO.Stream.Close method. If you do not close the response or the stream, your application can run out of connections to the server and become unable to process additional requests.

10. You can access the properties of the **WebResponse** or cast the **WebResponse** to a protocol-specific instance to read protocol-specific properties. For example, to access the HTTP-specific properties of HttpWebResponse, cast the **WebResponse** to an **HttpWebResponse** reference.

```
Console.WriteLine (((HttpWebResponse)response).StatusDescription);
```

```
Console.WriteLine(CType(response, HttpWebResponse).StatusDescription)
```

11. To get the stream containing response data sent by the server, call the GetResponseStream method of the **WebResponse**.

```
Stream data = response.GetResponseStream;
```

```
Dim data As Stream = response.GetResponseStream
```

12. After reading the data from the response, you must either close the response stream using the **Stream.Close** method or close the response using the **WebResponse.Close** method. It is not necessary to call the **Close** method on both the response stream and the **WebResponse**, but doing so is not harmful.

```
response.Close();
```

```
response.Close()
```

Example

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

namespace Examples.System.Net
{
    public class WebRequestPostExample
    {
        public static void Main ()
        {
            // Create a request using a URL that can receive a post.
            WebRequest request = WebRequest.Create ("http://www.contoso.com/PostAccepter.aspx ");
            // Set the Method property of the request to POST.
            request.Method = "POST";
            // Create POST data and convert it to a byte array.
            string postData = "This is a test that posts this string to a Web server.";
            byte[] byteArray = Encoding.UTF8.GetBytes (postData);
            // Set the ContentType property of the WebRequest.
            request.ContentType = "application/x-www-form-urlencoded";
            // Set the ContentLength property of the WebRequest.
            request.ContentLength = byteArray.Length;
            // Get the request stream.
            Stream dataStream = request.GetRequestStream ();
            // Write the data to the request stream.
            dataStream.Write (byteArray, 0, byteArray.Length);
            // Close the Stream object.
            dataStream.Close ();
            // Get the response.
            WebResponse response = request.GetResponse ();
            // Display the status.
            Console.WriteLine (((HttpWebResponse)response).StatusDescription);
            // Get the stream containing content returned by the server.
            dataStream = response.GetResponseStream ();
            // Open the stream using a StreamReader for easy access.
            StreamReader reader = new StreamReader (dataStream);
            // Read the content.
            string responseFromServer = reader.ReadToEnd ();
            // Display the content.
            Console.WriteLine (responseFromServer);
            // Clean up the streams.
            reader.Close ();
            dataStream.Close ();
            response.Close ();
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Net
Imports System.Text
Namespace Examples.System.Net
    Public Class WebRequestPostExample

        Public Shared Sub Main()
            ' Create a request using a URL that can receive a post.
            Dim request As WebRequest = WebRequest.Create("http://www.contoso.com/PostAccepter.aspx ")
            ' Set the Method property of the request to POST.
            request.Method = "POST"
            ' Create POST data and convert it to a byte array.
            Dim postData As String = "This is a test that posts this string to a Web server."
            Dim byteArray As Byte() = Encoding.UTF8.GetBytes(postData)
            ' Set the ContentType property of the WebRequest.
            request.ContentType = "application/x-www-form-urlencoded"
            ' Set the ContentLength property of the WebRequest.
            request.ContentLength = byteArray.Length
            ' Get the request stream.
            Dim dataStream As Stream = request.GetRequestStream()
            ' Write the data to the request stream.
            dataStream.Write(byteArray, 0, byteArray.Length)
            ' Close the Stream object.
            dataStream.Close()
            ' Get the response.
            Dim response As WebResponse = request.GetResponse()
            ' Display the status.
            Console.WriteLine(CType(response, HttpWebResponse).StatusDescription)
            ' Get the stream containing content returned by the server.
            dataStream = response.GetResponseStream()
            ' Open the stream using a StreamReader for easy access.
            Dim reader As New StreamReader(dataStream)
            ' Read the content.
            Dim responseFromServer As String = reader.ReadToEnd()
            ' Display the content.
            Console.WriteLine(responseFromServer)
            ' Clean up the streams.
            reader.Close()
            dataStream.Close()
            response.Close()
        End Sub
    End Class
End Namespace
```

See Also

Creating Internet Requests
Using Streams on the Network
Accessing the Internet Through a Proxy
Requesting Data
How to: Request Data Using the WebRequest Class

How to: Retrieve a Protocol-Specific WebResponse that Matches a WebRequest

7/29/2017 • 1 min to read • Edit Online

This example shows how to retrieve a protocol-specific WebResponse that matches a WebRequest.

Example

```
WebRequest req = WebRequest.Create("http://www.contoso.com/");
WebResponse resp = req.GetResponse();
```

```
Dim req As WebRequest = WebRequest.Create("http://www.contoso.com")
Dim resp As WebResponse = req.GetResponse()
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

See Also

Requesting Data

Using Streams on the Network

7/29/2017 • 3 min to read • Edit Online

Network resources are represented in the .NET Framework as streams. By treating streams generically, the .NET Framework offers the following capabilities:

- A common way to send and receive Web data. Whatever the actual contents of the file — HTML, XML, or anything else — your application will use System.IO.Stream.Write and System.IO.Stream.Read to send and receive data.
- Compatibility with streams across the .NET Framework. Streams are used throughout the .NET Framework, which has a rich infrastructure for handling them. For example, you can modify an application that reads XML data from a FileStream to read data from a NetworkStream instead by changing only the few lines of code that initialize the stream. The major differences between the **NetworkStream** class and other streams are that **NetworkStream** is not seekable, the CanSeek property always returns **false**, and the Seek and Position methods throw a NotSupportedException.
- Processing of data as it arrives. Streams provide access to data as it arrives from the network, rather than forcing your application to wait for an entire data set to be downloaded.

The System.Net.Sockets namespace contains a **NetworkStream** class that implements the Stream class specifically for use with network resources. Classes in the System.Net.Sockets namespace use the **NetworkStream** class to represent streams.

To send data to the network using the returned stream, call GetRequestStream on your WebRequest. The **WebRequest** will send request headers to the server; then you can send data to the network resource by calling the BeginWrite, EndWrite, or Write method on the returned stream. Some protocols, such as HTTP, may require you to set protocol-specific properties before sending data. The following code example shows how to set HTTP-specific properties for sending data. It assumes that the variable `sendData` contains the data to send and that the variable `sendLength` is the number of bytes of data to send.

```
HttpWebRequest request =
   (HttpWebRequest) WebRequest.Create("http://www.contoso.com/");
request.Method = "POST";
request.ContentLength = sendLength;
try
{
   Stream sendStream = request.GetRequestStream();
   sendStream.Write(sendData,0,sendLength);
   sendStream.Close();
}
catch
{
   // Handle errors . . .
}
```

```
Dim request As HttpWebRequest = _
   CType(WebRequest.Create("http://www.contoso.com/"), HttpWebRequest)
request.Method = "POST"
request.ContentLength = sendLength
Try
   Dim sendStream As Stream = request.GetRequestStream()
   sendStream.Write(sendData, 0, sendLength)
   sendStream.Close()
Catch
   ' Handle errors . . .
End Try
```

To receive data from the network, call GetResponseStream on your WebResponse. You can then read data from the network resource by calling the BeginRead, EndRead, or Read method on the returned stream.

When using streams from network resources, keep in mind the following points:

- The **CanSeek** property always returns **false** since the **NetworkStream** class cannot change position in the stream. The **Seek** and **Position** methods throw a **NotSupportedException**.
- When you use **WebRequest** and **WebResponse**, stream instances created by calling **GetResponseStream** are read-only and stream instances created by calling **GetRequestStream** are write-only.
- Use the StreamReader class to make encoding easier. The following code example uses a **StreamReader** to read an ASCII-encoded stream from a **WebResponse** (the example does not show creating the request).
- The call to **GetResponse** can block if network resources are not available. You should consider using an asynchronous request with the BeginGetResponse and EndGetResponse methods.
- The call to **GetRequestStream** can block while the connection to the server is created. You should consider using an asynchronous request for the stream with the BeginGetRequestStream and EndGetRequestStream methods.

```
// Create a response object.
WebResponse response = request.GetResponse();
// Get a readable stream from the server.
StreamReader sr =
   new StreamReader(response.GetResponseStream(), Encoding.ASCII);
// Use the stream. Remember when you are through with the stream to close it.
sr.Close();
```

```
' Create a response object.
Dim response As WebResponse = request.GetResponse()
' Get a readable stream from the server.
Dim sr As _
   New StreamReader(response.GetResponseStream(), Encoding.ASCII)
' Use the stream. Remember when you are through with the stream to close it.
sr.Close()
```

See Also

How to: Request Data Using the WebRequest Class
Requesting Data

Making Asynchronous Requests

7/29/2017 • 7 min to read • Edit Online

The System.Net classes use the .NET Framework's standard asynchronous programming model for asynchronous access to Internet resources. The BeginGetResponse and EndGetResponse methods of the WebRequest class start and complete asynchronous requests for an Internet resource.

> **NOTE**
>
> Using synchronous calls in asynchronous callback methods can result in severe performance penalties. Internet requests made with **WebRequest** and its descendants must use System.IO.Stream.BeginRead to read the stream returned by the System.Net.WebResponse.GetResponseStream method.

The following sample code demonstrates how to use asynchronous calls with the **WebRequest** class. The sample is a console program that takes a URI from the command line, requests the resource at the URI, and then prints data to the console as it is received from the Internet.

The program defines two classes for its own use, the **RequestState** class, which passes data across asynchronous calls, and the **ClientGetAsync** class, which implements the asynchronous request to an Internet resource.

The **RequestState** class preserves the state of the request across calls to the asynchronous methods that service the request. It contains **WebRequest** and Stream instances that contain the current request to the resource and the stream received in response, a buffer that contains the data currently received from the Internet resource, and a StringBuilder that contains the complete response. A **RequestState**is passed as the *state* parameter when the AsyncCallback method is registered with **WebRequest.BeginGetResponse**.

The **ClientGetAsync** class implements an asynchronous request to an Internet resource and writes the resulting response to the console. It contains the methods and properties described in the following list.

- The `allDone` property contains an instance of the ManualResetEvent class that signals the completion of the request.
- The `Main()` method reads the command line and begins the request for the specified Internet resource. It creates the **WebRequest** `wreq` and the **RequestState** `rs`, calls **BeginGetResponse** to begin processing the request, and then calls the `allDone.WaitOne()` method so that the application will not exit until the callback is complete. After the response is read from the Internet resource, `Main()` writes it to the console and the application ends.
- The `showusage()` method writes an example command line on the console. It is called by `Main()` when no URI is provided on the command line.
- The `RespCallBack()` method implements the asynchronous callback method for the Internet request. It creates the **WebResponse** instance containing the response from the Internet resource, gets the response stream, and then starts reading the data from the stream asynchronously.
- The `ReadCallBack()` method implements the asynchronous callback method for reading the response stream. It transfers data received from the Internet resource into the **ResponseData** property of the **RequestState** instance, then starts another asynchronous read of the response stream until no more data is returned. Once all the data has been read, `ReadCallBack()` closes the response stream and calls the `allDone.Set()` method to indicate that the entire response is present in **ResponseData**.

NOTE

It is critical that all network streams are closed. If you do not close each request and response stream, your application will run out of connections to the server and be unable to process additional requests.

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

// The RequestState class passes data across async calls.
public class RequestState
{
   const int BufferSize = 1024;
   public StringBuilder RequestData;
   public byte[] BufferRead;
   public WebRequest Request;
   public Stream ResponseStream;
   // Create Decoder for appropriate enconding type.
   public Decoder StreamDecode = Encoding.UTF8.GetDecoder();

   public RequestState()
   {
      BufferRead = new byte[BufferSize];
      RequestData = new StringBuilder(String.Empty);
      Request = null;
      ResponseStream = null;
   }
}

// ClientGetAsync issues the async request.
class ClientGetAsync
{
   public static ManualResetEvent allDone = new ManualResetEvent(false);
   const int BUFFER_SIZE = 1024;

   public static void Main(string[] args)
   {
      if (args.Length < 1)
      {
         showusage();
         return;
      }

      // Get the URI from the command line.
      Uri httpSite = new Uri(args[0]);

      // Create the request object.
      WebRequest wreq = WebRequest.Create(httpSite);

      // Create the state object.
      RequestState rs = new RequestState();

      // Put the request into the state object so it can be passed around.
      rs.Request = wreq;

      // Issue the async request.
      IAsyncResult r = (IAsyncResult) wreq.BeginGetResponse(
         new AsyncCallback(RespCallback), rs);

      // Wait until the ManualResetEvent is set so that the application
      // does not exit until after the callback is called.
      allDone.WaitOne();

      Console.WriteLine(rs.RequestData.ToString());
```

```
      Console.WriteLine(rs.RequestData.ToString());
   }

   public static void showusage() {
      Console.WriteLine("Attempts to GET a URL");
      Console.WriteLine("\r\nUsage:");
      Console.WriteLine("   ClientGetAsync URL");
      Console.WriteLine("   Example:");
      Console.WriteLine("      ClientGetAsync http://www.contoso.com/");
   }

   private static void RespCallback(IAsyncResult ar)
   {
      // Get the RequestState object from the async result.
      RequestState rs = (RequestState) ar.AsyncState;

      // Get the WebRequest from RequestState.
      WebRequest req = rs.Request;

      // Call EndGetResponse, which produces the WebResponse object
      //  that came from the request issued above.
      WebResponse resp = req.EndGetResponse(ar);

      //  Start reading data from the response stream.
      Stream ResponseStream = resp.GetResponseStream();

      // Store the response stream in RequestState to read
      // the stream asynchronously.
      rs.ResponseStream = ResponseStream;

      //  Pass rs.BufferRead to BeginRead. Read data into rs.BufferRead
      IAsyncResult iarRead = ResponseStream.BeginRead(rs.BufferRead, 0,
         BUFFER_SIZE, new AsyncCallback(ReadCallBack), rs);
   }

   private static void ReadCallBack(IAsyncResult asyncResult)
   {
      // Get the RequestState object from AsyncResult.
      RequestState rs = (RequestState)asyncResult.AsyncState;

      // Retrieve the ResponseStream that was set in RespCallback.
      Stream responseStream = rs.ResponseStream;

      // Read rs.BufferRead to verify that it contains data.
      int read = responseStream.EndRead( asyncResult );
      if (read > 0)
      {
         // Prepare a Char array buffer for converting to Unicode.
         Char[] charBuffer = new Char[BUFFER_SIZE];

         // Convert byte stream to Char array and then to String.
         // len contains the number of characters converted to Unicode.
      int len =
         rs.StreamDecode.GetChars(rs.BufferRead, 0, read, charBuffer, 0);

         String str = new String(charBuffer, 0, len);

         // Append the recently read data to the RequestData stringbuilder
         // object contained in RequestState.
         rs.RequestData.Append(
            Encoding.ASCII.GetString(rs.BufferRead, 0, read));

         // Continue reading data until
         // responseStream.EndRead returns –1.
         IAsyncResult ar = responseStream.BeginRead(
            rs.BufferRead, 0, BUFFER_SIZE,
            new AsyncCallback(ReadCallBack), rs);
      }
      else
      {
```

```
      {
         if(rs.RequestData.Length>0)
         {
            //  Display data to the console.
            string strContent;
            strContent = rs.RequestData.ToString();
         }
         // Close down the response stream.
         responseStream.Close();
         // Set the ManualResetEvent so the main thread can exit.
         allDone.Set();
      }
      return;
   }
}
```

```
Imports System
Imports System.Net
Imports System.Threading
Imports System.Text
Imports System.IO

' The RequestState class passes data across async calls.
Public Class RequestState

      Public RequestData As New StringBuilder("")
      Public BufferRead(1024) As Byte
      Public Request As HttpWebRequest
      Public ResponseStream As Stream
      ' Create Decoder for appropriate encoding type.
      Public StreamDecode As Decoder = Encoding.UTF8.GetDecoder()

      Public Sub New
         Request = Nothing
         ResponseStream = Nothing
      End Sub
End Class

' ClientGetAsync issues the async request.
Class ClientGetAsync
   Shared allDone As New ManualResetEvent(False)
      Const BUFFER_SIZE As Integer = 1024

    Shared Sub Main()
        Dim Args As String() = Environment.GetCommandLineArgs()

        If Args.Length < 2 Then
            ShowUsage()
            Return
        End If

      ' Get the URI from the command line.
        Dim HttpSite As Uri = New Uri(Args(1))

      ' Create the request object.
        Dim wreq As HttpWebRequest = _
           CType(WebRequest.Create(HttpSite), HttpWebRequest)

      ' Create the state object.
      Dim rs As RequestState = New RequestState()

      ' Put the request into the state so it can be passed around.
      rs.Request = wreq

      ' Issue the async request.
        Dim r As IAsyncResult = _
           CType(wreq.BeginGetResponse( _
           New AsyncCallback(AddressOf RespCallback), rs), IAsyncResult)
```

```
            New AsyncCallback(AddressOf RespCallback), rs), IAsyncResult)

        ' Wait until the ManualResetEvent is set so that the application
        ' does not exit until after the callback is called.
        allDone.WaitOne()
    End Sub

    Shared Sub ShowUsage()
        Console.WriteLine("Attempts to GET a URI")
        Console.WriteLine()
        Console.WriteLine("Usage:")
        Console.WriteLine("ClientGetAsync URI")
        Console.WriteLine("Examples:")
        Console.WriteLine("ClientGetAsync http://www.contoso.com/")
    End Sub

    Shared Sub RespCallback(ar As IAsyncResult)
       ' Get the RequestState object from the async result.
       Dim rs As RequestState = CType(ar.AsyncState, RequestState)

       ' Get the HttpWebRequest from RequestState.
       Dim req As HttpWebRequest= rs.Request

       ' Call EndGetResponse, which returns the HttpWebResponse object
       ' that came from the request issued above.
       Dim resp As HttpWebResponse = _
           CType(req.EndGetResponse(ar), HttpWebResponse)

       ' Start reading data from the respons stream.
       Dim ResponseStream As Stream = resp.GetResponseStream()

       ' Store the reponse stream in RequestState to read
       ' the stream asynchronously.
       rs.ResponseStream = ResponseStream

       ' Pass rs.BufferRead to BeginRead. Read data into rs.BufferRead.
       Dim iarRead As IAsyncResult = _
          ResponseStream.BeginRead(rs.BufferRead, 0, BUFFER_SIZE, _
          New AsyncCallback(AddressOf ReadCallBack), rs)
    End Sub

   Shared Sub ReadCallBack(asyncResult As IAsyncResult)
       ' Get the RequestState object from the AsyncResult.
      Dim rs As RequestState = CType(asyncResult.AsyncState, RequestState)

      ' Retrieve the ResponseStream that was set in RespCallback.
      Dim responseStream As Stream = rs.ResponseStream

      ' Read rs.BufferRead to verify that it contains data.
      Dim read As Integer = responseStream.EndRead( asyncResult )
      If read > 0 Then
         ' Prepare a Char array buffer for converting to Unicode.
         Dim charBuffer(1024) As Char

         ' Convert byte stream to Char array and then String.
         ' len contains the number of characters converted to Unicode.
         Dim len As Integer = _
           rs.StreamDecode.GetChars(rs.BufferRead, 0, read, charBuffer, 0)
         Dim str As String = new String(charBuffer, 0, len)

         ' Append the recently read data to the RequestData stringbuilder
         ' object contained in RequestState.
         rs.RequestData.Append( _
            Encoding.ASCII.GetString(rs.BufferRead, 0, read))

         ' Continue reading data until responseStream.EndRead
         ' returns –1.
         Dim ar As IAsyncResult = _
            responseStream.BeginRead(rs.BufferRead, 0, BUFFER_SIZE, _
```

```
                New AsyncCallback(AddressOf ReadCallBack), rs)
        Else
            If rs.RequestData.Length > 1 Then
              ' Display data to the console.
              Dim strContent As String
              strContent = rs.RequestData.ToString()
              Console.WriteLine(strContent)
           End If

           ' Close down the response stream.
           responseStream.Close()

           ' Set the ManualResetEvent so the main thread can exit.
           allDone.Set()
        End If

        Return
    End Sub
End Class
```

See Also

Requesting Data

Handling Errors

7/29/2017 • 3 min to read • Edit Online

The WebRequest and WebResponse classes throw both system exceptions (such as ArgumentException) and Web-specific exceptions (which are WebException thrown by the GetResponse method).

Each **WebException** includes a Status property that contains a value from the WebExceptionStatus enumeration. You can examine the **Status** property to determine the error that occurred and take the proper steps to resolve the error.

The following table describes the possible values for the **Status** property.

STATUS	DESCRIPTION
ConnectFailure	The remote service could not be contacted at the transport level.
ConnectionClosed	The connection was closed prematurely.
KeepAliveFailure	The server closed a connection made with the Keep-alive header set.
NameResolutionFailure	The name service could not resolve the host name.
ProtocolError	The response received from the server was complete but indicated an error at the protocol level.
ReceiveFailure	A complete response was not received from the remote server.
RequestCanceled	The request was canceled.
SecureChannelFailure	An error occurred in a secure channel link.
SendFailure	A complete request could not be sent to the remote server.
ServerProtocolViolation	The server response was not a valid HTTP response.
Success	No error was encountered.
Timeout	No response was received within the time-out set for the request.
TrustFailure	A server certificate could not be validated.
MessageLengthLimitExceeded	A message was received that exceeded the specified limit when sending a request or receiving a response from the server.
Pending	An internal asynchronous request is pending.
PipelineFailure	This value supports the .NET Framework infrastructure and is not intended to be used directly in your code.

STATUS	DESCRIPTION
ProxyNameResolutionFailure	The name resolver service could not resolve the proxy host name.
UnknownError	An exception of unknown type has occurred.

When the **Status** property is **WebExceptionStatus.ProtocolError**, a **WebResponse** that contains the response from the server is available. You can examine this response to determine the actual source of the protocol error.

The following example shows how to catch a **WebException**.

```
try
{
    // Create a request instance.
    WebRequest myRequest =
    WebRequest.Create("http://www.contoso.com");
    // Get the response.
    WebResponse myResponse = myRequest.GetResponse();
    //Get a readable stream from the server.
    Stream sr = myResponse.GetResponseStream();

    //Read from the stream and write any data to the console.
    bytesread = sr.Read( myBuffer, 0, length);
    while( bytesread > 0 )
    {
        for (int i=0; i<bytesread; i++) {
            Console.Write( "{0}", myBuffer[i]);
        }
        Console.WriteLine();
        bytesread = sr.Read( myBuffer, 0, length);
    }
    sr.Close();
    myResponse.Close();
}
catch (WebException webExcp)
{
    // If you reach this point, an exception has been caught.
    Console.WriteLine("A WebException has been caught.");
    // Write out the WebException message.
    Console.WriteLine(webExcp.ToString());
    // Get the WebException status code.
    WebExceptionStatus status =  webExcp.Status;
    // If status is WebExceptionStatus.ProtocolError,
    //   there has been a protocol error and a WebResponse
    //   should exist. Display the protocol error.
    if (status == WebExceptionStatus.ProtocolError) {
        Console.Write("The server returned protocol error ");
        // Get HttpWebResponse so that you can check the HTTP status code.
        HttpWebResponse httpResponse = (HttpWebResponse)webExcp.Response;
        Console.WriteLine((int)httpResponse.StatusCode + " - "
           + httpResponse.StatusCode);
    }
}
catch (Exception e)
{
    // Code to catch other exceptions goes here.
}
```

```
Try
    ' Create a request instance.
    Dim myRequest As WebRequest = WebRequest.Create("http://www.contoso.com")
    ' Get the response.
    Dim myResponse As WebResponse = myRequest.GetResponse()
    'Get a readable stream from the server.
    Dim sr As Stream = myResponse.GetResponseStream()

    Dim i As Integer
    'Read from the stream and write any data to the console.
    bytesread = sr.Read(myBuffer, 0, length)
    While bytesread > 0
        For i = 0 To bytesread - 1
            Console.Write("{0}", myBuffer(i))
        Next i
        Console.WriteLine()
        bytesread = sr.Read(myBuffer, 0, length)
    End While
    sr.Close()
    myResponse.Close()
Catch webExcp As WebException
    ' If you reach this point, an exception has been caught.
    Console.WriteLine("A WebException has been caught.")
    ' Write out the WebException message.
    Console.WriteLine(webExcp.ToString())
    ' Get the WebException status code.
    Dim status As WebExceptionStatus = webExcp.Status
    ' If status is WebExceptionStatus.ProtocolError,
    '   there has been a protocol error and a WebResponse
    '   should exist. Display the protocol error.
    If status = WebExceptionStatus.ProtocolError Then
        Console.Write("The server returned protocol error ")
        ' Get HttpWebResponse so that you can check the HTTP status code.
        Dim httpResponse As HttpWebResponse = _
           CType(webExcp.Response, HttpWebResponse)
        Console.WriteLine(CInt(httpResponse.StatusCode).ToString() & _
           " - " & httpResponse.StatusCode.ToString())
    End If
Catch e As Exception
    ' Code to catch other exceptions goes here.
End Try
```

Applications that use the Socket class throw SocketException when errors occur on the Windows socket. The TcpClient, TcpListener, and UdpClient classes are built on top of the **Socket** class and throw **SocketExceptions** as well.

When a **SocketException** is thrown, the **SocketException** class sets the ErrorCode property to the last operating system socket error that occurred. For more information about socket error codes, see the Winsock 2.0 API error code documentation in MSDN.

See Also

Exception Handling Fundamentals
Requesting Data

Programming Pluggable Protocols

7/29/2017 • 1 min to read • Edit Online

The abstract WebRequest and WebResponse classes provide the base for pluggable protocols. By deriving protocol-specific classes from WebRequest and WebResponse, an application can request data from an Internet resource and read the response without specifying the protocol being used.

Before you can create a protocol-specific WebRequest, you must register its Create method. Use the static RegisterPrefix(String, IWebRequestCreate) method of WebRequest to register a WebRequest descendant to handle a set of requests to a particular Internet scheme, to a scheme and server, or to a scheme, server, and path.

In most cases you will be able to send and receive data using the methods and properties of the WebRequest class. However, if you need to access protocol-specific properties, you can typecast a WebRequest to a specific derived-class instance.

To take advantage of pluggable protocols, your WebRequest descendants must provide a default request-and-response transaction that does not require protocol-specific properties to be set. For example, the HttpWebRequest class, which implements the WebRequest class for HTTP, provides a `GET` request by default and returns an HttpWebResponse that contains the stream returned from the Web server.

See Also

Deriving from WebRequest
Deriving from WebResponse
Network Programming in the .NET Framework
How to: Typecast a WebRequest to Access Protocol Specific Properties

How to: Register a Custom Protocol Using WebRequest

7/29/2017 • 1 min to read • Edit Online

This example shows how to register a protocol specific classthat is defined elsewhere. In this example, `CustomWebRequestCreator` is the user-implemented object that implements the **Create** method that returns the `CustomWebRequest` object. The code example assumes that you have written the `CustomWebRequest` code that implements the custom protocol.

Example

```
WebRequest.RegisterPrefix("custom", new CustomWebRequestCreator());
WebRequest req = WebRequest.Create("custom://customHost.contoso.com/");
```

```
WebRequest.RegisterPrefix("custom", New CustomWebRequestCreator())
Dim req As WebRequest = WebRequest.Create("custom://customHost.contoso.com/")
```

Compiling the Code

This example requires:

References to the System.Net namespace.

See Also

Programming Pluggable Protocols

How to: Typecast a WebRequest to Access Protocol Specific Properties

7/29/2017 • 1 min to read • Edit Online

This example shows how to typecast a WebRequest so that you can access protocol specific properties.

Example

```
HttpWebRequest httpreq =
   (HttpWebRequest) WebRequest.Create("http://www.contoso.com/");
```

```
Dim httpreq As HttpWebRequest = _
   CType(WebRequest.Create("http://www.contoso.com/"), HttpWebRequest)
```

See Also

Programming Pluggable Protocols

Deriving from WebRequest

7/29/2017 • 5 min to read • Edit Online

The WebRequest class is an abstract base class that provides the basic methods and properties for creating a protocol-specific request handler that fits the .NET Framework pluggable protocol model. Applications that use the **WebRequest** class can request data using any supported protocol without needing to specify the protocol used.

Two criteria must be met in order for a protocol-specific class to be used as a pluggable protocol: The class must implement the IWebRequestCreate interface, and it must register with the System.Net.WebRequest.RegisterPrefix method. The class must override all the abstract methods and properties of **WebRequest** to provide the pluggable interface.

WebRequest instances are intended for one-time use; if you want to make another request, create a new **WebRequest**. **WebRequest** supports the ISerializable interface to enable developers to serialize a template **WebRequest** and then reconstruct the template for additional requests.

IWebRequest Create Method

The Create method is responsible for initializing a new instance of the protocol-specific class. When a new **WebRequest** is created, the System.Net.WebRequest.Create method matches the requested URI with the URI prefixes registered with the **RegisterPrefix** method. The **Create** method of the proper protocol-specific descendant must return an initialized instance of the descendant capable of performing a standard request/response transaction for the protocol without needing any protocol-specific fields modified.

ConnectionGroupName Property

The ConnectionGroupName property is used to name a group of connections to a resource so that multiple requests can be made over a single connection. To implement connection-sharing, you must use a protocol-specific method of pooling and assigning connections. For example, the provided ServicePointManager class implements connection sharing for the HttpWebRequest class. The **ServicePointManager** class creates a ServicePoint that provides a connection to a specific server for each connection group.

ContentLength Property

The ContentLength property specifies the number of bytes of data that will be sent to the server when uploading data.

Typically the Method property must be set to indicate that an upload is taking place when the **ContentLength** property is set to a value greater than zero.

ContentType Property

The ContentType property provides any special information that your protocol requires you to send to the server to identify the type of content that you are sending. Typically this is the MIME content type of any data uploaded.

Credentials Property

The Credentials property contains information needed to authenticate the request with the server. You must implement the details of the authentication process for your protocol. The AuthenticationManager class is responsible for authenticating requests and providing an authentication token. The class that provides the credentials used by your protocol must implement the ICredentials interface.

Headers Property

The Headers property contains an arbitrary collection of name/value pairs of metadata associated with the request. Any metadata needed by the protocol that can be expressed as a name/value pair can be included in the **Headers** property. Typically this information must be set before calling the GetRequestStream or GetResponse methods; once the request has been made, the metadata is considered read-only.

You are not required to use the **Headers** property to use header metadata. Protocol-specific metadata can be exposed as properties; for example, the System.Net.HttpWebRequest.UserAgent property exposes the **User-Agent** HTTP header. When you expose header metadata as a property, you should not allow the same property to be set using the **Headers** property.

Method Property

The Method property contains the verb or action that the request is asking the server to perform. The default for the **Method** property must enable a standard request/response action without requiring any protocol-specific properties to be set. For example, the Method method defaults to GET, which requests a resource from a Web server and returns the response.

Typically the **ContentLength** property must be set to a value greater than zero when the **Method** property is set to a verb or action that indicates that an upload is taking place.

PreAuthenticate Property

Applications set the PreAuthenticate property to indicate that authentication information should be sent with the initial request rather than waiting for an authentication challenge. The **PreAuthenticate** property is only meaningful if the protocol supports authentication credentials sent with the initial request.

Proxy Property

The Proxy property contains an IWebProxy interface that is used to access the requested resource. The **Proxy** property is meaningful only if your protocol supports proxied requests. You must set the default proxy if one is required by your protocol.

In some environments, such as behind a corporate firewall, your protocol might be required to use a proxy. In that case, you must implement the **IWebProxy** interface to create a proxy class that will work for your protocol.

RequestUri Property

The RequestUri property contains the URI that was passed to the **WebRequest.Create** method. It is read-only and cannot be changed once the **WebRequest** has been created. If your protocol supports redirection, the response can come from a resource identified by a different URI. If you need to provide access to the URI that responded, you must provide an additional property containing that URI.

Timeout Property

The Timeout property contains the length of time, in milliseconds, to wait before the request times out and throws an exception. **Timeout** applies only to synchronous requests made with the GetResponse method; asynchronous requests must use the Abort method to cancel a pending request.

Setting the **Timeout** property is meaningful only if the protocol-specific class implements a time-out process.

Abort Method

The Abort method cancels a pending asynchronous request to a server. After the request has been canceled, calling

GetResponse, **BeginGetResponse**, **EndGetResponse**, **GetRequestStream**, **BeginGetRequestStream**, or **EndGetRequestStream** will throw a WebException with the Status property set to WebExceptionStatus.

BeginGetRequestStream and EndGetRequestStream Methods

The BeginGetRequestStream method starts an asynchronous request for the stream that is used to upload data to the server. The EndGetRequestStream method completes the asynchronous request and returns the requested stream. These methods implement the **GetRequestStream** method using the standard .NET Framework asynchronous pattern.

BeginGetResponse and EndGetResponse Methods

The BeginGetResponse method starts an asynchronous request to a server. The EndGetResponse method completes the asynchronous request and returns the requested response. These methods implement the **GetResponse** method using the standard .NET Framework asynchronous pattern.

GetRequestStream Method

The GetRequestStream method returns a stream that is used to write data to the requested server. The stream returned should be a write-only stream that does not seek; it is intended as a one-way stream of data that is written to the server. The stream returns false for the CanRead and CanSeek properties and true for the CanWrite property.

The **GetRequestStream** method typically opens a connection to the server and, before returning the stream, sends header information that indicates that data is being sent to the server. Because **GetRequestStream** begins the request, setting any **Header** properties or the **ContentLength** property is typically not allowed after calling **GetRequestStream**.

GetResponse Method

The GetResponse method returns a protocol-specific descendant of the WebResponse class that represents the response from the server. Unless the request has already been initiated by the **GetRequestStream** method, the **GetResponse** method creates a connection to the resource identified by **RequestUri**, sends header information indicating the type of request being made, and then receives the response from the resource.

Once the **GetResponse** method is called, all properties should be considered read-only. **WebRequest** instances are intended for one-time use; if you want to make another request, you should create a new **WebRequest**.

The **GetResponse** method is responsible for creating an appropriate **WebResponse** descendant to contain the incoming response.

See Also

WebRequest
HttpWebRequest
FileWebRequest
Programming Pluggable Protocols
Deriving from WebResponse

Deriving from WebResponse

7/29/2017 • 2 min to read • Edit Online

The WebResponse class is an abstract base class that provides the basic methods and properties for creating a protocol-specific response that fits the .NET Framework pluggable protocol model. Applications that use the WebRequest class to request data from resources receive the responses in a **WebResponse**. Protocol-specific **WebResponse** descendants must implement the abstract members of the **WebResponse** class.

The associated **WebRequest** class must create **WebResponse** descendants. For example, HttpWebResponse instances are created only as the result of calling System.Net.HttpWebRequest.GetResponse or System.Net.HttpWebRequest.EndGetResponse. Each **WebResponse** contains the result of a request to a resource and is not intended to be reused.

ContentLength Property

The ContentLength property indicates the number of bytes of data that are available from the stream returned by the GetResponseStream method. The **ContentLength** property does not indicate the number of bytes of header or metadata information returned by the server; it indicates only the number of bytes of data in the requested resource itself.

ContentType Property

The ContentType property provides any special information that your protocol requires you to send to the client to identify the type of content being sent by the server. Typically this is the MIME content type of any data returned.

Headers Property

The Headers property contains an arbitrary collection of name/value pairs of metadata associated with the response. Any metadata needed by the protocol that can be expressed as a name/value pair can be included in the **Headers** property.

You are not required to use the **Headers** property to use header metadata. Protocol-specific metadata can be exposed as properties; for example, the System.Net.HttpWebResponse.LastModified property exposes the **Last-Modified** HTTP header. When you expose header metadata as a property, you should not allow the same property to be set using the **Headers** property.

ResponseUri Property

The ResponseUri property contains the URI of the resource that actually provided the response. For protocols that do not support redirection, **ResponseUri** will be the same as the RequestUri property of the **WebRequest** that created the response. If the protocol supports redirecting the request, **ResponseUri** will contain the URI of the response.

Close Method

The Close method closes any connections made by the request and response and cleans up resources used by the response. The **Close** method closes any stream instances used by the response, but it does not throw an exception if the response stream was previously closed by a call to the System.IO.Stream.Close method.

GetResponseStream Method

The GetResponseStream method returns a stream containing the response from the requested resource. The response stream contains only the data returned by the resource; any header or metadata included in the response should be stripped from the response and exposed to the application through protocol-specific properties or the **Headers** property.

The stream instance returned by the **GetResponseStream** method is owned by the application and can be closed without closing the **WebResponse**. By convention, calling the **WebResponse.Close** method also closes the stream returned by **GetResponse**.

See Also

WebResponse
HttpWebResponse
FileWebResponse
Programming Pluggable Protocols
Deriving from WebRequest

Using Application Protocols

7/29/2017 • 1 min to read • Edit Online

The .NET Framework supports commonly used Internet application protocols. This section includes information on using the HTTP, "TCP", and "UDP" protocols, as well as information on using the Windows Sockets interface to implement custom protocols.

See Also

Network Programming in the .NET Framework
Network Programming Samples
Networking Samples for .NET on MSDN Code Gallery

HTTP

7/29/2017 • 1 min to read • Edit Online

The .NET Framework provides comprehensive support for the HTTP protocol, which makes up the majority of all Internet traffic, with theHttpWebRequest and HttpWebResponse classes. These classes, derived from WebRequest and WebResponse, are returned by default whenever the static method System.Net.WebRequest.Create encounters a URI beginning with "http" or "https". In most cases, the **WebRequest** and **WebResponse** classes provide all that is necessary to make the request, but if you need access to the HTTP-specific features exposed as properties, you can typecast these classes to **HttpWebRequest** or **HttpWebResponse**.

HttpWebRequest and **HttpWebResponse** encapsulate a standard HTTP request-and-response transaction and provide access to common HTTP headers. These classes also support most HTTP 1.1 features, including pipelining, sending and receiving data in chunks, authentication, preauthentication, encryption, proxy support, server certificate validation, and connection management. Custom headers and headers not provided through properties can be stored in and accessed through the **Headers** property.

HttpWebRequest is the default class used by **WebRequest** and does not need to be registered before you can pass a URI to the **WebRequest.Create** method.

You can make your application follow HTTP redirects automatically by setting the AllowAutoRedirect property to **true** (the default). The application will redirect requests, and the ResponseUri property of **HttpWebResponse** will contain the actual Web resource that responded to the request. If you set **AllowAutoRedirect** to **false**, your application must be able to handle redirects as HTTP protocol errors.

Applications receive HTTP protocol errors by catching a WebException with the Status set to WebExceptionStatus. The Response property contains the **WebResponse** sent by the server and indicates the actual HTTP error encountered.

See Also

Accessing the Internet Through a Proxy
Using Application Protocols
How to: Access HTTP-Specific Properties

HttpListener

7/29/2017 • 1 min to read • Edit Online

The HttpListener class provides a programmatically controlled HTTP protocol listener. The listener is active for the lifetime of the HttpListener object and runs within your application.

HTTP.SYS

The HttpListener class is built on top of HTTP.sys, which is the kernel mode listener that handles all HTTP traffic for Windows. HTTP.sys provides connection management, bandwidth throttling, and Web server logging. Use the `HttpCfg.exe` tool to add SSL certificates. For more information, see the documentation on the HttpCfg.exe tool in the Server documentation.

See Also

HttpListener
HttpWebRequest
HttpWebResponse
HTTP Server
Security Enhancements in Internet Information
HttpListener ASPX Host Application Sample
HttpListener Technology Sample
Network Programming Samples

How to: Access HTTP-Specific Properties

7/29/2017 • 1 min to read • Edit Online

This sample shows how to turn off the HTTP **Keep-alive** behavior and get the protocol version number from the Web server.

Example

```
Dim HttpWReq As HttpWebRequest= _
    CType(WebRequest.Create("http://www.contoso.com"), HttpWebRequest)
' Turn off connection keep-alives.
HttpWReq.KeepAlive = False

Dim HttpWResp As HttpWebResponse = _
    CType(HttpWReq.GetResponse(), HttpWebResponse)

' Get the HTTP protocol version number returned by the server.
Dim ver As String = HttpWResp.ProtocolVersion.ToString()
HttpWResp.Close()
```

```
HttpWebRequest HttpWReq =
    (HttpWebRequest)WebRequest.Create("http://www.contoso.com");
// Turn off connection keep-alives.
HttpWReq.KeepAlive = false;

HttpWebResponse HttpWResp = (HttpWebResponse)HttpWReq.GetResponse();

// Get the HTTP protocol version number returned by the server.
String ver = HttpWResp.ProtocolVersion.ToString();
HttpWResp.Close();
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

See Also

Accessing the Internet Through a Proxy
Using Application Protocols
HTTP

Managing Connections

7/29/2017 • 2 min to read • Edit Online

Applications that use HTTP to connect to data resources can use the .NET Framework's ServicePoint and ServicePointManager classes to manage connections to the Internet and to help them achieve optimum scale and performance.

The **ServicePoint** class provides an application with an endpoint to which the application can connect to access Internet resources. Each **ServicePoint** contains information that helps optimize connections with an Internet server by sharing optimization information between connections to improve performance.

Each **ServicePoint** is identified by a Uniform Resource Identifier (URI) and is categorized according to the scheme identifier and host fragments of the URI. For example, the same **ServicePoint** instance would provide requests to the URIs http://www.contoso.com/index.htm and http://www.contoso.com/news.htm?date=today since they have the same scheme identifier (http) and host fragments (www.contoso.com). If the application already has a persistent connection to the server www.contoso.com, it uses that connection to retrieve both requests, avoiding the need to create two connections.

ServicePointManager is a static class that manages the creation and destruction of **ServicePoint** instances. The **ServicePointManager** creates a **ServicePoint** when the application requests an Internet resource that is not in the collection of existing **ServicePoint** instances. **ServicePoint** instances are destroyed when they have exceeded their maximum idle time or when the number of existing **ServicePoint** instances exceeds the maximum number of **ServicePoint** instances for the application. You can control both the default maximum idle time and the maximum number of **ServicePoint** instances by setting the MaxServicePointIdleTime and MaxServicePoints properties on the **ServicePointManager**.

The number of connections between a client and server can have a dramatic impact on application throughput. By default, an application using the HttpWebRequest class uses a maximum of two persistent connections to a given server, but you can set the maximum number of connections on a per-application basis.

> **NOTE**
>
> The HTTP/1.1 specification limits the number of connections from an application to two connections per server.

The optimum number of connections depends on the actual conditions in which the application runs. Increasing the number of connections available to the application may not affect application performance. To determine the impact of more connections, run performance tests while varying the number of connections. You can change the number of connections that an application uses by changing the static DefaultConnectionLimit property on the **ServicePointManager** class at application initialization, as shown in the following code sample.

```
// Set the maximum number of connections per server to 4.
ServicePointManager.DefaultConnectionLimit = 4;
```

```
' Set the maximum number of connections per server to 4.
ServicePointManager.DefaultConnectionLimit = 4
```

Changing the **ServicePointManager.DefaultConnectionLimit** property does not affect previously initialized **ServicePoint** instances. The following code demonstrates changing the connection limit on an existing **ServicePoint** for the server http://www.contoso.com to the value stored in `newLimit`.

```
Uri uri = new Uri("http://www.contoso.com/");
ServicePoint sp = ServicePointManager.FindServicePoint(uri);
sp.ConnectionLimit = newLimit;
```

```
Dim uri As New Uri("http://www.contoso.com/")
Dim sp As ServicePoint = ServicePointManager.FindServicePoint(uri)
sp.ConnectionLimit = newLimit
```

See Also

Connection Grouping
Using Application Protocols

Connection Grouping

7/29/2017 • 1 min to read • Edit Online

Connection grouping associates specific requests within a single application to a defined connection pool. This can be required by a middle-tier application that connects to a back-end server on behalf of a user and uses an authentication protocol that supports delegation, such as Kerberos, or by a middle-tier application that supplies its own credentials, as in the example below. For example, suppose a user, Joe, visits an internal Web site that displays his payroll information. After authenticating Joe, the middle-tier application server uses Joe's credentials to connect to the back-end server to retrieve his payroll information. Next, Susan visits the site and requests her payroll information. Because the middle-tier application has already made a connection using Joe's credentials, the back-end server responds with Joe's information. However, if the application assigns each request sent to the back-end server to a connection group formed from the user name, then each user belongs to a separate connection pool and cannot accidentally share authentication information with another user.

To assign a request to a specific connection group, you must assign a name to the ConnectionGroupName property of your WebRequest before making the request.

See Also

Managing Connections
How to: Assign User Information to Group Connections

How to: Assign User Information to Group Connections

7/29/2017 • 1 min to read • Edit Online

The following example demonstrates how to assign user information to group connections, assuming that the application sets the variables *UserName*, *SecurelyStoredPassword*, and *Domain* before this section of code is called and that *UserName* is unique.

To assign user information to a group connection

1. Create a connection group name.

```
SHA1Managed Sha1 = new SHA1Managed();
Byte[] updHash = Sha1.ComputeHash(Encoding.UTF8.GetBytes(UserName + SecurelyStoredPassword + Domain));
String secureGroupName = Encoding.Default.GetString(updHash);
```

```
Dim Sha1 As New SHA1Managed()
Dim updHash As [Byte]() = Sha1.ComputeHash(Encoding.UTF8.GetBytes((UserName + SecurelyStoredPassword +
Domain)))
Dim secureGroupName As [String] = Encoding.Default.GetString(updHash)
```

2. Create a request for a specific URL. For example, the following code creates a request for the URL `http://www.contoso.com.`

```
WebRequest myWebRequest=WebRequest.Create("http://www.contoso.com");
```

```
Dim myWebRequest As WebRequest = WebRequest.Create("http://www.contoso.com")
```

3. Set the credentials and Connection GroupName for the Web request, and call **GetResponse** to retrieve a **WebResponse** object.

```
myWebRequest.Credentials = new NetworkCredential(UserName, SecurelyStoredPassword, Domain);
myWebRequest.ConnectionGroupName = secureGroupName;

WebResponse myWebResponse=myWebRequest.GetResponse();
```

```
myWebRequest.Credentials = New NetworkCredential(UserName, SecurelyStoredPassword, Domain)
myWebRequest.ConnectionGroupName = secureGroupName

Dim myWebResponse As WebResponse = myWebRequest.GetResponse()
```

4. Close the response stream after using the WebRespose object.

```
MyWebResponse.Close();
```

```
MyWebResponse.Close()
```

Example

```
// Create a connection group name.
SHA1Managed Sha1 = new SHA1Managed();
Byte[] updHash = Sha1.ComputeHash(Encoding.UTF8.GetBytes(UserName + SecurelyStoredPassword + Domain));
String secureGroupName = Encoding.Default.GetString(updHash);

// Create a request for a specific URL.
WebRequest myWebRequest=WebRequest.Create("http://www.contoso.com");

myWebRequest.Credentials = new NetworkCredential(UserName, SecurelyStoredPassword, Domain);
myWebRequest.ConnectionGroupName = secureGroupName;

WebResponse myWebResponse=myWebRequest.GetResponse();

// Insert the code that uses myWebResponse.

MyWebResponse.Close();
```

```
' Create a secure group name.
Dim Sha1 As New SHA1Managed()
Dim updHash As [Byte]() = Sha1.ComputeHash(Encoding.UTF8.GetBytes((UserName + SecurelyStoredPassword +
Domain)))
Dim secureGroupName As [String] = Encoding.Default.GetString(updHash)

' Create a request for a specific URL.
Dim myWebRequest As WebRequest = WebRequest.Create("http://www.contoso.com")

myWebRequest.Credentials = New NetworkCredential(UserName, SecurelyStoredPassword, Domain)
myWebRequest.ConnectionGroupName = secureGroupName

Dim myWebResponse As WebResponse = myWebRequest.GetResponse()

' Insert the code that uses myWebResponse.
MyWebResponse.Close()
```

See Also

Managing Connections
Connection Grouping

TCP-UDP

7/29/2017 • 1 min to read • Edit Online

Applications can use Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) services with the TcpClient, TcpListener, and UdpClient classes. These protocol classes are built on top of the System.Net.Sockets.Socket class and take care of the details of transferring data.

The protocol classes use the synchronous methods of the **Socket** class to provide simple and straightforward access to network services without the overhead of maintaining state information or knowing the details of setting up protocol-specific sockets. To use asynchronous **Socket** methods, you can use the asynchronous methods supplied by the NetworkStream class. To access features of the **Socket** class not exposed by the protocol classes, you must use the **Socket** class.

TcpClient and **TcpListener** represent the network using the **NetworkStream** class. You use the GetStream method to return the network stream, and then call the stream's Read and Write methods. The **NetworkStream** does not own the protocol classes' underlying socket, so closing it does not affect the socket.

The **UdpClient** class uses an array of bytes to hold the UDP datagram. You use the Send method to send the data to the network and the Receive method to receive an incoming datagram.

See Also

Using TCP Services
Using UDP Services
Using Streams on the Network
Using an Asynchronous Server Socket
Using an Asynchronous Client Socket
Using Application Protocols

Using TCP Services

7/29/2017 • 3 min to read • Edit Online

The TcpClient class requests data from an Internet resource using TCP. The methods and properties of **TcpClient** abstract the details for creating a Socket for requesting and receiving data using TCP. Because the connection to the remote device is represented as a stream, data can be read and written with .NET Framework stream-handling techniques.

The TCP protocol establishes a connection with a remote endpoint and then uses that connection to send and receive data packets. TCP is responsible for ensuring that data packets are sent to the endpoint and assembled in the correct order when they arrive.

To establish a TCP connection, you must know the address of the network device hosting the service you need and you must know the TCP port that the service uses to communicate. The Internet Assigned Numbers Authority (Iana) defines port numbers for common services (see www.iana.org/assignments/port-numbers). Services not on the Iana list can have port numbers in the range 1,024 to 65,535.

The following example demonstrates setting up a **TcpClient** to connect to a time server on TCP port 13.

```
Imports System
Imports System.Net.Sockets
Imports System.Text

Public Class TcpTimeClient
    Private const portNum As Integer = 13
    Private const hostName As String = "host.contoso.com"

    ' Entry point  that delegates to C-style main Private Function.
    Public Overloads Shared Sub Main()
        System.Environment.ExitCode = _
            Main(System.Environment.GetCommandLineArgs())
    End Sub

    Overloads Public Shared Function Main(args() As [String]) As Integer
        Try
            Dim client As New TcpClient(hostName, portNum)

            Dim ns As NetworkStream = client.GetStream()

            Dim bytes(1024) As Byte
            Dim bytesRead As Integer = ns.Read(bytes, 0, bytes.Length)

            Console.WriteLine(Encoding.ASCII.GetString(bytes, 0, bytesRead))

        Catch e As Exception
            Console.WriteLine(e.ToString())
        End Try

        client.Close()

        Return 0
    End Function 'Main
End Class 'TcpTimeClient
```

```
using System;
using System.Net.Sockets;
using System.Text;

public class TcpTimeClient {
    private const int portNum = 13;
    private const string hostName = "host.contoso.com";

    public static int Main(String[] args) {
        try {
            TcpClient client = new TcpClient(hostName, portNum);

            NetworkStream ns = client.GetStream();

            byte[] bytes = new byte[1024];
            int bytesRead = ns.Read(bytes, 0, bytes.Length);

            Console.WriteLine(Encoding.ASCII.GetString(bytes,0,bytesRead));

            client.Close();

        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }

        return 0;
    }
}
```

TcpListener is used to monitor a TCP port for incoming requests and then create either a **Socket** or a **TcpClient** that manages the connection to the client. The Start method enables listening, and the Stop method disables listening on the port. The AcceptTcpClient method accepts incoming connection requests and creates a **TcpClient** to handle the request, and the AcceptSocket method accepts incoming connection requests and creates a **Socket** to handle the request.

The following example demonstrates creating a network time server using a **TcpListener** to monitor TCP port 13. When an incoming connection request is accepted, the time server responds with the current date and time from the host server.

```
Imports System
Imports System.Net.Sockets
Imports System.Text

Public Class TcpTimeServer

    Private const portNum As Integer = 13

    ' Entry point that delegates to C-style main Private Function.
    Public Overloads Shared Sub Main()
        System.Environment.ExitCode = _
            Main(System.Environment.GetCommandLineArgs())
    End Sub

    Overloads Public Shared Function Main(args() As [String]) As Integer
        Dim done As Boolean = False

        Dim listener As New TcpListener(portNum)

        listener.Start()

        While Not done
            Console.Write("Waiting for connection...")
            Dim client As TcpClient = listener.AcceptTcpClient()

            Console.WriteLine("Connection accepted.")
            Dim ns As NetworkStream = client.GetStream()

            Dim byteTime As Byte() = _
                Encoding.ASCII.GetBytes(DateTime.Now.ToString())

            Try
                ns.Write(byteTime, 0, byteTime.Length)
                ns.Close()
                client.Close()
            Catch e As Exception
                Console.WriteLine(e.ToString())
            End Try
        End While

        listener.Stop()

        Return 0
    End Function 'Main
End Class 'TcpTimeServer
```

```
using System;
using System.Net.Sockets;
using System.Text;

public class TcpTimeServer {

    private const int portNum = 13;

    public static int Main(String[] args) {
        bool done = false;

        TcpListener listener = new TcpListener(portNum);

        listener.Start();

        while (!done) {
            Console.Write("Waiting for connection...");
            TcpClient client = listener.AcceptTcpClient();

            Console.WriteLine("Connection accepted.");
            NetworkStream ns = client.GetStream();

            byte[] byteTime = Encoding.ASCII.GetBytes(DateTime.Now.ToString());

            try {
                ns.Write(byteTime, 0, byteTime.Length);
                ns.Close();
                client.Close();
            } catch (Exception e) {
                Console.WriteLine(e.ToString());
            }
        }

        listener.Stop();

        return 0;
    }

}
```

See Also

Using UDP Services

7/29/2017 • 3 min to read • Edit Online

The UdpClient class communicates with network services using UDP. The properties and methods of the UdpClient class abstract the details of creating a Socket for requesting and receiving data using UDP.

User Datagram Protocol (UDP) is a simple protocol that makes a best effort to deliver data to a remote host. However, because the UDP protocol is a connectionless protocol, UDP datagrams sent to the remote endpoint are not guaranteed to arrive, nor are they guaranteed to arrive in the same sequence in which they are sent. Applications that use UDP must be prepared to handle missing, duplicate, and out-of-sequence datagrams.

To send a datagram using UDP, you must know the network address of the network device hosting the service you need and the UDP port number that the service uses to communicate. The Internet Assigned Numbers Authority (Iana) defines port numbers for common services (see www.iana.org/assignments/port-numbers). Services not on the Iana list can have port numbers in the range 1,024 to 65,535.

Special network addresses are used to support UDP broadcast messages on IP-based networks. The following discussion uses the IP version 4 address family used on the Internet as an example.

IP version 4 addresses use 32 bits to specify a network address. For class C addresses using a netmask of 255.255.255.0, these bits are separated into four octets. When expressed in decimal, the four octets form the familiar dotted-quad notation, such as 192.168.100.2. The first two octets (192.168 in this example) form the network number, the third octet (100) defines the subnet, and the final octet (2) is the host identifier.

Setting all the bits of an IP address to one, or 255.255.255.255, forms the limited broadcast address. Sending a UDP datagram to this address delivers the message to any host on the local network segment. Because routers never forward messages sent to this address, only hosts on the network segment receive the broadcast message.

Broadcasts can be directed to specific portions of a network by setting all bits of the host identifier. For example, to send a broadcast to all hosts on the network identified by IP addresses starting with 192.168.1, use the address 192.168.1.255.

The following code example uses a UdpClient to listen for UDP datagrams sent to the directed broadcast address 192.168.1.255 on port 11,000. The client receives a message string and writes the message to the console.

```
Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Text

Public Class UDPListener
   Private Const listenPort As Integer = 11000

   Private Shared Sub StartListener()
      Dim done As Boolean = False
      Dim listener As New UdpClient(listenPort)
      Dim groupEP As New IPEndPoint(IPAddress.Any, listenPort)
      Try
         While Not done
            Console.WriteLine("Waiting for broadcast")
            Dim bytes As Byte() = listener.Receive(groupEP)
            Console.WriteLine("Received broadcast from {0} :", _
                groupEP.ToString())
            Console.WriteLine( _
                Encoding.ASCII.GetString(bytes, 0, bytes.Length))
            Console.WriteLine()
         End While
      Catch e As Exception
         Console.WriteLine(e.ToString())
      Finally
         listener.Close()
      End Try
   End Sub 'StartListener

   Public Shared Function Main() As Integer
      StartListener()
      Return 0
   End Function 'Main
End Class 'UDPListener
```

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

public class UDPListener
{
    private const int listenPort = 11000;

    private static void StartListener()
    {
        bool done = false;

        UdpClient listener = new UdpClient(listenPort);
        IPEndPoint groupEP = new IPEndPoint(IPAddress.Any,listenPort);

        try
        {
            while (!done)
            {
                Console.WriteLine("Waiting for broadcast");
                byte[] bytes = listener.Receive( ref groupEP);

                Console.WriteLine("Received broadcast from {0} :\n {1}\n",
                    groupEP.ToString(),
                    Encoding.ASCII.GetString(bytes,0,bytes.Length));
            }

        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }
        finally
        {
            listener.Close();
        }
    }

    public static int Main()
    {
        StartListener();

        return 0;
    }
}
```

The following code example uses a UdpClient to send UDP datagrams to the directed broadcast address 192.168.1.255, using port 11,000. The client sends the message string specified on the command line.

```
Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Text

Public Class Program

    Overloads Public Shared Function Main(args() As [String]) As Integer
      Dim s As New Socket(AddressFamily.InterNetwork, SocketType.Dgram,
          ProtocolType.Udp)
      Dim broadcast As IPAddress = IPAddress.Parse("192.168.1.255")
      Dim sendbuf As Byte() = Encoding.ASCII.GetBytes(args(0))
      Dim ep As New IPEndPoint(broadcast, 11000)
      s.SendTo(sendbuf, ep)
      Console.WriteLine("Message sent to the broadcast address")
   End Function 'Main
End Class
```

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
            ProtocolType.Udp);

        IPAddress broadcast = IPAddress.Parse("192.168.1.255");

        byte[] sendbuf = Encoding.ASCII.GetBytes(args[0]);
        IPEndPoint ep = new IPEndPoint(broadcast, 11000);

        s.SendTo(sendbuf, ep);

        Console.WriteLine("Message sent to the broadcast address");
    }
}
```

See Also

UdpClient
IPAddress

Sockets

7/29/2017 • 1 min to read • Edit Online

The System.Net.Sockets namespace contains a managed implementation of the Windows Sockets interface. All other network-access classes in the System.Net namespace are built on top of this implementation of sockets.

The .NET Framework Socket class is a managed-code version of the socket services provided by the Winsock32 API. In most cases, the **Socket** class methods simply marshal data into their native Win32 counterparts and handle any necessary security checks.

The **Socket** class supports two basic modes, synchronous and asynchronous. In synchronous mode, calls to functions that perform network operations (such as Send and Receive) wait until the operation completes before returning control to the calling program. In asynchronous mode, these calls return immediately.

See Also

How to: Create a Socket

Using Application Protocols

How to: Create a Socket

7/29/2017 • 1 min to read • Edit Online

Before you can use a socket to communicate with remote devices, the socket must be initialized with protocol and network address information. The constructor for the Socket class has parameters that specify the address family, socket type, and protocol type that the socket uses to make connections.

Example

The following example creates a Socket that can be used to communicate on a TCP/IP-based network, such as the Internet.

```
Socket s = new Socket(AddressFamily.InterNetwork,
   SocketType.Stream, ProtocolType.Tcp);
```

```
Dim s as New Socket(AddressFamily.InterNetwork, _
   SocketType.Stream, ProtocolType.Tcp)
```

To use UDP instead of TCP, change the protocol type, as in the following example:

```
Socket s = new Socket(AddressFamily.InterNetwork,
   SocketType.Dgram, ProtocolType.Udp);
```

```
Dim s as New Socket(AddressFamily.InterNetwork, _
   SocketType.Dgram, ProtocolType.Udp)
```

The AddressFamily enumeration specifies the standard address families used by the **Socket** class to resolve network addresses (for example, the **AddressFamily.InterNetwork** member specifies the IP version 4 address family).

The SocketType enumeration specifies the type of socket (for example, the **SocketType.Stream** member indicates a standard socket for sending and receiving data with flow control).

The ProtocolType enumeration specifies the network protocol to use when communicating on the **Socket** (for example, **ProtocolType.Tcp** indicates that the socket uses TCP; **ProtocolType.Udp** indicates that the socket uses UDP).

After a **Socket** is created, it can either initiate a connection to a remote endpoint or receive connections from remote devices.

See Also

Using Client Sockets
Listening with Sockets

Using Client Sockets

7/29/2017 • 1 min to read • Edit Online

Before you can initiate a conversation through a Socket, you must create a data pipe between your application and the remote device. Although other network address families and protocols exist, this example shows how to create a TCP/IP connection to a remote service.

TCP/IP uses a network address and a service port number to uniquely identify a service. The network address identifies a specific device on the network; the port number identifies the specific service on that device to connect to. The combination of network address and service port is called an endpoint, which is represented in the .NET Framework by the EndPoint class. A descendant of **EndPoint** is defined for each supported address family; for the IP address family, the class is IPEndPoint.

The Dns class provides domain-name services to applications that use TCP/IP Internet services. The Resolve method queries a DNS server to map a user-friendly domain name (such as "host.contoso.com") to a numeric Internet address (such as 192.168.1.1). **Resolve** returns an IPHostEntry that contains a list of addresses and aliases for the requested name. In most cases, you can use the first address returned in the AddressList array. The following code gets an IPAddress containing the IP address for the server host.contoso.com.

```
Dim ipHostInfo As IPHostEntry = Dns.Resolve("host.contoso.com")
Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
```

```
IPHostEntry ipHostInfo = Dns.Resolve("host.contoso.com");
IPAddress ipAddress = ipHostInfo.AddressList[0];
```

The Internet Assigned Numbers Authority (Iana) defines port numbers for common services (for more information, see www.iana.org/assignments/port-numbers). Other services can have registered port numbers in the range 1,024 to 65,535. The following code combines the IP address for host.contoso.com with a port number to create a remote endpoint for a connection.

```
Dim ipe As New IPEndPoint(ipAddress, 11000)
```

```
IPEndPoint ipe = new IPEndPoint(ipAddress,11000);
```

After determining the address of the remote device and choosing a port to use for the connection, the application can attempt to establish a connection with the remote device. The following example uses an existing **IPEndPoint** to connect to a remote device and catches any exceptions that are thrown.

```
Try
    s.Connect(ipe)
Catch ae As ArgumentNullException
    Console.WriteLine("ArgumentNullException : {0}", _
        ae.ToString())
Catch se As SocketException
    Console.WriteLine("SocketException : {0}", se.ToString())
Catch e As Exception
    Console.WriteLine("Unexpected exception : {0}", e.ToString())
End Try
```

```
try {
    s.Connect(ipe);
} catch(ArgumentNullException ae) {
    Console.WriteLine("ArgumentNullException : {0}", ae.ToString());
} catch(SocketException se) {
    Console.WriteLine("SocketException : {0}", se.ToString());
} catch(Exception e) {
    Console.WriteLine("Unexpected exception : {0}", e.ToString());
}
```

See Also

Using a Synchronous Client Socket
Using an Asynchronous Client Socket
How to: Create a Socket
Sockets

Using a Synchronous Client Socket

7/29/2017 • 1 min to read • Edit Online

A synchronous client socket suspends the application program while the network operation completes. Synchronous sockets are not suitable for applications that make heavy use of the network for their operation, but they can enable simple access to network services for other applications.

To send data, pass a byte array to one of the Socket class's send-data methods (Send and SendTo). The following example encodes a string into a byte array buffer using the System.Text.Encoding.ASCII property and then transmits the buffer to the network device using the **Send** method. The **Send** method returns the number of bytes sent to the network device.

```
Dim msg As Byte() = _
    System.Text.Encoding.ASCII.GetBytes("This is a test.")
Dim bytesSent As Integer = s.Send(msg)
```

```
byte[] msg = System.Text.Encoding.ASCII.GetBytes("This is a test");
int bytesSent = s.Send(msg);
```

The **Send** method removes the bytes from the buffer and queues them with the network interface to be sent to the network device. The network interface might not send the data immediately, but it will send it eventually, as long as the connection is closed normally with the Shutdown method.

To receive data from a network device, pass a buffer to one of the **Socket** class's receive-data methods (Receive and ReceiveFrom). Synchronous sockets will suspend the application until bytes are received from the network or until the socket is closed. The following example receives data from the network and then displays it on the console. The example assumes that the data coming from the network is ASCII-encoded text. The **Receive** method returns the number of bytes received from the network.

```
Dim bytes(1024) As Byte
Dim bytesRec = s.Receive(bytes)
Console.WriteLine("Echoed text = {0}", _
    System.Text.Encoding.ASCII.GetString(bytes, 0, bytesRec))
```

```
byte[] bytes = new byte[1024];
int bytesRec = s.Receive(bytes);
Console.WriteLine("Echoed text = {0}",
    System.Text.Encoding.ASCII.GetString(bytes, 0, bytesRec));
```

When the socket is no longer needed, you need to release it by calling the Shutdown method and then calling the **Close** method. The following example releases a **Socket**. The SocketShutdown enumeration defines constants that indicate whether the socket should be closed for sending, for receiving, or for both.

```
s.Shutdown(SocketShutdown.Both)
s.Close()
```

```
s.Shutdown(SocketShutdown.Both);
s.Close();
```

See Also

Using an Asynchronous Client Socket
Listening with Sockets
Synchronous Client Socket Example

Using an Asynchronous Client Socket

7/29/2017 • 7 min to read • Edit Online

An asynchronous client socket does not suspend the application while waiting for network operations to complete. Instead, it uses the standard .NET Framework asynchronous programming model to process the network connection on one thread while the application continues to run on the original thread. Asynchronous sockets are appropriate for applications that make heavy use of the network or that cannot wait for network operations to complete before continuing.

The Socket class follows the .NET Framework naming pattern for asynchronous methods; for example, the synchronous Receive method corresponds to the asynchronous BeginReceive and EndReceive methods.

Asynchronous operations require a callback method to return the result of the operation. If your application does not need to know the result, then no callback method is required. The example code in this section demonstrates using a method to start connecting to a network device and a callback method to complete the connection, a method to start sending data and a callback method to complete the send, and a method to start receiving data and a callback method to end receiving data.

Asynchronous sockets use multiple threads from the system thread pool to process network connections. One thread is responsible for initiating the sending or receiving of data; other threads complete the connection to the network device and send or receive the data. In the following examples, instances of the System.Threading.ManualResetEvent class are used to suspend execution of the main thread and signal when execution can continue.

In the following example, to connect an asynchronous socket to a network device, the `Connect` method initializes a **Socket** and then calls the System.Net.Sockets.Socket.Connect method, passing a remote endpoint that represents the network device, the connect callback method, and a state object (the client **Socket**), which is used to pass state information between asynchronous calls. The example implements the `Connect` method to connect the specified **Socket** to the specified endpoint. It assumes a global **ManualResetEvent** named `connectDone`.

```
Public Shared Sub Connect(remoteEP As EndPoint, client As Socket)
    client.BeginConnect(remoteEP, _
       AddressOf ConnectCallback, client)

    connectDone.WaitOne()
End Sub 'Connect
```

```
public static void Connect(EndPoint remoteEP, Socket client) {
    client.BeginConnect(remoteEP,
        new AsyncCallback(ConnectCallback), client );

   connectDone.WaitOne();
}
```

The connect callback method `ConnectCallback` implements the AsyncCallback delegate. It connects to the remote device when the remote device is available and then signals the application thread that the connection is complete by setting the **ManualResetEvent** `connectDone`. The following code implements the `ConnectCallback` method.

```
Private Shared Sub ConnectCallback(ar As IAsyncResult)
    Try
        ' Retrieve the socket from the state object.
        Dim client As Socket = CType(ar.AsyncState, Socket)

        ' Complete the connection.
        client.EndConnect(ar)

        Console.WriteLine("Socket connected to {0}", _
            client.RemoteEndPoint.ToString())

        ' Signal that the connection has been made.
        connectDone.Set()
    Catch e As Exception
        Console.WriteLine(e.ToString())
    End Try
End Sub 'ConnectCallback
```

```
private static void ConnectCallback(IAsyncResult ar) {
    try {
        // Retrieve the socket from the state object.
        Socket client = (Socket) ar.AsyncState;

        // Complete the connection.
        client.EndConnect(ar);

        Console.WriteLine("Socket connected to {0}",
            client.RemoteEndPoint.ToString());

        // Signal that the connection has been made.
        connectDone.Set();
    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }
}
```

The example method `Send` encodes the specified string data in ASCII format and sends it asynchronously to the network device represented by the specified socket. The following example implements the `Send` method.

```
Private Shared Sub Send(client As Socket, data As [String])
    ' Convert the string data to byte data using ASCII encoding.
    Dim byteData As Byte() = Encoding.ASCII.GetBytes(data)

    ' Begin sending the data to the remote device.
    client.BeginSend(byteData, 0, byteData.Length, SocketFlags.None, _
        AddressOf SendCallback, client)
End Sub 'Send
```

```
private static void Send(Socket client, String data) {
    // Convert the string data to byte data using ASCII encoding.
    byte[] byteData = Encoding.ASCII.GetBytes(data);

    // Begin sending the data to the remote device.
    client.BeginSend(byteData, 0, byteData.Length, SocketFlags.None,
        new AsyncCallback(SendCallback), client);
}
```

The send callback method `SendCallback` implements the AsyncCallback delegate. It sends the data when the network device is ready to receive. The following example shows the implementation of the `SendCallback` method. It assumes a global **ManualResetEvent** named `sendDone`.

```
Private Shared Sub SendCallback(ar As IAsyncResult)
    Try
        ' Retrieve the socket from the state object.
        Dim client As Socket = CType(ar.AsyncState, Socket)

        ' Complete sending the data to the remote device.
        Dim bytesSent As Integer = client.EndSend(ar)
        Console.WriteLine("Sent {0} bytes to server.", bytesSent)

        ' Signal that all bytes have been sent.
        sendDone.Set()
    Catch e As Exception
        Console.WriteLine(e.ToString())
    End Try
End Sub 'SendCallback
```

```
private static void SendCallback(IAsyncResult ar) {
    try {
        // Retrieve the socket from the state object.
        Socket client = (Socket) ar.AsyncState;

        // Complete sending the data to the remote device.
        int bytesSent = client.EndSend(ar);
        Console.WriteLine("Sent {0} bytes to server.", bytesSent);

        // Signal that all bytes have been sent.
        sendDone.Set();
    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }
}
```

Reading data from a client socket requires a state object that passes values between asynchronous calls. The following class is an example state object for receiving data from a client socket. It contains a field for the client socket, a buffer for the received data, and a StringBuilder to hold the incoming data string. Placing these fields in the state object allows their values to be preserved across multiple calls to read data from the client socket.

```
Public Class StateObject
    ' Client socket.
    Public workSocket As Socket = Nothing
    ' Size of receive buffer.
    Public BufferSize As Integer = 256
    ' Receive buffer.
    Public buffer(256) As Byte
    ' Received data string.
    Public sb As New StringBuilder()
End Class 'StateObject
```

```
public class StateObject {
    // Client socket.
    public Socket workSocket = null;
    // Size of receive buffer.
    public const int BufferSize = 256;
    // Receive buffer.
    public byte[] buffer = new byte[BufferSize];
    // Received data string.
    public StringBuilder sb = new StringBuilder();
}
```

The example `Receive` method sets up the state object and then calls the **BeginReceive** method to read the data

from the client socket asynchronously. The following example implements the `Receive` method.

```
Private Shared Sub Receive(client As Socket)
    Try
        ' Create the state object.
        Dim state As New StateObject()
        state.workSocket = client

        ' Begin receiving the data from the remote device.
        client.BeginReceive(state.buffer, 0, state.BufferSize, 0, _
            AddressOf ReceiveCallback, state)
    Catch e As Exception
        Console.WriteLine(e.ToString())
    End Try
End Sub 'Receive
```

```
private static void Receive(Socket client) {
    try {
        // Create the state object.
        StateObject state = new StateObject();
        state.workSocket = client;

        // Begin receiving the data from the remote device.
        client.BeginReceive( state.buffer, 0, StateObject.BufferSize, 0,
            new AsyncCallback(ReceiveCallback), state);
    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }
}
```

The receive callback method `ReceiveCallback` implements the **AsyncCallback** delegate. It receives the data from the network device and builds a message string. It reads one or more bytes of data from the network into the data buffer and then calls the **BeginReceive** method again until the data sent by the client is complete. Once all the data is read from the client, `ReceiveCallback` signals the application thread that the data is complete by setting the **ManualResetEvent** `sendDone`.

The following example code implements the `ReceiveCallback` method. It assumes a global string named `response` that holds the received string and a global **ManualResetEvent** named `receiveDone`. The server must shut down the client socket gracefully to end the network session.

```
Private Shared Sub ReceiveCallback(ar As IAsyncResult)
    Try
        ' Retrieve the state object and the client socket
        ' from the asynchronous state object.
        Dim state As StateObject = CType(ar.AsyncState, StateObject)
        Dim client As Socket = state.workSocket

        ' Read data from the remote device.
        Dim bytesRead As Integer = client.EndReceive(ar)

        If bytesRead > 0 Then
            ' There might be more data, so store the data received so far.
            state.sb.Append(Encoding.ASCII.GetString(state.buffer, 0, _
                bytesRead))

            '  Get the rest of the data.
            client.BeginReceive(state.buffer, 0, state.BufferSize, 0, _
                AddressOf ReceiveCallback, state)
        Else
            ' All the data has arrived; put it in response.
            If state.sb.Length > 1 Then
                response = state.sb.ToString()
            End If
            ' Signal that all bytes have been received.
            receiveDone.Set()
        End If
    Catch e As Exception
        Console.WriteLine(e.ToString())
    End Try
End Sub 'ReceiveCallback
```

```
private static void ReceiveCallback( IAsyncResult ar ) {
    try {
        // Retrieve the state object and the client socket
        // from the asynchronous state object.
        StateObject state = (StateObject) ar.AsyncState;
        Socket client = state.workSocket;
        // Read data from the remote device.
        int bytesRead = client.EndReceive(ar);
        if (bytesRead > 0) {
            // There might be more data, so store the data received so far.
            state.sb.Append(Encoding.ASCII.GetString(state.buffer,0,bytesRead));
                //  Get the rest of the data.
            client.BeginReceive(state.buffer,0,StateObject.BufferSize,0,
                new AsyncCallback(ReceiveCallback), state);
        } else {
            // All the data has arrived; put it in response.
            if (state.sb.Length > 1) {
                response = state.sb.ToString();
            }
            // Signal that all bytes have been received.
            receiveDone.Set();
        }
    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }
}
```

See Also

Using a Synchronous Client Socket
Listening with Sockets
Asynchronous Client Socket Example

Listening with Sockets

7/29/2017 • 1 min to read • Edit Online

Listener or server sockets open a port on the network and then wait for a client to connect to that port. Although other network address families and protocols exist, this example shows how to create remote service for a TCP/IP network.

The unique address of a TCP/IP service is defined by combining the IP address of the host with the port number of the service to create an endpoint for the service. The Dns class provides methods that return information about the network addresses supported by the local network device. When the local network device has more than one network address, or if the local system supports more than one network device, the **Dns** class returns information about all network addresses, and the application must choose the proper address for the service. The Internet Assigned Numbers Authority (Iana) defines port numbers for common services (for more information, see www.iana.org/assignments/port-numbers). Other services can have registered port numbers in the range 1,024 to 65,535.

The following example creates an IPEndPoint for a server by combining the first IP address returned by **Dns** for the host computer with a port number chosen from the registered port numbers range.

```
Dim ipHostInfo As IPHostEntry = Dns.Resolve(Dns.GetHostName())
Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
Dim localEndPoint As New IPEndPoint(ipAddress, 11000)
```

```
IPHostEntry ipHostInfo = Dns.Resolve(Dns.GetHostName());
IPAddress ipAddress = ipHostInfo.AddressList[0];
IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);
```

After the local endpoint is determined, the Socket must be associated with that endpoint using the Bind method and set to listen on the endpoint using the Listen method. **Bind** throws an exception if the specific address and port combination is already in use. The following example demonstrates associating a **Socket** with an **IPEndPoint**.

```
listener.Bind(localEndPoint)
listener.Listen(100)
```

```
listener.Bind(localEndPoint);
listener.Listen(100);
```

The **Listen** method takes a single parameter that specifies how many pending connections to the **Socket** are allowed before a server busy error is returned to the connecting client. In this case, up to 100 clients are placed in the connection queue before a server busy response is returned to client number 101.

See Also

Using a Synchronous Server Socket
Using an Asynchronous Server Socket
Using Client Sockets
How to: Create a Socket

Sockets

Using a Synchronous Server Socket

7/29/2017 • 1 min to read • Edit Online

Synchronous server sockets suspend the execution of the application until a connection request is received on the socket. Synchronous server sockets are not suitable for applications that make heavy use of the network in their operation, but they can be suitable for simple network applications.

After a Socket is set to listen on an endpoint using the Bind and Listen methods, it is ready to accept incoming connection requests using the Accept method. The application is suspended until a connection request is received when the **Accept** method is called.

When a connection request is received, **Accept** returns a new **Socket** instance that is associated with the connecting client. The following example reads data from the client, displays it on the console, and echoes the data back to the client. The **Socket** does not specify any messaging protocol, so the string "<EOF>" marks the end of the message data. It assumes that a **Socket** named `listener` has been initialized and bound to an endpoint.

```
Console.WriteLine("Waiting for a connection...")
Dim handler As Socket = listener.Accept()
Dim data As String = Nothing

While True
    bytes = New Byte(1024) {}
    Dim bytesRec As Integer = handler.Receive(bytes)
    data += Encoding.ASCII.GetString(bytes, 0, bytesRec)
    If data.IndexOf("<EOF>") > - 1 Then
        Exit While
    End If
End While

Console.WriteLine("Text received : {0}", data)

Dim msg As Byte() = Encoding.ASCII.GetBytes(data)
handler.Send(msg)
handler.Shutdown(SocketShutdown.Both)
handler.Close()
```

```
Console.WriteLine("Waiting for a connection...");
Socket handler = listener.Accept();
String data = null;

while (true) {
    bytes = new byte[1024];
    int bytesRec = handler.Receive(bytes);
    data += Encoding.ASCII.GetString(bytes,0,bytesRec);
    if (data.IndexOf("<EOF>") > -1) {
        break;
    }
}

Console.WriteLine( "Text received : {0}", data);

byte[] msg = Encoding.ASCII.GetBytes(data);
handler.Send(msg);
handler.Shutdown(SocketShutdown.Both);
handler.Close();
```

See Also

Using an Asynchronous Server Socket
Synchronous Server Socket Example
Listening with Sockets

Using an Asynchronous Server Socket

7/29/2017 • 6 min to read • Edit Online

Asynchronous server sockets use the .NET Framework asynchronous programming model to process network service requests. The Socket class follows the standard .NET Framework asynchronous naming pattern; for example, the synchronous Accept method corresponds to the asynchronous BeginAccept and EndAccept methods.

An asynchronous server socket requires a method to begin accepting connection requests from the network, a callback method to handle the connection requests and begin receiving data from the network, and a callback method to end receiving the data. All these methods are discussed further in this section.

In the following example, to begin accepting connection requests from the network, the method `StartListening` initializes the **Socket** and then uses the **BeginAccept** method to start accepting new connections. The accept callback method is called when a new connection request is received on the socket. It is responsible for getting the **Socket** instance that will handle the connection and handing that **Socket** off to the thread that will process the request. The accept callback method implements the AsyncCallback delegate; it returns a void and takes a single parameter of type IAsyncResult. The following example is the shell of an accept callback method.

```
Sub acceptCallback(ar As IAsyncResult)
    ' Add the callback code here.
End Sub 'acceptCallback
```

```
void acceptCallback( IAsyncResult ar) {
    // Add the callback code here.
}
```

The **BeginAccept** method takes two parameters, an **AsyncCallback** delegate that points to the accept callback method and an object that is used to pass state information to the callback method. In the following example, the listening **Socket** is passed to the callback method through the *state* parameter. This example creates an **AsyncCallback** delegate and starts accepting connections from the network.

```
listener.BeginAccept( _
    New AsyncCallback(SocketListener.acceptCallback),_
    listener)
```

```
listener.BeginAccept(
    new AsyncCallback(SocketListener.acceptCallback),
    listener);
```

Asynchronous sockets use threads from the system thread pool to process incoming connections. One thread is responsible for accepting connections, another thread is used to handle each incoming connection, and another thread is responsible for receiving data from the connection. These could be the same thread, depending on which thread is assigned by the thread pool. In the following example, the System.Threading.ManualResetEvent class suspends execution of the main thread and signals when execution can continue.

The following example shows an asynchronous method that creates an asynchronous TCP/IP socket on the local computer and begins accepting connections. It assumes that there is a global **ManualResetEvent** named `allDone`, that the method is a member of a class named `SocketListener`, and that a callback method named `acceptCallback` is defined.

```
Public Sub StartListening()
    Dim ipHostInfo As IPHostEntry = Dns.Resolve(Dns.GetHostName())
    Dim localEP = New IPEndPoint(ipHostInfo.AddressList(0), 11000)

    Console.WriteLine("Local address and port : {0}", localEP.ToString())

    Dim listener As New Socket(localEP.Address.AddressFamily, _
       SocketType.Stream, ProtocolType.Tcp)

    Try
        listener.Bind(localEP)
        listener.Listen(10)

        While True
            allDone.Reset()

            Console.WriteLine("Waiting for a connection...")
            listener.BeginAccept(New _
                AsyncCallback(SocketListener.acceptCallback), _
                listener)

            allDone.WaitOne()
        End While
    Catch e As Exception
        Console.WriteLine(e.ToString())
    End Try
    Console.WriteLine("Closing the listener...")
End Sub 'StartListening
```

```
public void StartListening() {
    IPHostEntry ipHostInfo = Dns.Resolve(Dns.GetHostName());
    IPEndPoint localEP = new IPEndPoint(ipHostInfo.AddressList[0],11000);

    Console.WriteLine("Local address and port : {0}",localEP.ToString());

    Socket listener = new Socket( localEP.Address.AddressFamily,
        SocketType.Stream, ProtocolType.Tcp );

    try {
        listener.Bind(localEP);
        listener.Listen(10);

        while (true) {
            allDone.Reset();

            Console.WriteLine("Waiting for a connection...");
            listener.BeginAccept(
                new AsyncCallback(SocketListener.acceptCallback),
                listener );

            allDone.WaitOne();
        }
    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }

    Console.WriteLine( "Closing the listener...");
}
```

The accept callback method (`acceptCallback` in the preceding example) is responsible for signaling the main application thread to continue processing, establishing the connection with the client, and starting the asynchronous read of data from the client. The following example is the first part of an implementation of the `acceptCallback` method. This section of the method signals the main application thread to continue processing

and establishes the connection to the client. It assumes a global **ManualResetEvent** named `allDone`.

```
Public Sub acceptCallback(ar As IAsyncResult)
    allDone.Set()

    Dim listener As Socket = CType(ar.AsyncState, Socket)
    Dim handler As Socket = listener.EndAccept(ar)

    ' Additional code to read data goes here.
End Sub 'acceptCallback
```

```
public void acceptCallback(IAsyncResult ar) {
    allDone.Set();

    Socket listener = (Socket) ar.AsyncState;
    Socket handler = listener.EndAccept(ar);

    // Additional code to read data goes here.
}
```

Reading data from a client socket requires a state object that passes values between asynchronous calls. The following example implements a state object for receiving a string from the remote client. It contains fields for the client socket, a data buffer for receiving data, and a StringBuilder for creating the data string sent by the client. Placing these fields in the state object allows their values to be preserved across multiple calls to read data from the client socket.

```
Public Class StateObject
    Public workSocket As Socket = Nothing
    Public BufferSize As Integer = 1024
    Public buffer(BufferSize) As Byte
    Public sb As New StringBuilder()
End Class 'StateObject
```

```
public class StateObject {
    public Socket workSocket = null;
    public const int BufferSize = 1024;
    public byte[] buffer = new byte[BufferSize];
    public StringBuilder sb = new StringBuilder();
}
```

The section of the `acceptCallback` method that starts receiving the data from the client socket first initializes an instance of the `StateObject` class and then calls the BeginReceive method to start reading the data from the client socket asynchronously.

The following example shows the complete `acceptCallback` method. It assumes that there is a global **ManualResetEvent** named `allDone,` that the `StateObject` class is defined, and that the `readCallback` method is defined in a class named `SocketListener`.

```
Public Shared Sub acceptCallback(ar As IAsyncResult)
    ' Get the socket that handles the client request.
    Dim listener As Socket = CType(ar.AsyncState, Socket)
    Dim handler As Socket = listener.EndAccept(ar)

    ' Signal the main thread to continue.
    allDone.Set()

    ' Create the state object.
    Dim state As New StateObject()
    state.workSocket = handler
    handler.BeginReceive(state.buffer, 0, state.BufferSize, 0, _
        AddressOf AsynchronousSocketListener.readCallback, state)
End Sub 'acceptCallback
```

```
public static void acceptCallback(IAsyncResult ar) {
    // Get the socket that handles the client request.
    Socket listener = (Socket) ar.AsyncState;
    Socket handler = listener.EndAccept(ar);

    // Signal the main thread to continue.
    allDone.Set();

    // Create the state object.
    StateObject state = new StateObject();
    state.workSocket = handler;
    handler.BeginReceive( state.buffer, 0, StateObject.BufferSize, 0,
        new AsyncCallback(AsynchronousSocketListener.readCallback), state);
}
```

The final method that needs to be implemented for the asynchronous socket server is the read callback method that returns the data sent by the client. Like the accept callback method, the read callback method is an **AsyncCallback** delegate. This method reads one or more bytes from the client socket into the data buffer and then calls the **BeginReceive** method again until the data sent by the client is complete. Once the entire message has been read from the client, the string is displayed on the console and the server socket handling the connection to the client is closed.

The following sample implements the `readCallback` method. It assumes that the `StateObject` class is defined.

```
Public Shared Sub readCallback(ar As IAsyncResult)
    Dim state As StateObject = CType(ar.AsyncState, StateObject)
    Dim handler As Socket = state.workSocket

    ' Read data from the client socket.
    Dim read As Integer = handler.EndReceive(ar)

    ' Data was read from the client socket.
    If read > 0 Then
        state.sb.Append(Encoding.ASCII.GetString(state.buffer, 0, read))
        handler.BeginReceive(state.buffer, 0, state.BufferSize, 0, _
            AddressOf readCallback, state)
    Else
        If state.sb.Length > 1 Then
            ' All the data has been read from the client;
            ' display it on the console.
            Dim content As String = state.sb.ToString()
            Console.WriteLine("Read {0} bytes from socket." + _
                ControlChars.Cr + " Data : {1}", content.Length, content)
        End If
    End If
End Sub 'readCallback
```

```
public static void readCallback(IAsyncResult ar) {
    StateObject state = (StateObject) ar.AsyncState;
    Socket handler = state.WorkSocket;

    // Read data from the client socket.
    int read = handler.EndReceive(ar);

    // Data was read from the client socket.
    if (read > 0) {
        state.sb.Append(Encoding.ASCII.GetString(state.buffer,0,read));
        handler.BeginReceive(state.buffer,0,StateObject.BufferSize, 0,
            new AsyncCallback(readCallback), state);
    } else {
        if (state.sb.Length > 1) {
            // All the data has been read from the client;
            // display it on the console.
            string content = state.sb.ToString();
            Console.WriteLine("Read {0} bytes from socket.\n Data : {1}",
               content.Length, content);
        }
        handler.Close();
    }
}
```

See Also

Using a Synchronous Server Socket
Asynchronous Server Socket Example
Threading
Listening with Sockets

Socket Code Examples

7/29/2017 • 1 min to read • Edit Online

The following code examples demonstrate how to use the Socket class as a client to connect to remote network services and as a server to listen for connections from remote clients.

In This Section

Synchronous Client Socket Example
Shows how to implement a synchronous Socket client that connects to a server and displays the data returned from the server.

Synchronous Server Socket Example
Shows how to implement a synchronous Socket server that accepts connections from a client and echoes back the data received from the client.

Asynchronous Client Socket Example
Shows how to implement an asynchronous Socket client that connects to a server and displays the data returned from the server.

Asynchronous Server Socket Example
Shows how to implement an asynchronous Socket server that accepts connections from a client and echoes back the data received from the client.

Related Sections

Sockets
Provides basic information about the System.Net.Sockets namespace and the Socket class.

Security in Network Programming
Describes how to use standard Internet security and authentication techniques.

Synchronous Client Socket Example

7/29/2017 • 2 min to read • Edit Online

The following example program creates a client that connects to a server. The client is built with a synchronous socket, so execution of the client application is suspended until the server returns a response. The application sends a string to the server and then displays the string returned by the server on the console.

```
Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Text

Public Class SynchronousSocketClient

    Public Shared Sub Main()
        ' Data buffer for incoming data.
        Dim bytes(1024) As Byte

        ' Connect to a remote device.

        ' Establish the remote endpoint for the socket.
        ' This example uses port 11000 on the local computer.
        Dim ipHostInfo As IPHostEntry = Dns.Resolve(Dns.GetHostName())
        Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
        Dim remoteEP As New IPEndPoint(ipAddress, 11000)

        ' Create a TCP/IP socket.
        Dim sender As New Socket(AddressFamily.InterNetwork, _
            SocketType.Stream, ProtocolType.Tcp)

        ' Connect the socket to the remote endpoint.
        sender.Connect(remoteEP)

        Console.WriteLine("Socket connected to {0}", _
            sender.RemoteEndPoint.ToString())

        ' Encode the data string into a byte array.
        Dim msg As Byte() = _
            Encoding.ASCII.GetBytes("This is a test<EOF>")

        ' Send the data through the socket.
        Dim bytesSent As Integer = sender.Send(msg)

        ' Receive the response from the remote device.
        Dim bytesRec As Integer = sender.Receive(bytes)
        Console.WriteLine("Echoed test = {0}", _
            Encoding.ASCII.GetString(bytes, 0, bytesRec))

        ' Release the socket.
        sender.Shutdown(SocketShutdown.Both)
        sender.Close()
    End Sub

End Class 'SynchronousSocketClient
```

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

public class SynchronousSocketClient {

    public static void StartClient() {
        // Data buffer for incoming data.
        byte[] bytes = new byte[1024];

        // Connect to a remote device.
        try {
            // Establish the remote endpoint for the socket.
            // This example uses port 11000 on the local computer.
            IPHostEntry ipHostInfo = Dns.Resolve(Dns.GetHostName());
            IPAddress ipAddress = ipHostInfo.AddressList[0];
            IPEndPoint remoteEP = new IPEndPoint(ipAddress,11000);

            // Create a TCP/IP  socket.
            Socket sender = new Socket(AddressFamily.InterNetwork,
                SocketType.Stream, ProtocolType.Tcp );

            // Connect the socket to the remote endpoint. Catch any errors.
            try {
                sender.Connect(remoteEP);

                Console.WriteLine("Socket connected to {0}",
                    sender.RemoteEndPoint.ToString());

                // Encode the data string into a byte array.
                byte[] msg = Encoding.ASCII.GetBytes("This is a test<EOF>");

                // Send the data through the socket.
                int bytesSent = sender.Send(msg);

                // Receive the response from the remote device.
                int bytesRec = sender.Receive(bytes);
                Console.WriteLine("Echoed test = {0}",
                    Encoding.ASCII.GetString(bytes,0,bytesRec));

                // Release the socket.
                sender.Shutdown(SocketShutdown.Both);
                sender.Close();

            } catch (ArgumentNullException ane) {
                Console.WriteLine("ArgumentNullException : {0}",ane.ToString());
            } catch (SocketException se) {
                Console.WriteLine("SocketException : {0}",se.ToString());
            } catch (Exception e) {
                Console.WriteLine("Unexpected exception : {0}", e.ToString());
            }

        } catch (Exception e) {
            Console.WriteLine( e.ToString());
        }
    }

    public static int Main(String[] args) {
        StartClient();
        return 0;
    }
}
```

See Also

Synchronous Server Socket Example

7/29/2017 • 2 min to read • Edit Online

The following example program creates a server that receives connection requests from clients. The server is built with a synchronous socket, so execution of the server application is suspended while it waits for a connection from a client. The application receives a string from the client, displays the string on the console, and then echoes the string back to the client. The string from the client must contain the string "<EOF>" to signal the end of the message.

```
Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Text
Imports Microsoft.VisualBasic

Public Class SynchronousSocketListener

    ' Incoming data from the client.
    Public Shared data As String = Nothing

    Public Shared Sub Main()
        ' Data buffer for incoming data.
        Dim bytes() As Byte = New [Byte](1024) {}

        ' Establish the local endpoint for the socket.
        ' Dns.GetHostName returns the name of the
        ' host running the application.
        Dim ipHostInfo As IPHostEntry = Dns.Resolve(Dns.GetHostName())
        Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
        Dim localEndPoint As New IPEndPoint(ipAddress, 11000)

        ' Create a TCP/IP socket.
        Dim listener As New Socket(AddressFamily.InterNetwork, _
            SocketType.Stream, ProtocolType.Tcp)

        ' Bind the socket to the local endpoint and
        ' listen for incoming connections.

        listener.Bind(localEndPoint)
        listener.Listen(10)

        ' Start listening for connections.
        While True
            Console.WriteLine("Waiting for a connection...")
            ' Program is suspended while waiting for an incoming connection.
            Dim handler As Socket = listener.Accept()
            data = Nothing

            ' An incoming connection needs to be processed.
            While True
                bytes = New Byte(1024) {}
                Dim bytesRec As Integer = handler.Receive(bytes)
                data += Encoding.ASCII.GetString(bytes, 0, bytesRec)
                If data.IndexOf("<EOF>") > -1 Then
                    Exit While
                End If
            End While
            ' Show the data on the console.
            Console.WriteLine("Text received : {0}", data)
            ' Echo the data back to the client.
            Dim msg As Byte() = Encoding.ASCII.GetBytes(data)
            handler.Send(msg)
            handler.Shutdown(SocketShutdown.Both)
            handler.Close()
        End While
    End Sub

End Class 'SynchronousSocketListener
```

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

public class SynchronousSocketListener {
```

```
    // Incoming data from the client.
    public static string data = null;

    public static void StartListening() {
        // Data buffer for incoming data.
        byte[] bytes = new Byte[1024];

        // Establish the local endpoint for the socket.
        // Dns.GetHostName returns the name of the
        // host running the application.
        IPHostEntry ipHostInfo = Dns.Resolve(Dns.GetHostName());
        IPAddress ipAddress = ipHostInfo.AddressList[0];
        IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);

        // Create a TCP/IP socket.
        Socket listener = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp );

        // Bind the socket to the local endpoint and
        // listen for incoming connections.
        try {
            listener.Bind(localEndPoint);
            listener.Listen(10);

            // Start listening for connections.
            while (true) {
                Console.WriteLine("Waiting for a connection...");
                // Program is suspended while waiting for an incoming connection.
                Socket handler = listener.Accept();
                data = null;

                // An incoming connection needs to be processed.
                while (true) {
                    bytes = new byte[1024];
                    int bytesRec = handler.Receive(bytes);
                    data += Encoding.ASCII.GetString(bytes,0,bytesRec);
                    if (data.IndexOf("<EOF>") > -1) {
                        break;
                    }
                }

                // Show the data on the console.
                Console.WriteLine( "Text received : {0}", data);

                // Echo the data back to the client.
                byte[] msg = Encoding.ASCII.GetBytes(data);

                handler.Send(msg);
                handler.Shutdown(SocketShutdown.Both);
                handler.Close();
            }

        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }

        Console.WriteLine("\nPress ENTER to continue...");
        Console.Read();

    }

    public static int Main(String[] args) {
        StartListening();
        return 0;
    }
}
```

See Also

Synchronous Client Socket Example
Using a Synchronous Server Socket
Socket Code Examples

Asynchronous Client Socket Example

7/29/2017 • 5 min to read • Edit Online

The following example program creates a client that connects to a server. The client is built with an asynchronous socket, so execution of the client application is not suspended while the server returns a response. The application sends a string to the server and then displays the string returned by the server on the console.

```
Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Threading
Imports System.Text

' State object for receiving data from remote device.

Public Class StateObject
    ' Client socket.
    Public workSocket As Socket = Nothing
    ' Size of receive buffer.
    Public Const BufferSize As Integer = 256
    ' Receive buffer.
    Public buffer(BufferSize) As Byte
    ' Received data string.
    Public sb As New StringBuilder
End Class 'StateObject

Public Class AsynchronousClient
    ' The port number for the remote device.
    Private Const port As Integer = 11000

    ' ManualResetEvent instances signal completion.
    Private Shared connectDone As New ManualResetEvent(False)
    Private Shared sendDone As New ManualResetEvent(False)
    Private Shared receiveDone As New ManualResetEvent(False)

    ' The response from the remote device.
    Private Shared response As String = String.Empty

    Public Shared Sub Main()
        ' Establish the remote endpoint for the socket.
        ' For this example use local machine.
        Dim ipHostInfo As IPHostEntry = Dns.Resolve(Dns.GetHostName())
        Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
        Dim remoteEP As New IPEndPoint(ipAddress, port)

        ' Create a TCP/IP socket.
        Dim client As New Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp)

        ' Connect to the remote endpoint.
        client.BeginConnect(remoteEP, New AsyncCallback(AddressOf ConnectCallback), client)

        ' Wait for connect.
        connectDone.WaitOne()

        ' Send test data to the remote device.
        Send(client, "This is a test<EOF>")
        sendDone.WaitOne()

        ' Receive the response from the remote device.
        Receive(client)
        receiveDone.WaitOne()
```

```
        ' Write the response to the console.
        Console.WriteLine("Response received : {0}", response)

        ' Release the socket.
        client.Shutdown(SocketShutdown.Both)
        client.Close()
    End Sub 'Main

    Private Shared Sub ConnectCallback(ByVal ar As IAsyncResult)
        ' Retrieve the socket from the state object.
        Dim client As Socket = CType(ar.AsyncState, Socket)

        ' Complete the connection.
        client.EndConnect(ar)

        Console.WriteLine("Socket connected to {0}", client.RemoteEndPoint.ToString())

        ' Signal that the connection has been made.
        connectDone.Set()
    End Sub 'ConnectCallback

    Private Shared Sub Receive(ByVal client As Socket)

        ' Create the state object.
        Dim state As New StateObject
        state.workSocket = client

        ' Begin receiving the data from the remote device.
        client.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, New AsyncCallback(AddressOf
ReceiveCallback), state)
    End Sub 'Receive

    Private Shared Sub ReceiveCallback(ByVal ar As IAsyncResult)

        ' Retrieve the state object and the client socket
        ' from the asynchronous state object.
        Dim state As StateObject = CType(ar.AsyncState, StateObject)
        Dim client As Socket = state.workSocket

        ' Read data from the remote device.
        Dim bytesRead As Integer = client.EndReceive(ar)

        If bytesRead > 0 Then
            ' There might be more data, so store the data received so far.
            state.sb.Append(Encoding.ASCII.GetString(state.buffer, 0, bytesRead))

            ' Get the rest of the data.
            client.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, New AsyncCallback(AddressOf
ReceiveCallback), state)
        Else
            ' All the data has arrived; put it in response.
            If state.sb.Length > 1 Then
                response = state.sb.ToString()
            End If
            ' Signal that all bytes have been received.
            receiveDone.Set()
        End If
    End Sub 'ReceiveCallback

    Private Shared Sub Send(ByVal client As Socket, ByVal data As String)
        ' Convert the string data to byte data using ASCII encoding.
        Dim byteData As Byte() = Encoding.ASCII.GetBytes(data)

        ' Begin sending the data to the remote device.
        client.BeginSend(byteData, 0, byteData.Length, 0, New AsyncCallback(AddressOf SendCallback), client)
    End Sub 'Send

    Private Shared Sub SendCallback(ByVal ar As IAsyncResult)
        ' Retrieve the socket from the state object.
```

```
        ' Retrieve the socket from the state object.
        Dim client As Socket = CType(ar.AsyncState, Socket)

        ' Complete sending the data to the remote device.
        Dim bytesSent As Integer = client.EndSend(ar)
        Console.WriteLine("Sent {0} bytes to server.", bytesSent)

        ' Signal that all bytes have been sent.
        sendDone.Set()
    End Sub 'SendCallback
End Class 'AsynchronousClient
```

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Text;

// State object for receiving data from remote device.
public class StateObject {
    // Client socket.
    public Socket workSocket = null;
    // Size of receive buffer.
    public const int BufferSize = 256;
    // Receive buffer.
    public byte[] buffer = new byte[BufferSize];
    // Received data string.
    public StringBuilder sb = new StringBuilder();
}

public class AsynchronousClient {
    // The port number for the remote device.
    private const int port = 11000;

    // ManualResetEvent instances signal completion.
    private static ManualResetEvent connectDone =
        new ManualResetEvent(false);
    private static ManualResetEvent sendDone =
        new ManualResetEvent(false);
    private static ManualResetEvent receiveDone =
        new ManualResetEvent(false);

    // The response from the remote device.
    private static String response = String.Empty;

    private static void StartClient() {
        // Connect to a remote device.
        try {
            // Establish the remote endpoint for the socket.
            // The name of the
            // remote device is "host.contoso.com".
            IPHostEntry ipHostInfo = Dns.Resolve("host.contoso.com");
            IPAddress ipAddress = ipHostInfo.AddressList[0];
            IPEndPoint remoteEP = new IPEndPoint(ipAddress, port);

            // Create a TCP/IP socket.
            Socket client = new Socket(AddressFamily.InterNetwork,
                SocketType.Stream, ProtocolType.Tcp);

            // Connect to the remote endpoint.
            client.BeginConnect( remoteEP,
                new AsyncCallback(ConnectCallback), client);
            connectDone.WaitOne();

            // Send test data to the remote device.
            Send(client,"This is a test<EOF>");
            sendDone.WaitOne();
```

```
        // Receive the response from the remote device.
        Receive(client);
        receiveDone.WaitOne();

        // Write the response to the console.
        Console.WriteLine("Response received : {0}", response);

        // Release the socket.
        client.Shutdown(SocketShutdown.Both);
        client.Close();

    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }
}

private static void ConnectCallback(IAsyncResult ar) {
    try {
        // Retrieve the socket from the state object.
        Socket client = (Socket) ar.AsyncState;

        // Complete the connection.
        client.EndConnect(ar);

        Console.WriteLine("Socket connected to {0}",
            client.RemoteEndPoint.ToString());

        // Signal that the connection has been made.
        connectDone.Set();
    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }
}

private static void Receive(Socket client) {
    try {
        // Create the state object.
        StateObject state = new StateObject();
        state.workSocket = client;

        // Begin receiving the data from the remote device.
        client.BeginReceive( state.buffer, 0, StateObject.BufferSize, 0,
            new AsyncCallback(ReceiveCallback), state);
    } catch (Exception e) {
        Console.WriteLine(e.ToString());
    }
}

private static void ReceiveCallback( IAsyncResult ar ) {
    try {
        // Retrieve the state object and the client socket
        // from the asynchronous state object.
        StateObject state = (StateObject) ar.AsyncState;
        Socket client = state.workSocket;

        // Read data from the remote device.
        int bytesRead = client.EndReceive(ar);

        if (bytesRead > 0) {
            // There might be more data, so store the data received so far.
        state.sb.Append(Encoding.ASCII.GetString(state.buffer,0,bytesRead));

            // Get the rest of the data.
            client.BeginReceive(state.buffer,0,StateObject.BufferSize,0,
                new AsyncCallback(ReceiveCallback), state);
        } else {
            // All the data has arrived; put it in response.
            if (state.sb.Length > 1) {
```

```
                    response = state.sb.ToString();
                }
                // Signal that all bytes have been received.
                receiveDone.Set();
            }
        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    private static void Send(Socket client, String data) {
        // Convert the string data to byte data using ASCII encoding.
        byte[] byteData = Encoding.ASCII.GetBytes(data);

        // Begin sending the data to the remote device.
        client.BeginSend(byteData, 0, byteData.Length, 0,
            new AsyncCallback(SendCallback), client);
    }

    private static void SendCallback(IAsyncResult ar) {
        try {
            // Retrieve the socket from the state object.
            Socket client = (Socket) ar.AsyncState;

            // Complete sending the data to the remote device.
            int bytesSent = client.EndSend(ar);
            Console.WriteLine("Sent {0} bytes to server.", bytesSent);

            // Signal that all bytes have been sent.
            sendDone.Set();
        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    public static int Main(String[] args) {
        StartClient();
        return 0;
    }
}
```

See Also

Asynchronous Server Socket Example
Using a Synchronous Server Socket
Socket Code Examples

Asynchronous Server Socket Example

7/29/2017 • 5 min to read • Edit Online

The following example program creates a server that receives connection requests from clients. The server is built with an asynchronous socket, so execution of the server application is not suspended while it waits for a connection from a client. The application receives a string from the client, displays the string on the console, and then echoes the string back to the client. The string from the client must contain the string "<EOF>" to signal the end of the message.

```
Imports System
Imports System.Net
Imports System.Net.Sockets
Imports System.Text
Imports System.Threading
Imports Microsoft.VisualBasic

' State object for reading client data asynchronously

Public Class StateObject
    ' Client  socket.
    Public workSocket As Socket = Nothing
    ' Size of receive buffer.
    Public Const BufferSize As Integer = 1024
    ' Receive buffer.
    Public buffer(BufferSize) As Byte
    ' Received data string.
    Public sb As New StringBuilder
End Class 'StateObject

Public Class AsynchronousSocketListener
    ' Thread signal.
    Public Shared allDone As New ManualResetEvent(False)

    ' This server waits for a connection and then uses  asychronous operations to
    ' accept the connection, get data from the connected client,
    ' echo that data back to the connected client.
    ' It then disconnects from the client and waits for another client.
    Public Shared Sub Main()
        ' Data buffer for incoming data.
        Dim bytes() As Byte = New [Byte](1023) {}

        ' Establish the local endpoint for the socket.
        Dim ipHostInfo As IPHostEntry = Dns.Resolve(Dns.GetHostName())
        Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
        Dim localEndPoint As New IPEndPoint(ipAddress, 11000)

        ' Create a TCP/IP socket.
        Dim listener As New Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp)

        ' Bind the socket to the local endpoint and listen for incoming connections.
        listener.Bind(localEndPoint)
        listener.Listen(100)

        While True
            ' Set the event to nonsignaled state.
            allDone.Reset()

            ' Start an asynchronous socket to listen for connections.
            Console.WriteLine("Waiting for a connection...")
            listener.BeginAccept(New AsyncCallback(AddressOf AcceptCallback), listener)
```

```
            ' Wait until a connection is made and processed before continuing.
            allDone.WaitOne()
        End While
    End Sub 'Main

    Public Shared Sub AcceptCallback(ByVal ar As IAsyncResult)
        ' Get the socket that handles the client request.
        Dim listener As Socket = CType(ar.AsyncState, Socket)
        ' End the operation.
        Dim handler As Socket = listener.EndAccept(ar)

        ' Create the state object for the async receive.
        Dim state As New StateObject
        state.workSocket = handler
        handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, New AsyncCallback(AddressOf
ReadCallback), state)
    End Sub 'AcceptCallback

    Public Shared Sub ReadCallback(ByVal ar As IAsyncResult)
        Dim content As String = String.Empty

        ' Retrieve the state object and the handler socket
        ' from the asynchronous state object.
        Dim state As StateObject = CType(ar.AsyncState, StateObject)
        Dim handler As Socket = state.workSocket

        ' Read data from the client socket.
        Dim bytesRead As Integer = handler.EndReceive(ar)

        If bytesRead > 0 Then
            ' There  might be more data, so store the data received so far.
            state.sb.Append(Encoding.ASCII.GetString(state.buffer, 0, bytesRead))

            ' Check for end-of-file tag. If it is not there, read
            ' more data.
            content = state.sb.ToString()
            If content.IndexOf("<EOF>") > -1 Then
                ' All the data has been read from the
                ' client. Display it on the console.
                Console.WriteLine("Read {0} bytes from socket. " + vbLf + " Data : {1}", content.Length,
content)
                ' Echo the data back to the client.
                Send(handler, content)
            Else
                ' Not all data received. Get more.
                handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, New AsyncCallback(AddressOf
ReadCallback), state)
            End If
        End If
    End Sub 'ReadCallback

    Private Shared Sub Send(ByVal handler As Socket, ByVal data As String)
        ' Convert the string data to byte data using ASCII encoding.
        Dim byteData As Byte() = Encoding.ASCII.GetBytes(data)

        ' Begin sending the data to the remote device.
        handler.BeginSend(byteData, 0, byteData.Length, 0, New AsyncCallback(AddressOf SendCallback), handler)
    End Sub 'Send

    Private Shared Sub SendCallback(ByVal ar As IAsyncResult)
        ' Retrieve the socket from the state object.
        Dim handler As Socket = CType(ar.AsyncState, Socket)

        ' Complete sending the data to the remote device.
        Dim bytesSent As Integer = handler.EndSend(ar)
        Console.WriteLine("Sent {0} bytes to client.", bytesSent)

        handler.Shutdown(SocketShutdown.Both)
        handler.Close()
```

```
        ' Signal the main thread to continue.
        allDone.Set()
    End Sub 'SendCallback
End Class 'AsynchronousSocketListener
```

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

// State object for reading client data asynchronously
public class StateObject {
    // Client  socket.
    public Socket workSocket = null;
    // Size of receive buffer.
    public const int BufferSize = 1024;
    // Receive buffer.
    public byte[] buffer = new byte[BufferSize];
// Received data string.
    public StringBuilder sb = new StringBuilder();
}

public class AsynchronousSocketListener {
    // Thread signal.
    public static ManualResetEvent allDone = new ManualResetEvent(false);

    public AsynchronousSocketListener() {
    }

    public static void StartListening() {
        // Data buffer for incoming data.
        byte[] bytes = new Byte[1024];

        // Establish the local endpoint for the socket.
        // The DNS name of the computer
        // running the listener is "host.contoso.com".
        IPHostEntry ipHostInfo = Dns.Resolve(Dns.GetHostName());
        IPAddress ipAddress = ipHostInfo.AddressList[0];
        IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);

        // Create a TCP/IP socket.
        Socket listener = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp );

        // Bind the socket to the local endpoint and listen for incoming connections.
        try {
            listener.Bind(localEndPoint);
            listener.Listen(100);

            while (true) {
                // Set the event to nonsignaled state.
                allDone.Reset();

                // Start an asynchronous socket to listen for connections.
                Console.WriteLine("Waiting for a connection...");
                listener.BeginAccept(
                    new AsyncCallback(AcceptCallback),
                    listener );

                // Wait until a connection is made before continuing.
                allDone.WaitOne();
            }

        } catch (Exception e) {
            Console.WriteLine(e.ToString());
```

```
        }

        Console.WriteLine("\nPress ENTER to continue...");
        Console.Read();

    }

    public static void AcceptCallback(IAsyncResult ar) {
        // Signal the main thread to continue.
        allDone.Set();

        // Get the socket that handles the client request.
        Socket listener = (Socket) ar.AsyncState;
        Socket handler = listener.EndAccept(ar);

        // Create the state object.
        StateObject state = new StateObject();
        state.workSocket = handler;
        handler.BeginReceive( state.buffer, 0, StateObject.BufferSize, 0,
            new AsyncCallback(ReadCallback), state);
    }

    public static void ReadCallback(IAsyncResult ar) {
        String content = String.Empty;

        // Retrieve the state object and the handler socket
        // from the asynchronous state object.
        StateObject state = (StateObject) ar.AsyncState;
        Socket handler = state.workSocket;

        // Read data from the client socket.
        int bytesRead = handler.EndReceive(ar);

        if (bytesRead > 0) {
            // There  might be more data, so store the data received so far.
            state.sb.Append(Encoding.ASCII.GetString(
                state.buffer,0,bytesRead));

            // Check for end-of-file tag. If it is not there, read
            // more data.
            content = state.sb.ToString();
            if (content.IndexOf("<EOF>") > -1) {
                // All the data has been read from the
                // client. Display it on the console.
                Console.WriteLine("Read {0} bytes from socket. \n Data : {1}",
                    content.Length, content );
                // Echo the data back to the client.
                Send(handler, content);
            } else {
                // Not all data received. Get more.
                handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,
                new AsyncCallback(ReadCallback), state);
            }
        }
    }

    private static void Send(Socket handler, String data) {
        // Convert the string data to byte data using ASCII encoding.
        byte[] byteData = Encoding.ASCII.GetBytes(data);

        // Begin sending the data to the remote device.
        handler.BeginSend(byteData, 0, byteData.Length, 0,
            new AsyncCallback(SendCallback), handler);
    }

    private static void SendCallback(IAsyncResult ar) {
        try {
            // Retrieve the socket from the state object.
            Socket handler = (Socket) ar.AsyncState;
```

```
            // Complete sending the data to the remote device.
            int bytesSent = handler.EndSend(ar);
            Console.WriteLine("Sent {0} bytes to client.", bytesSent);

            handler.Shutdown(SocketShutdown.Both);
            handler.Close();

        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    public static int Main(String[] args) {
        StartListening();
        return 0;
    }
}
```

See Also

Asynchronous Client Socket Example
Using an Asynchronous Server Socket
Socket Code Examples

FTP

7/29/2017 • 1 min to read • Edit Online

The .NET Framework provides comprehensive support for the FTP protocol with the FtpWebRequest and FtpWebResponse classes. These classes are derived from WebRequest and WebResponse. In most cases, the WebRequest and WebResponse classes provide all that is necessary to make the request, but if you need access to the FTP-specific features exposed as properties, you can typecast these classes to FtpWebRequest or FtpWebResponse.

Examples

For more information, see the following topics: How to: Download Files with FTP, How to: Upload Files with FTP, and How to: List Directory Contents with FTP.

FTP and proxies

If a proxy (specified by the Proxy property) is an HTTP proxy, then only the DownloadFile, ListDirectory, and ListDirectoryDetails commands are supported.

See Also

Accessing the Internet Through a Proxy
Network Programming Samples
FTP Client Technology Sample
FTP Explorer Technology Sample
Using Application Protocols

How to: Download Files with FTP

7/29/2017 • 1 min to read • Edit Online

This sample shows how to download a file from an FTP server.

Example

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

namespace Examples.System.Net
{
    public class WebRequestGetExample
    {
        public static void Main ()
        {
            // Get the object used to communicate with the server.
            FtpWebRequest request = (FtpWebRequest)WebRequest.Create("ftp://www.contoso.com/test.htm");
            request.Method = WebRequestMethods.Ftp.DownloadFile;

            // This example assumes the FTP site uses anonymous logon.
            request.Credentials = new NetworkCredential ("anonymous","janeDoe@contoso.com");

            FtpWebResponse response = (FtpWebResponse)request.GetResponse();

            Stream responseStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(responseStream);
            Console.WriteLine(reader.ReadToEnd());

            Console.WriteLine("Download Complete, status {0}", response.StatusDescription);

            reader.Close();
            response.Close();
        }
    }
}
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

Robust Programming

.NET Framework Security

How to: Upload Files with FTP

7/29/2017 • 1 min to read • Edit Online

This sample shows how to upload a file to an FTP server.

Example

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

namespace Examples.System.Net
{
    public class WebRequestGetExample
    {
        public static void Main ()
        {
            // Get the object used to communicate with the server.
            FtpWebRequest request = (FtpWebRequest)WebRequest.Create("ftp://www.contoso.com/test.htm");
            request.Method = WebRequestMethods.Ftp.UploadFile;

            // This example assumes the FTP site uses anonymous logon.
            request.Credentials = new NetworkCredential ("anonymous","janeDoe@contoso.com");

            // Copy the contents of the file to the request stream.
            StreamReader sourceStream = new StreamReader("testfile.txt");
            byte [] fileContents = Encoding.UTF8.GetBytes(sourceStream.ReadToEnd());
            sourceStream.Close();
            request.ContentLength = fileContents.Length;

            Stream requestStream = request.GetRequestStream();
            requestStream.Write(fileContents, 0, fileContents.Length);
            requestStream.Close();

            FtpWebResponse response = (FtpWebResponse)request.GetResponse();

            Console.WriteLine("Upload File Complete, status {0}", response.StatusDescription);

            response.Close();
            }
        }
    }
}
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

Robust Programming

.NET Framework Security

How to: List Directory Contents with FTP

7/29/2017 • 1 min to read • Edit Online

This sample shows how to list the directory contents of an FTP server.

Example

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

namespace Examples.System.Net
{
    public class WebRequestGetExample
    {
        public static void Main ()
        {
            // Get the object used to communicate with the server.
            FtpWebRequest request = (FtpWebRequest)WebRequest.Create("ftp://www.contoso.com/");
            request.Method = WebRequestMethods.Ftp.ListDirectoryDetails;

            // This example assumes the FTP site uses anonymous logon.
            request.Credentials = new NetworkCredential ("anonymous","janeDoe@contoso.com");

            FtpWebResponse response = (FtpWebResponse)request.GetResponse();

            Stream responseStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(responseStream);
            Console.WriteLine(reader.ReadToEnd());

            Console.WriteLine("Directory List Complete, status {0}", response.StatusDescription);

            reader.Close();
            response.Close();
        }
    }
}
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

Robust Programming

.NET Framework Security

Understanding WebRequest Problems and Exceptions

7/29/2017 • 3 min to read • Edit Online

WebRequest and its derived classes (HttpWebRequest, FtpWebRequest, and FileWebRequest) throw exceptions to signal an abnormal condition. Sometimes the resolution of these problems is not obvious.

Solutions

Examine the Status property of the WebException to determine the problem. The following table shows several status values and some possible resolutions.

STATUS	DETAILS	SOLUTION
SendFailure -or- ReceiveFailure	There is a problem with the underlying socket. The connection may have been reset.	Reconnect and resend the request. Make sure the latest service pack is installed. Increase the value of the System.Net.ServicePointManager.MaxServicePointIdleTime property. Set System.Net.HttpWebRequest.KeepAlive to `false`. Increase the number of maximum connections with the DefaultConnectionLimit property. Check the proxy configuration. If using SSL, make sure the server process has permission to access the Certificate store. If sending a large amount of data, set AllowWriteStreamBuffering to `false`.

STATUS	DETAILS	SOLUTION
TrustFailure	The server certificate could not be validated.	Try to open the URI using Internet Explorer. Resolve any Security Alerts displayed by IE. If you cannot resolve the security alert, then you can create a certificate policy class that implements ICertificatePolicy that returns `true`, and pass it to CertificatePolicy. Refer to http://support.microsoft.com/?id=823177. Make sure that the certificate of the Certificate Authority that signed the server certificate is added to the Trusted Certificate Authority list in Internet Explorer. Make sure that the host name in the URL matches the common name on the server certificate.
SecureChannelFailure	An error occurred in the SSL transaction, or there is a certificate problem.	The .NET Framework version 1.1 only supports SSL version 3.0. If the server is using only TLS version 1.0 or SSL version 2.0, the exception is thrown. Upgrade to .NET Framework version 2.0, and set SecurityProtocol to match the server. The client certificate was signed by a Certificate Authority (CA) that the server does not trust. Install the CA's certificate on the server. See http://support.microsoft.com/?id=332077. Make sure you have the latest service pack installed.
ConnectFailure	The connection failed.	A firewall or proxy is blocking the connection. Modify the firewall or proxy to allow the connection. Explicitly designate a WebProxy in the client application by calling the WebProxy constructor (WebServiceProxyClass.Proxy = new WebProxy(http://server:80, true)). Run Filemon or Regmon to ensure that the worker process identity has the necessary permissions to access WSPWSP.dll, HKLM\System\CurrentControlSet\Services\DnsCache or HKLM\System\CurrentControlSet\Services\WinSock2.

STATUS	DETAILS	SOLUTION
NameResolutionFailure	The Domain Name Service could not resolve the host name.	Configure the proxy correctly. See http://support.microsoft.com/?id=318140. Ensure that any installed anti-virus software or firewall is not blocking the connection.
RequestCanceled	Abort was called, or an error occurred.	This problem might be caused by a heavy load on the client or server. Reduce the load. Increase the DefaultConnectionLimit setting. See http://support.microsoft.com/?id=821268 to modify Web service performance settings.
ConnectionClosed	The application attempted to write to a socket that has already been closed.	The client or server is overloaded. Reduce the load. Increase the DefaultConnectionLimit setting. See http://support.microsoft.com/?id=821268 to modify Web service performance settings.
MessageLengthLimitExceeded	The limit set (MaximumResponseHeadersLength) on the message length was exceeded.	Increase the value of the MaximumResponseHeadersLength property.
ProxyNameResolutionFailure	The Domain Name Service could not resolve the proxy host name.	Configure the proxy correctly. See http://support.microsoft.com/?id=318140. Force HttpWebRequest to use no proxy by setting the Proxy property to `null`.
ServerProtocolViolation	The response from the server is not a valid HTTP response. This problem occurs when the .NET Framework detects that the server response does not comply with HTTP 1.1 RFC. This problem may occur when the response contains incorrect headers or incorrect header delimiters.RFC 2616 defines HTTP 1.1 and the valid format for the response from the server. For more information, see http://www.ietf.org.	Get a network trace of the transaction and examine the headers in the response. If your application requires the server response without parsing (this could be a security issue), set `useUnsafeHeaderParsing` to `true` in the configuration file. See <httpWebRequest> Element (Network Settings).

See Also

HttpWebRequest
HttpWebResponse
Dns

Internet Protocol Version 6

7/29/2017 • 1 min to read • Edit Online

The Internet Protocol version 6 (IPv6) is a new suite of standard protocols for the network layer of the Internet. IPv6 is designed to solve many of the problems of the current version of the Internet Protocol suite (known as IPv4) with regard to address depletion, security, auto-configuration, extensibility, and so on. IPv6 expands the capabilities of the Internet to enable new kinds of applications, including peer-to-peer and mobile applications. The following are the main issues of the current IPv4 protocol:

- Rapid depletion of the address space.

  This has led to the use of Network Address Translators (NATs) that map multiple private addresses to a single public IP address. The main problems created by this mechanism are processing overhead and lack of end-to-end connectivity.

- Lack of hierarchy support.

  Because of its inherent predefined class organization, IPv4 lacks true hierarchical support. It is impossible to structure the IP addresses in a way that truly maps the network topology. This crucial design flaw creates the need for large routing tables to deliver IPv4 packets to any location on the Internet.

- Complex network configuration.

  With IPv4, addresses must be assigned statically or using a configuration protocol such as DHCP. In an ideal situation, hosts would not have to rely on the administration of a DHCP infrastructure. Instead, they would be able to configure themselves based on the network segment in which they are located.

- Lack of built-in authentication and confidentiality.

  IPv4 does not require the support for any mechanism that provides authentication or encryption of the exchanged data. This changes with IPv6. Internet Protocol security (IPSec) is an IPv6 support requirement.

A new protocol suite must satisfy the following basic requirements:

- Large-scale routing and addressing with low overhead.
- Auto-configuration for various connecting situations.
- Built-in authentication and confidentiality.

For more information, see IPv6 Addressing, IPv6 Routing, IPv6 Auto-Configuration, Enabling and Disabling IPv6, and How to: Modify the Computer Configuration File to Enable IPv6 Support.

References

The following are selected RFC documents that you can find at the Internet Engineering Task Force site (http://www.ietf.org):

- RFC 1287, Towards the Future Internet Architecture.
- RFC 1454, Comparison of Proposals for Next Version of IP.
- RFC 2373, IP Version 6 Addressing Architecture.
- RFC 2374, An IPv6 Aggregatable Global Unicast Address Format.

You can also find IPv6-related information on the IPv6 area on Technet.

See Also

IPv6 Sockets Sample
Network Programming Samples
Sockets

IPv6 Addressing

7/29/2017 • 3 min to read • Edit Online

In the Internet Protocol version 6 (IPv6), addresses are 128 bits long. One reason for such a large address space is to subdivide the available addresses into a hierarchy of routing domains that reflect the Internet's topology. Another reason is to map the addresses of network adapters (or interfaces) that connect devices to the network. IPv6 features an inherent capability to resolve addresses at their lowest level, which is at the network interface level, and also has auto-configuration capabilities.

Text Representation

The following are the three conventional forms used to represent the IPv6 addresses as text strings:

- **Colon-hexadecimal form**. This is the preferred form n:n:n:n:n:n:n:n. Each n represents the hexadecimal value of one of the eight 16-bit elements of the address. For example: `3FFE:FFFF:7654:FEDA:1245:BA98:3210:4562`.
- **Compressed form**. Due to the address length, it is common to have addresses containing a long string of zeros. To simplify writing these addresses, use the compressed form, in which a single contiguous sequence of 0 blocks are represented by a double-colon symbol (::). This symbol can appear only once in an address. For example, the multicast address `FFED:0:0:0:0:BA98:3210:4562` in compressed form is `FFED::BA98:3210:4562`. The unicast address `3FFE:FFFF:0:0:8:800:20C4:0` in compressed form is `3FFE:FFFF::8:800:20C4:0`. The loopback address `0:0:0:0:0:0:0:1` in compressed form is `::`1. The unspecified address `0:0:0:0:0:0:0:0` in compressed form is `::`.
- **Mixed form**. This form combines IPv4 and IPv6 addresses. In this case, the address format is n:n:n:n:n:n:d.d.d.d, where each n represents the hexadecimal values of the six IPv6 high-order 16-bit address elements, and each d represents the decimal value of an IPv4 address.

Address Types

The leading bits in the address define the specific IPv6 address type. The variable-length field containing these leading bits is called a Format Prefix (FP).

An IPv6 unicast address is divided into two parts. The first part contains the address prefix, and the second part contains the interface identifier. A concise way to express an IPv6 address/prefix combination is as follows: ipv6-address/prefix-length.

The following is an example of an address with a 64-bit prefix.

`3FFE:FFFF:0:CD30:0:0:0:0/64`.

The prefix in this example is `3FFE:FFFF:0:CD30`. The address can also be written in a compressed form, as `3FFE:FFFF:0:CD30::/64`.

IPv6 defines the following address types:

- **Unicast address**. An identifier for a single interface. A packet sent to this address is delivered to the identified interface. The unicast addresses are distinguished from the multicast addresses by the value of the high-order octet. The multicast addresses' high-order octet has the hexadecimal value of FF. Any other value for this octet identifies a unicast address. The following are different types of unicast addresses:
  - **Link-local addresses**. These addresses are used on a single link and have the following format:

FE80::*InterfaceID*. Link-local addresses are used between nodes on a link for auto-address configuration, neighbor discovery, or when no routers are present. A link-local address is used primarily at startup and when the system has not yet acquired addresses of larger scope.

  - **Site-local addresses**. These addresses are used on a single site and have the following format: FEC0::*SubnetID*:*InterfaceID*. The site-local addresses are used for addressing inside a site without the need for a global prefix.

  - **Global IPv6 unicast addresses**. These addresses can be used across the Internet and have the following format: 010(FP, 3 bits) TLA ID (13 bits) Reserved (8 bits) NLA ID (24 bits) SLA ID (16 bits) *InterfaceID* (64 bits).

- **Multicast address**. An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to this address is delivered to all the interfaces identified by the address. The multicast address types supersede the IPv4 broadcast addresses.

- **Anycast address**. An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to this address is delivered to only one interface identified by the address. This is the nearest interface as identified by routing metrics. Anycast addresses are taken from the unicast address space and are not syntactically distinguishable. The addressed interface performs the distinction between unicast and anycast addresses as a function of its configuration.

In general, a node always has a link-local address. It might have a site-local address and one or more global addresses.

See Also

Internet Protocol Version 6
Sockets

IPv6 Routing

7/29/2017 • 1 min to read • Edit Online

A flexible routing mechanism is a benefit of IPv6. Due to the way in which IPv4 network IDs were and are allocated, large routing tables need to be maintained by the routers that are on the Internet backbones. These routers must know all the routes in order to forward packets that are potentially directed to any node on the Internet. With its ability to aggregate addresses, IPv6 allows flexible addressing and drastically reduces the size of routing tables. In this new addressing architecture, intermediate routers must keep track only of the local portion of their network in order to forward the messages appropriately.

Neighbor Discovery

Some of the features provided by Neighbor Discovery are:

- Router discovery. This allows hosts to identify local routers.
- Address resolution. This allows nodes to resolve a link-layer address for a corresponding next-hop address (a replacement for Address Resolution Protocol [ARP]).
- Address auto-configuration. This allows hosts to automatically configure site-local and global addresses.

Neighbor Discovery uses Internet Control Message Protocol for IPv6 (ICMPv6) messages that include:

- Router advertisement. Sent by a router on a pseudo-periodic basis or in response to a router solicitation. IPv6 routers use router advertisements to advertise their availability, address prefixes, and other parameters.
- Router solicitation. Sent by a host to request that routers on the link send a router advertisement immediately.
- Neighbor solicitation. Sent by nodes for address resolution, duplicate address detection, or to verify that a neighbor is still reachable.
- Neighbor advertisement. Sent by nodes to respond to a neighbor solicitation or to notify neighbors of a change in link-layer address.
- Redirect. Sent by routers to indicate a better next-hop address to a particular destination for a sending node.

See Also

Internet Protocol Version 6
Sockets

IPv6 Auto-Configuration

7/29/2017 • 1 min to read • Edit Online

One important goal for IPv6 is to support node Plug and Play. That is, it should be possible to plug a node into an IPv6 network and have it automatically configured without any human intervention.

Type of Auto-Configuration

IPv6 supports the following types of auto-configuration:

- **Stateful auto-configuration**. This type of configuration requires a certain level of human intervention because it needs a Dynamic Host Configuration Protocol for IPv6 (DHCPv6) server for the installation and administration of the nodes. The DHCPv6 server keeps a list of nodes to which it supplies configuration information. It also maintains state information so the server knows how long each address is in use, and when it might be available for reassignment.
- **Stateless auto-configuration**. This type of configuration is suitable for small organizations and individuals. In this case, each host determines its addresses from the contents of received router advertisements. Using the IEEE EUI-64 standard to define the network ID portion of the address, it is reasonable to assume the uniqueness of the host address on the link.

Regardless of how the address is determined, the node must verify that its potential address is unique to the local link. This is done by sending a neighbor solicitation message to the potential address. If the node receives any response, it knows that the address is already in use and must determine another address.

IPv6 Mobility

The proliferation of mobile devices has introduced a new requirement: A device must be able to arbitrarily change locations on the IPv6 Internet and still maintain existing connections. To provide this functionality, a mobile node is assigned a home address at which it can always be reached. When the mobile node is at home, it connects to the home link and uses its home address. When the mobile node is away from home, a home agent, which is usually a router, relays messages between the mobile node and nodes with which it is communicating.

See Also

Internet Protocol Version 6
Sockets

Enabling and Disabling IPv6

7/29/2017 • 1 min to read • Edit Online

To use the IPv6 protocol, ensure that you are running a version of the operating system that supports IPv6 and ensure that the operating system and the networking classes are configured properly.

Configuration Steps

The following table lists various configurations

OPERATING SYSTEM IPV6-ENABLED?	NETWORKING CLASSES IPV6-ENABLED?	DESCRIPTION
No	No	Can parse IPv6 addresses.
No	Yes	Can parse IPv6 addresses.
Yes	No	Can parse IPv6 addresses and resolve IPv6 addresses using name resolution methods not marked obsolete.
Yes	Yes	Can parse and resolve IPv6 addresses using all methods including those marked obsolete.

Be aware that to enable the IPv6 support for all classes in the System.Net namespace, you must modify the computer configuration file or the configuration file for the application. The configuration file for an application has precedence over the computer configuration file.

For an example of how to modify the computer configuration file, *machine.config*, to enable Ipv6 support see, How to: Modify the Computer Configuration File to Enable Ipv6 Support. Also, ensure that the IPv6 support is enabled for the operating system.

The .NET Framework has a configuration switch set in a configuration file as follows

```
<system.net>…
    <settings>…
        <ipv6 enabled="true"/>…
    </settings>…
</system.net>
```

For .NET Framework version 1.1 and earlier, the value of the **ipv6 enabled** configuration switch specifies whether members of the System.Net.Dns class return IPv6 addresses.

For .NET Framework version 2.0 and later, if Windows supports IPv6, then members of the System.Net.Dns class, (for example, the System.Net.Dns.GetHostEntry method), will return IPv6 addresses with one limitation. Obsolete members of the DNS System.Net.Dns (for example, the System.Net.Dns.Resolve method) will read and recognize the value in the configuration file for the ipv6 enabled setting.

See Also

Internet Protocol Version 6
Sockets

How to: Modify the Computer Configuration File to Enable IPv6 Support

7/29/2017 • 1 min to read • Edit Online

The following code example shows how to modify the computer configuration file, *machine.config*, to enable IPv6 support. The *machine.config* file is stored in the *%Windir%\Microsoft.NET\Framework* folder in the directory where Windows was installed. There is a separate *machine.config* file in the folders under *%Windir%\Microsoft.NET\Framework* for each version of the .NET Framework installed on the computer (for example, *C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\machine.config*).

These settings can also be made in the configuration file for the application, which has precedence over the computer configuration file.

For .NET Framework version 1.1 and earlier, the value of the **ipv6 enabled** configuration switch specifies whether members of the System.Net.Dns class return IPv6 addresses.

For .NET Framework version 2.0 and later, if Windows supports IPv6, then all members of the System.Net.Dns class (for example, the System.Net.Dns.GetHostEntry method), will return IPv6 addresses with one limitation. Obsolete members of the System.Net.Dns class (for example, the System.Net.Dns.Resolve method) will read and recognize the value in the configuration file.

> **NOTE**
> For .NET Framework version 2.0 and later, IPv6 is enabled by default. For .NET Framework version 1.1 and earlier, IPv6 is disabled by default.

Example

```
<system.net>
    ……
    <settings>
        ……
        <ipv6 enabled="true"/>
    ……
    </settings>
    ……
<system.net>
```

See Also

IPv6 Addressing
Network Settings Schema
<ipv6> Element (Network Settings)

Configuring Internet Applications

7/29/2017 • 1 min to read • Edit Online

The <system.Net> Element (Network Settings) configuration element contains network configuration information for applications. Using the <system.Net> Element (Network Settings) element, you can set proxy servers, set connection management parameters, and include custom authentication and request modules in your application.

The <defaultProxy> Element (Network Settings) element defines the proxy server returned by the `GlobalProxySelection` class. Any HttpWebRequest that does not have its own Proxy property set to a specific value uses the default proxy. In addition to setting the proxy address, you can create a list of server addresses that will not use the proxy, and you can indicate that the proxy should not be used for local addresses.

It is important to note that the Microsoft Internet Explorer settings are combined with the configuration settings, with the latter taking precedence.

The following example sets the default proxy server address to http://proxyserver, indicates that the proxy should not be used for local addresses, and specifies that all requests to servers located in the contoso.com domain should bypass the proxy.

```
<configuration>
    <system.net>
        <defaultProxy>
            <proxy
                usesystemdefault = "false"
                proxyaddress = "http://proxyserver:80"
                bypassonlocal = "true"
            />
            <bypasslist>
                <add address="http://[a-z]+\.contoso\.com/" />
            </bypasslist>
        </defaultProxy>
    </system.net>
</configuration>
```

Use the <connectionManagement> Element (Network Settings) element to configure the number of persistent connections that can be made to a specific server or to all other servers. The following example configures the application to use two persistent connections to the server www.contoso.com, four persistent connections to the server with the IP address 192.168.1.2, and one persistent connection to all other servers.

```
<configuration>
    <system.net>
        <connectionManagement>
            <add address="http://www.contoso.com" maxconnection="2" />
            <add address="192.168.1.2" maxconnection="4" />
            <add address="*" maxconnection="1" />
        </connectionManagement>
    </system.net>
</configuration>
```

Custom authentication modules are configured with the <authenticationModules> Element (Network Settings) element. Custom authentication modules must implement the IAuthenticationModule interface.

The following example configures a custom authentication module.

```
<configuration>
    <system.net>
        <authenticationModules>
            <add type="MyAuthModule, MyAuthModule.dll" />
        </authenticationModules>
    </system.net>
</configuration>
```

You can use the <webRequestModules> Element (Network Settings) element to configure your application to use custom protocol-specific modules to request information from Internet resources. The specified modules must implement the IWebRequestCreate interface. You can override the default HTTP, HTTPS, and file request modules by specifying your custom module in the configuration file, as in the following example.

```
<configuration>
    <system.net>
        <webRequestModules>
            <add
                prefix="HTTP"
                type = "MyHttpRequest.dll, MyHttpRequestCreator"
            />
        </webRequestModules>
    </system.net>
</configuration>
```

See Also

Network Programming in the .NET Framework
Network Settings Schema
<system.Net> Element (Network Settings)

Network Tracing in the .NET Framework

7/29/2017 • 1 min to read • Edit Online

Network tracing in the .NET Framework provides access to information about method invocations and network traffic generated by a managed application. This feature is useful for debugging applications under development as well as for analyzing deployed applications. The output provided by network tracing is customizable to support different usage scenarios at development time and in a production environment.

To enable network tracing in the .NET Framework, you must select a destination for tracing output and add network tracing configuration settings to either the application or machine configuration file. For descriptions of configuration files and how they are used, see Configuration Files. For information about how to enable network tracing, see Enabling Network Tracing. For information about the settings that you need to add to the configuration file, see How to: Configure Network Tracing.

When tracing is enabled, you can capture trace information that is output by **System.Net** classes. Networking class members that generate tracing information include the following note in the Remarks section of their NET Framework class library documentation:

> **NOTE**
>
> This member outputs trace information when you enable network tracing in your application. For more information, see Network Tracing.

See Also

Enabling Network Tracing
How to: Configure Network Tracing
Interpreting Network Tracing
Introduction to Instrumentation and Tracing

Interpreting Network Tracing

7/29/2017 • 1 min to read • Edit Online

When network tracing is enabled, you can use tracing to capture calls your application makes to various System.Net class members. The output from these calls may be similar to the following examples.

```
[588]   (4357)   Entering Socket#33574638::Send()
[588]   (4387)   Exiting Socket#33574638::Send()-> 61#61
```

In the preceding example, [588] is the current thread's unique identifier. (4357) and (4387) are timestamps denoting the number of milliseconds that have elapsed since the application started. The data following the timestamp shows the application entering and exiting the method **Socket.Send**. The object executing the **Send** method has 33574638 as its unique identifier. The method exit trace includes the return value (61 in the preceding example).

Network traces can capture network traffic that is sent from or received by your application using application-level protocols such as Hypertext Transfer Protocol (HTTP). This data can be captured as text and, optionally, hexadecimal data. Hexadecimal data is available when you specify **includehex** as the value of the **tracemode** attribute. (For detailed information about this attribute, see How to: Configure Network Tracing.) The following example trace was generated using **includehex**.

```
[1692] (1142) 00000000 : 47 45 54 20 2F 77 70 61-64 2E 64 61 74 20 48 54 : GET /wpad.dat HT
```

```
[1692] (1142) 00000010 : 54 50 2F 31 2E 31 0D 0A-48 6F 73 74 3A 20 69 74 : TP/1.1..Host: it
```

```
[1692] (1142) 00000020 : 67 70 72 6F 78 79 0D 0A-43 6F 6E 6E 65 63 74 69 : gproxy..Connecti
```

```
[1692] (1142) 00000030 : 6F 6E 3A 20 43 6C 6F 73-65 0D 0A 0D 0A : on: Close....
```

To omit hexadecimal data, specify **protocolonly** as the value for the **tracemode** attribute. The following example shows the trace when **protocolonly** is specified.

```
[2444] (594) Data from ConnectStream#33574638::WriteHeaders<<GET /wpad.dat HTTP/1.1
```

```
Host: itgproxy
```

```
Connection: Close
```

See Also

Enabling Network Tracing
How to: Configure Network Tracing
Network Tracing in the .NET Framework

Enabling Network Tracing

7/29/2017 • 1 min to read • Edit Online

Network tracing provides access to information about method invocations and network traffic generated by a managed application. You must complete the following tasks to enable network tracing in your application:

- Compile your code with tracing enabled. See How to: Compile Conditionally with Trace and Debug for more information about the compiler switches required to enable tracing.
- Specify a destination for tracing output.
- Configure the behavior of network tracing. See How to: Configure Network Tracing for detailed information.

The most common trace destinations, also referred to as trace listeners, are the default listener and the log file.

Tracing uses the default listener if you do not specify a trace listener. You can view messages sent to the default listener by executing your code in a managed code-enabled debugger such as the CLR Debugger shipped with the .NET Framework SDK, or DBwin32.exe shipped with the Windows SDK. Using the CLR Debugger, the trace messages appear in the **Output** window.

If you prefer to use a file to receive traces, you can specify a log file by using configuration settings, as shown in the following example. (For a general discussion of configuration files, see Configuration Files.)

To send traces to a log file, add the following node to the `<system.diagnostics>` node of the appropriate configuration file (application or machine). You can change the name of the file (trace.log) to suit your needs.

```
<system.diagnostics>
  <trace autoflush="true" indentsize="4">
    <listeners>
      <add name="file" type="System.Diagnostics.TextWriterTraceListener" initializeData="trace.log"/>
    </listeners>
  </trace>
</system.diagnostics>
```

See Also

Interpreting Network Tracing
Network Tracing in the .NET Framework
Introduction to Instrumentation and Tracing

How to: Configure Network Tracing

7/29/2017 • 2 min to read • Edit Online

The application or computer configuration file holds the settings that determine the format and content of network traces. Before performing this procedure, be sure tracing is enabled. For information about enabling tracing, see Enabling Network Tracing.

The computer configuration file, machine.config, is stored in the %Windir%\Microsoft.NET\Framework folder in the directory where Windows was installed. There is a separate machine.config file in the folders under %Windir%\Microsoft.NET\Framework for each version of the .NET Framework installed on the computer (for example, C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\machine.config).

These settings can also be made in the configuration file for the application, which has precedence over the computer configuration file.

To configure network tracing

- Add the following lines to the appropriate configuration file. The values and options for these settings are described in the tables below.

```
<configuration>
  <system.diagnostics>
    <sources>
      <source name="System.Net" tracemode="includehex" maxdatasize="1024">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
      <source name="System.Net.Cache">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
      <source name="System.Net.Http">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
      <source name="System.Net.Sockets">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
      <source name="System.Net.WebSockets">
        <listeners>
          <add name="System.Net"/>
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="System.Net" value="Verbose"/>
      <add name="System.Net.Cache" value="Verbose"/>
      <add name="System.Net.Http" value="Verbose"/>
      <add name="System.Net.Sockets" value="Verbose"/>
      <add name="System.Net.WebSockets" value="Verbose"/>
    </switches>
    <sharedListeners>
      <add name="System.Net"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="network.log"
      />
    </sharedListeners>
    <trace autoflush="true"/>
  </system.diagnostics>
</configuration>
```

When you add a name to the `<switches>` block, the trace output includes information from some methods related to the name. The following table describes the output.

NAME	OUTPUT FROM
`System.Net.Sockets`	Some public methods of the Socket, TcpListener, TcpClient, and Dns classes
`System.Net`	Some public methods of the HttpWebRequest, HttpWebResponse, FtpWebRequest, and FtpWebResponse classes, and SSL debug information (invalid certificates, missing issuers list, and client certificate errors.)
`System.Net.HttpListener`	Some public methods of the HttpListener, HttpListenerRequest, and HttpListenerResponse classes.

NAME	OUTPUT FROM
`System.Net.Cache`	Some private and internal methods in `System.Net.Cache`.
`System.Net.Http`	Some public methods of the HttpClient, DelegatingHandler, HttpClientHandler, HttpMessageHandler, MessageProcessingHandler, and WebRequestHandler classes.
`System.Net.WebSockets.WebSocket`	Some public methods of the ClientWebSocket and WebSocket classes.

The attributes listed in the following table configure trace output.

ATTRIBUTE NAME	ATTRIBUTE VALUE
`Value`	Required String attribute. Sets the verbosity of the output. Legitimate values are `Critical`, `Error`, `Verbose`, `Warning`, and `Information`. This attribute must be set on the <add name> element of the <switches> element as shown in the example. An exception is thrown if this attribute is set on the <source> element.
`maxdatasize`	Optional Int32 attribute. Sets the maximum number of bytes of network data included in each line trace. The default value is 1024. This attribute must be set on the <source> element as shown in the example. An exception is thrown if this attribute is set on an element under the <switches> element.
`Tracemode`	Optional String attribute. Set to `includehex` to show protocol traces in hexadecimal and text format. Set to `protocolonly` to show only text. The default value is `includehex`. This attribute must be set on the <switches> element as shown in the example. An exception is thrown if this attribute is set on an element under the <source> element.

See Also

Interpreting Network Tracing
Network Tracing in the .NET Framework
Enabling Network Tracing
Introduction to Instrumentation and Tracing

Cache Management for Network Applications

7/29/2017 • 1 min to read • Edit Online

This topic and its related subtopics describe caching for resources obtained using the WebClient, WebRequest, HttpWebRequest, and FtpWebRequest classes.

A cache provides temporary storage of resources that have been requested by an application. If an application requests the same resource more than once, the resource can be returned from the cache, avoiding the overhead of re-requesting it from the server. Caching can improve application performance by reducing the time required to get a requested resource. Caching can also decrease network traffic by reducing the number of trips to the server. While caching improves performance, it increases the risk that the resource returned to the application is stale, meaning that it is not identical to the resource that would have been sent by the server if caching were not in use.

Caching may allow unauthorized users or processes to read sensitive data. An authenticated response that is cached may be retrieved from the cache without an additional authorization. If caching is enabled, change to CachePolicy to BypassCache or NoCacheNoStore to disable caching for this request.

Due to security concerns, caching is **not** recommended for middle tier scenarios.

In This Section

Cache Policy
Explains what a cache policy is and how to define one.

Location-Based Cache Policies
Defines each type of location-based cache policy available for Hypertext Transfer Protocol (http and https) resources.

Time-Based Cache Policies
Describes the criteria that can be used to customize a time-based cache policy.

Configuring Caching in Network Applications
Describes how to programmatically create cache policies and requests that use caching.

Reference

System.Net.Cache
Defines the types and enumerations used to define cache policies for resources obtained using the WebRequest, HttpWebRequest, and FtpWebRequest classes.

Cache Policy

7/29/2017 • 1 min to read • Edit Online

A cache policy defines rules that are used to determine whether a request can be satisfied using a cached copy of the requested resource. Applications specify client cache requirements for freshness, but the effective cache policy is determined by the client cache requirements, the server's content expiration requirements, and the server's revalidation requirements. The interaction of client cache policy and server requirements always results in the most conservative cache policy, to help ensure that the freshest content is returned to the client application.

Cache policies are either location-based or time-based. A location-based cache policy defines the freshness of cached entries based on where the requested resource can be taken from. A time-based cache policy defines the freshness of cached entries using the time the resource was retrieved, headers returned with the resource, and the current time. Most applications can use the default time-based cache policy, which implements the caching policy specified in RFC 2616, available at http://www.ietf.org.

The classes described in the following table are used to specify cache policies.

CLASS NAME	DESCRIPTION
HttpRequestCachePolicy	Represents location-based and time-based cache policies for resources requested using HttpWebRequest objects.
RequestCachePolicy	Represents location-based cache policies or the Default time-based cache policy for resources requested using WebRequest objects.
HttpCacheAgeControl	Specifies values used to create time-based HttpRequestCachePolicy objects.
HttpRequestCacheLevel	Specifies values used to create location-based and time-based HttpRequestCachePolicy objects.
RequestCacheLevel	Specifies values used to create location-based or the Default time-based RequestCachePolicy objects.

You can define a cache policy for all requests made by your application or for individual requests. When you specify both an application-level cache policy and a request-level cache policy, the request-level policy is used. You can specify an application-level cache policy programmatically or by using the application or machine configuration files. For more information, see <requestCaching> Element (Network Settings).

To create a cache policy, you must create a policy object by creating an instance of the RequestCachePolicy or HttpRequestCachePolicy class. To specify the policy on a request, set the request's CachePolicy property to the policy object. When setting an application-level policy programmatically, set the DefaultCachePolicy property to the policy object.

For code examples that demonstrate creating and using cache policies, see Configuring Caching in Network Applications.

See Also

Cache Management for Network Applications
Location-Based Cache Policies

Location-Based Cache Policies

7/29/2017 • 2 min to read • Edit Online

A location-based cache policy defines the freshness of valid cached entries based on where the requested resource can be taken from. A cached resource is valid if using it does not does not violate server-specified revalidation requirements. A location-based cache policy is created programmatically by using a RequestCachePolicy or HttpRequestCachePolicy class constructor. The type of location-based policy is passed to the constructor using a RequestCacheLevel or HttpRequestCacheLevel enumeration value. For code examples that create location-based cache policies, see How to: Set a Location-Based Cache Policy for an Application. The following sections explain each type of location-based cache policy for Hypertext Transfer Protocol (http and https) resources.

Cache If Available Policy

If a valid requested resource is in the local cache, the cached resource is used; otherwise, the request for the resource is sent to the server. If the requested resource is available in any cache between the client and the server, the request can be satisfied by an intermediate cache.

Cache Only Policy

If a valid requested resource is in the local cache, the cached resource is used. When this cache policy level is specified, a WebException exception is thrown if the item is not in the local cache.

Cache Or Next Cache Only Policy

If a valid requested resource is in the local cache or an intermediate cache on the local area network, the cached resource is used. Otherwise, a WebException exception is thrown. In the HTTP caching protocol, this is achieved using the only-if-cached cache control directive.

No Cache No Store Policy

A requested resource is never used from any cache and is never placed in any cache. If a requested resource is present in the local cache, it is removed. This policy level indicates to intermediate caches that they should also remove the resource. In the HTTP caching protocol, this is achieved using the no-store cache control directive.

Refresh Policy

A requested resource can be used if it is obtained from the server or found in a cache other than the local cache. Before the request can be satisfied by an intermediate cache, that cache must revalidate its cached entry with the server. In the HTTP caching protocol, this is achieved using the max-age = 0 cache control directive and the no-cache Pragma header.

Reload Policy

Requested resources must be obtained from the server. The response might be saved in the local cache. In the HTTP caching protocol, this is achieved using the no-cache cache control directive and the no-cache Pragma header.

Revalidate Policy

Compares the copy of the resource in the cache with the copy on the server. If the copy on the server is newer, it is used to satisfy the request and replaces the copy in the cache. If the copy in the cache is the same as the server copy, the cached copy is used. In the HTTP caching protocol, this is achieved using a conditional request.

See Also

Cache Management for Network Applications
Cache Policy
Time-Based Cache Policies
Configuring Caching in Network Applications
<requestCaching> Element (Network Settings)

Time-Based Cache Policies

7/29/2017 • 3 min to read • Edit Online

A time-based cache policy defines the freshness of cached entries using the time the resource was retrieved, the headers returned with the resource, and the current time. When setting a time-based cache policy, you can either use the Default time-based policy or create a customized time-based policy. When using the default time-based policy for resources obtained using Hypertext Transfer Protocol (HTTP), the exact cache behavior is determined by the headers included in the cached response and by the behaviors specified in sections 13 and 14 of RFC 2616, available at http://www.ietf.org. For a code example that demonstrates setting the default time-based policy for HTTP resources, see How to: Set the Default Time-Based Cache Policy for an Application. For code examples that demonstrate creating and using cache policies, see Configuring Caching in Network Applications.

Criteria to Determine Freshness of Cached Entries

To customize a time-based cache policy, you can specify that one or more of the following criteria be used to determine the freshness of cached entries:

- Maximum age
- Maximum staleness
- Minimum freshness
- Cache synchronization date

> **NOTE**
> Using the default time-based cache policy should not be confused with setting a default cache policy for your application. The default time-based policy is a specific policy that can be used at the request or application level. The default cache policy for your application is a policy (location-based or time-based) that takes effect when no policy is set on a request. For details on setting a default cache policy for your application, see DefaultCachePolicy.

Maximum Age

The maximum age policy criterion specifies the amount of time a cached copy of a resource can be used. If the cached copy of the resource is older than the amount of time specified, the resource must be revalidated by checking it against the content on the server. If the maximum age would allow the resource to be used after it expires, this criteria is not honored unless a maximum staleness value is also specified.

Maximum Staleness

The maximum staleness policy criterion specifies the length of time after content expiration that the cached copy of the resource can be used. This is the only cache policy criterion that permits resources to be used after they have expired.

Minimum Freshness

The minimum freshness policy criterion specifies the length of time before content expiration that the cached copy of the resource can be used. This policy has the effect of causing a cache entry to expire before its expiration date; therefore, the minimum freshness and maximum staleness settings are mutually exclusive.

Cache Synchronization Date

The cache synchronization date policy criterion determines when a cached copy of a resource must be revalidated

by checking it against the content on the server. If the content has changed since the item was cached, it is retrieved from the server, stored in the cache, and returned to the application. If the content has not changed, its timestamp is updated and the application gets the cached content.

The cache synchronization date allows you to specify an absolute date when cached contents must be revalidated. If a fresh cache entry was last revalidated prior to the cache synchronization date, revalidation with the server still occurs. If the cache entry was revalidated after the cache synchronization date and there are no additional freshness or server revalidation requirements that invalidate the cached entry, the entry from the cache is used. If the cache synchronization date is set to a future date, the entry is revalidated every time it is requested, until the cache synchronization date passes.

The following topics provide information about the effects of combining time-based cache policy criteria:

- Cache Policy Interaction—Maximum Age and Maximum Staleness
- Cache Policy Interaction—Maximum Age and Minimum Freshness

See Also

Cache Management for Network Applications
Cache Policy
Location-Based Cache Policies
Configuring Caching in Network Applications
<requestCaching> Element (Network Settings)

Cache Policy Interaction—Maximum Age and Maximum Staleness

7/29/2017 • 1 min to read • Edit Online

To help ensure that the freshest content is returned to the client application, the interaction of client cache policy and server revalidation requirements always results in the most conservative cache policy. All the examples in this topic illustrate the cache policy for a resource that is cached on January 1 and expires on January 4.

In the following examples, the maximum staleness value (`maxStale`) is used in conjunction with a maximum age (`maxAge`):

- If the cache policy sets `maxAge` = 5 days and does not specify a `maxStale` value, according to the `maxAge` value, the content is usable until January 6. However, according to the server's revalidation requirements, the content expires on January 4. Because the content expiration date is more conservative (sooner), it takes precedence over the `maxAge` policy. Therefore, the content expires on January 4 and must be revalidated even though its maximum age has not been reached.
- If the cache policy sets `maxAge` = 5 days and `maxStale` = 3 days, according to the `maxAge` value, the content is usable until January 6. According to the `maxStale` value, the content is usable until January 7. Therefore, the content gets revalidated on January 6.
- If the cache policy sets `maxAge` = 5 days and `maxStale` = 1 day, according to the `maxAge` value, the content is usable until January 6. According to the `maxStale` value, the content is usable until January 5. Therefore, the content gets revalidated on January 5.

When the maximum age is less than the content expiration date, the more conservative caching behavior always prevails and the maximum staleness value has no effect. The following examples illustrate the effect of setting a maximum staleness (`maxStale`) value when the maximum age (`maxAge`) is reached before the content expires:

- If the cache policy sets `maxAge` = 1 day and does not specify a value for `maxStale` value, the content is revalidated on January 2 even though it has not expired.
- If the cache policy sets `maxAge` = 1 day and `maxStale` = 3 days, the content is revalidated on January 2 to enforce the more conservative policy setting.
- If the cache policy sets `maxAge` = 1 day and `maxStale` = 1 day, the content is revalidated on January 2.

See Also

Cache Management for Network Applications
Cache Policy
Location-Based Cache Policies
Time-Based Cache Policies
Configuring Caching in Network Applications
Cache Policy Interaction—Maximum Age and Minimum Freshness

Cache Policy Interaction—Maximum Age and Minimum Freshness

7/29/2017 • 1 min to read • Edit Online

To help ensure that the freshest content is returned to the client application, the interaction of client cache policy and server revalidation requirements always results in the most conservative cache policy. All the examples in this topic illustrate the cache policy for a resource that is cached on January 1 and expires on January 4.

The following examples illustrate the cache policy that results from the interaction of the maximum age (`maxAge`) and minimum freshness (`minFresh`) values.

- If the cache policy sets `maxAge` = 2 days and `minFresh` is not specified, the content is revalidated on January 3.
- If the cache policy sets `maxAge` = 2 days and `minFresh` = 1 day, according to `maxAge` , the content is fresh until January 3. According to `minFresh` , the content is fresh until January 3. Therefore, the content must be revalidated on January 3.
- If the cache policy sets `maxAge` = 2 days and `minFresh` = 2 days, according to `maxAge` , the content is fresh until January 3. According to `minFresh` the content is fresh until January 2. Therefore, the content must be revalidated on January 2.

See Also

Cache Management for Network Applications
Cache Policy
Location-Based Cache Policies
Time-Based Cache Policies
Configuring Caching in Network Applications
Cache Policy Interaction—Maximum Age and Maximum Staleness

Configuring Caching in Network Applications

7/29/2017 • 1 min to read • Edit Online

To configure caching, you must specify a cache policy at the application or WebRequest level. The following topics provide code examples that demonstrate configuring applications and requests to use caching.

- How to: Set a Location-Based Cache Policy for an Application
- How to: Set the Default Time-Based Cache Policy for an Application
- How to: Customize a Time-Based Cache Policy
- How to: Set Cache Policy for a Request

You can also configure cache policy using application or machine configuration files. For more information, see | <requestCaching> Element (Network Settings).

See Also

Cache Management for Network Applications
Cache Policy
Location-Based Cache Policies
Time-Based Cache Policies

How to: Set a Location-Based Cache Policy for an Application

7/29/2017 • 2 min to read • Edit Online

Location-based cache policies allow an application to explicitly define caching behavior based on the location of the requested resource. This topic demonstrates setting the cache policy programmatically. For information on setting the policy for an application using the configuration files, see <requestCaching> Element (Network Settings).

To set a location-based cache policy for an application

1. Create a RequestCachePolicy or HttpRequestCachePolicy object.
2. Set the policy object as the default for the application domain.

To set a policy that takes requested resources from a cache

- Create a policy that takes requested resources from a cache if available, and otherwise, sends requests to the server, by setting the cache level to CacheIfAvailable. A request can be fulfilled by any cache between the client and server, including remote caches.

```
public static void UseCacheIfAvailable()
{
    HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.CacheIfAvailable);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

```
Public Shared Sub UseCacheIfAvailable()
    Dim policy As New HttpRequestCachePolicy _
         (HttpRequestCacheLevel.CacheIfAvailable)
    HttpWebRequest.DefaultCachePolicy = policy
End Sub
```

To set a policy that prevents any cache from supplying resources

- Create a policy that prevents any cache from supplying requested resources by setting the cache level to NoCacheNoStore. This policy level removes the resource from the local cache if it is present and indicates to remote caches that they should also remove the resource.

```
public static void DoNotUseCache()
{
HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.NoCacheNoStore);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

```
Public Shared Sub DoNotUseCache()
    Dim policy As New HttpRequestCachePolicy _
        (HttpRequestCacheLevel.NoCacheNoStore)
    HttpWebRequest.DefaultCachePolicy = policy
End Sub
```

To set a policy that returns requested resources only if they are in the local cache

- Create a policy that returns requested resources only if they are in the local cache by setting the cache level to CacheOnly. If the requested resource is not in the cache, a WebException exception is thrown.

```
public static void OnlyUseCache()
{
    HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.CacheOnly);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

```
Public Shared Sub OnlyUseCache()
    Dim policy As New HttpRequestCachePolicy _
        (HttpRequestCacheLevel.CacheOnly)
    HttpWebRequest.DefaultCachePolicy = policy
End Sub
```

To set a policy that prevents the local cache from supplying resources

- Create a policy that prevents the local cache from supplying requested resources by setting the cache level to Refresh. If the requested resource is in an intermediate cache and is successfully revalidated, the intermediate cache can supply the requested resource.

```
public static void DoNotUseLocalCache()
{
 HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.Refresh);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

```
Public Shared Sub DoNotUseLocalCache()
    Dim policy As New HttpRequestCachePolicy _
        (HttpRequestCacheLevel.Refresh)
    HttpWebRequest.DefaultCachePolicy = policy
End Sub
```

To set a policy that prevents any cache from supplying requested resources

- Create a policy that prevents any cache from supplying requested resources by setting the cache level to Reload. The resource returned by the server can be stored in the cache.

```
public static void SendToServer()
{
HttpRequestCachePolicy policy = new HttpRequestCachePolicy
        (HttpRequestCacheLevel.Reload);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

```
Public Shared Sub SendToServer()
    Dim policy As New HttpRequestCachePolicy _
        (HttpRequestCacheLevel.Reload)
    HttpWebRequest.DefaultCachePolicy = policy
End Sub
```

To set a policy that allows any cache to supply requested resources if the resource on the server is not newer than the cached copy

- Create a policy that allows any cache to supply requested resources if the resource on the server is not newer than the cached copy by setting the cache level to Revalidate.

```
public static void CheckServer()
{
HttpRequestCachePolicy policy = new HttpRequestCachePolicy
         (HttpRequestCacheLevel.Revalidate);
    HttpWebRequest.DefaultCachePolicy = policy;
}
```

```
Public Shared Sub CheckServer()
    Dim policy As New HttpRequestCachePolicy _
        (HttpRequestCacheLevel.Revalidate)
    HttpWebRequest.DefaultCachePolicy = policy
End Sub
```

See Also

Cache Management for Network Applications
Cache Policy
Location-Based Cache Policies
Time-Based Cache Policies
<requestCaching> Element (Network Settings)

How to: Set the Default Time-Based Cache Policy for an Application

7/29/2017 • 1 min to read • Edit Online

The default time-based cache policy allows an application to have its cache behavior defined by the headers sent with the cached resource and the cache behavior defined in sections 13 and 14 of RFC 2616, available at http://www.ietf.org. This is the appropriate cache behavior for most applications.

To set the default automatic policy for an application

1. Create a default time-based policy object.
2. Set the policy object as the default for the application domain.

Example

The two examples in this section produce identical policies.

The following example creates a default time-based policy and sets it as the default for the application domain.

```
public static void SetDefaultTimeBasedPolicy ()
{
    HttpRequestCachePolicy policy = new HttpRequestCachePolicy ();
    HttpWebRequest.DefaultCachePolicy = policy ;
}
```

```
Public Shared Sub SetDefaultTimeBasedPolicy ()
    Dim policy = New HttpRequestCachePolicy ()
    HttpWebRequest.DefaultCachePolicy = policy
End Sub
```

You can also create the default time-based cache policy using the RequestCachePolicy class as shown in the following example:

```
public static void SetDefaultTimeBasedPolicy2()
{
    RequestCachePolicy policy = new RequestCachePolicy ();
    HttpWebRequest.DefaultCachePolicy = policy ;
}
```

```
Public Shared Sub SetDefaultTimeBasedPolicy2()
    Dim policy As New RequestCachePolicy()
    HttpWebRequest.DefaultCachePolicy = policy
End Sub
```

See Also

Cache Management for Network Applications
Cache Policy
Location-Based Cache Policies

Time-Based Cache Policies
<requestCaching> Element (Network Settings)

How to: Customize a Time-Based Cache Policy

7/29/2017 • 1 min to read • Edit Online

When creating a time-based cache policy, you can customize caching behavior by specifying values for maximum age, minimum freshness, maximum staleness, or cache synchronization date. The HttpRequestCachePolicy object provides several constructors that allow you to specify valid combinations of these values.

To create a time-based cache policy that uses a cache synchronization date

- Create a time-based cache policy that uses a cache synchronization date by passing a DateTime object to the HttpRequestCachePolicy constructor.

```
public static HttpRequestCachePolicy CreateLastSyncPolicy(DateTime when)
{
    HttpRequestCachePolicy policy =
        new HttpRequestCachePolicy(when);
    Console.WriteLine("When: {0}", when);
    Console.WriteLine(policy.ToString());
    return policy;
}
```

```
Public Shared Function CreateLastSyncPolicy([when] As DateTime) As HttpRequestCachePolicy
    Dim policy As New HttpRequestCachePolicy([when])
    Console.WriteLine("When: {0}", [when])
    Console.WriteLine(policy.ToString())
    Return policy
End Function
```

The output is similar to the following:

```
When: 1/14/2004 8:07:30 AM
Level:Default CacheSyncDate:1/14/2004 8:07:30 AM
```

To create a time-based cache policy that is based on minimum freshness

- Create a time-based cache policy that is based on minimum freshness by specifying MinFresh as the `cacheAgeControl` parameter value and passing a TimeSpan object to the HttpRequestCachePolicy constructor.

```
public static HttpRequestCachePolicy CreateMinFreshPolicy(TimeSpan span)
{
    HttpRequestCachePolicy policy =
        new HttpRequestCachePolicy(HttpCacheAgeControl.MinFresh, span);
    Console.WriteLine(policy.ToString());
    return policy;
}
```

```
Public Shared Function CreateMinFreshPolicy(span As TimeSpan) As HttpRequestCachePolicy
    Dim policy As New HttpRequestCachePolicy(HttpCacheAgeControl.MinFresh, span)
    Console.WriteLine(policy.ToString())
    Return policy
End Function
```

For the following invocation:

```
CreateMinFreshPolicy(new TimeSpan(1,0,0));
```

```
Level:Default MinFresh:3600
```

To create a time-based cache policy that is based on minimum freshness and maximum age

- Create a time-based cache policy that is based on minimum freshness and maximum age by specifying MaxAgeAndMinFresh as the `cacheAgeControl` parameter value and passing two TimeSpan objects to the HttpRequestCachePolicy constructor, one to specify the maximum age for resources and a second to specify the minimum freshness permitted for an object returned from the cache.

```
public static HttpRequestCachePolicy CreateFreshAndAgePolicy(TimeSpan freshMinimum, TimeSpan ageMaximum)
{
    HttpRequestCachePolicy policy =
    new HttpRequestCachePolicy(HttpCacheAgeControl.MaxAgeAndMinFresh, ageMaximum, freshMinimum);
    Console.WriteLine(policy.ToString());
    return policy;
}
```

```
Public Shared Function CreateFreshAndAgePolicy(freshMinimum As TimeSpan, ageMaximum As TimeSpan) As
HttpRequestCachePolicy
    Dim policy As New HttpRequestCachePolicy(HttpCacheAgeControl.MaxAgeAndMinFresh, ageMaximum,
freshMinimum)
    Console.WriteLine(policy.ToString())
    Return policy
End Function
```

For the following invocation:

```
CreateFreshAndAgePolicy(new TimeSpan(5,0,0), new TimeSpan(10,0,0));
```

```
Level:Default MaxAge:36000 MinFresh:18000
```

See Also

Cache Management for Network Applications
Cache Policy
Location-Based Cache Policies
Time-Based Cache Policies
<requestCaching> Element (Network Settings)

How to: Set Cache Policy for a Request

7/29/2017 • 2 min to read • Edit Online

The following example demonstrates setting a cache policy for a request. The example input is a URI such as http://www.contoso.com/.

Example

The following code example creates a cache policy that allows the requested resource to be used from the cache if it has not been in the cache for longer than one day. The example displays a message that indicates whether the resource was used from the cache—for example, `"The response was retrieved from the cache : False."` —and then displays the resource. A request can be fulfilled by any cache between the client and server.

```
using System;
using System.Net;
using System.Net.Cache;
using System.IO;

namespace Examples.System.Net.Cache
{
    public class CacheExample
    {
        public static void UseCacheForOneDay(Uri resource)
        {
            // Create a policy that allows items in the cache
            // to be used if they have been cached one day or less.
            HttpRequestCachePolicy requestPolicy =
                new HttpRequestCachePolicy (HttpCacheAgeControl.MaxAge,
                TimeSpan.FromDays(1));

            WebRequest request = WebRequest.Create (resource);
            // Set the policy for this request only.
            request.CachePolicy = requestPolicy;
            HttpWebResponse response = (HttpWebResponse)request.GetResponse();
            // Determine whether the response was retrieved from the cache.
            Console.WriteLine ("The response was retrieved from the cache : {0}.",
               response.IsFromCache);
            Stream s = response.GetResponseStream ();
            StreamReader reader = new StreamReader (s);
            // Display the requested resource.
            Console.WriteLine(reader.ReadToEnd());
            reader.Close ();
            s.Close();
            response.Close();
        }
        public static void Main(string[] args)
        {
            if (args == null || args.Length != 1)
            {
                Console.WriteLine ("You must specify the URI to retrieve.");
                return;
            }
            UseCacheForOneDay (new Uri(args[0]));
        }
    }
}
```

```
Imports System
Imports System.Net
Imports System.Net.Cache
Imports System.IO
Namespace Examples.System.Net.Cache
    Public Class CacheExample
        Public Shared Sub UseCacheForOneDay(ByVal resource As Uri)
            ' Create a policy that allows items in the cache
            ' to be used if they have been cached one day or less.
            Dim requestPolicy As New HttpRequestCachePolicy _
                  (HttpCacheAgeControl.MaxAge, TimeSpan.FromDays(1))
            Dim request As WebRequest = WebRequest.Create(resource)
            ' Set the policy for this request only.
            request.CachePolicy = requestPolicy
            Dim response As HttpWebResponse = _
             CType(request.GetResponse(), HttpWebResponse)
            ' Determine whether the response was retrieved from the cache.
            Console.WriteLine("The response was retrieved from the cache : {0}.", _
                response.IsFromCache)
            Dim s As Stream = response.GetResponseStream()
            Dim reader As New StreamReader(s)
            ' Display the requested resource.
            Console.WriteLine(reader.ReadToEnd())
            reader.Close()
            s.Close()
            response.Close()
        End Sub
        Public Shared Sub Main(ByVal args() As String)
            If args Is Nothing OrElse args.Length <> 1 Then
                Console.WriteLine("You must specify the URI to retrieve.")
                Return
            End If
            UseCacheForOneDay(New Uri(args(0)))
        End Sub
    End Class
End Namespace
```

See Also

Cache Management for Network Applications
Cache Policy
Location-Based Cache Policies
Time-Based Cache Policies
<requestCaching> Element (Network Settings)

Security in Network Programming

7/29/2017 • 1 min to read • Edit Online

The .NET Framework System.Net namespace classes provide built-in support for popular Internet application authentication mechanisms and for .NET Framework code access permissions.

In This Section

Using Secure Sockets Layer
Describes how to use Secure Sockets Layer (SSL) connections.

Internet Authentication
Describes how to use HTTP authentication methods to establish authenticated connections to HTTP servers.

Web and Socket Permissions
Describes how to set code access security for applications that use Internet connections.

Related Sections

Network Programming in the .NET Framework
Introduces the classes in the System.Net and System.Net.Sockets namespaces.

Using Secure Sockets Layer

7/29/2017 • 1 min to read • Edit Online

The System.Net classes use the Secure Sockets Layer (SSL) to encrypt the connection for several network protocols.

For http connections, the WebRequest and WebResponse classes use SSL to communicate with web hosts that support SSL. The decision to use SSL is made by the WebRequest class, based on the URI it is given. If the URI begins with "https:", SSL is used; if the URI begins with "http:", an unencrypted connection is used.

To use SSL with File Transfer Protocol (FTP), set the EnableSsl property to true prior to calling GetResponse(). Similarly, to use SSL with Simple Mail Transport Protocol (SMTP), set the EnableSsl property to true prior to sending the e-mail.

The SslStream class provides a stream-based abstraction for SSL, and offers many ways to configure the SSL handshake.

Example

Code

```
Dim MyURI As String = "https://www.contoso.com/"
Dim Wreq As WebRequest = WebRequest.Create(MyURI)

Dim serverUri As String = "ftp://ftp.contoso.com/file.txt"
Dim request As FtpWebRequest = CType(WebRequest.Create(serverUri), FtpWebRequest)
request.Method = WebRequestMethods.Ftp.DeleteFile
request.EnableSsl = True
Dim response As FtpWebResponse = CType(request.GetResponse(), FtpWebResponse)
```

```
String MyURI = "https://www.contoso.com/";
WebRequest WReq = WebRequest.Create(MyURI);

String serverUri = "ftp://ftp.contoso.com/file.txt"
FtpWebRequest request = (FtpWebRequest)WebRequest.Create(serverUri);
request.EnableSsl = true;
request.Method = WebRequestMethods.Ftp.DeleteFile;
FtpWebResponse response = (FtpWebResponse)request.GetResponse();
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

See Also

Security in Network Programming
Network Programming in the .NET Framework
Certificate Selection and Validation

Certificate Selection and Validation

7/29/2017 • 3 min to read • Edit Online

The System.Net classes support several ways to select and validate System.Security.Cryptography.X509Certificates for Secure Socket Layer (SSL) connections. A client can select one or more certificates to authenticate itself to a server. A server can require that a client certificate have one or more specific attributes for authentication.

Definition

A certificate is an ASCII byte stream that contains a public key, attributes (such as version number, serial number, and expiration date) and a digital signature from a Certificate Authority. Certificates are used to establish an encrypted connection or to authenticate a client to a server.

Client Certificate Selection and Validation

A client can select one or more certificates for a specific SSL connection. Client certificates can be associated with the SSL connection to a web server or an SMTP mail server. A client adds certificates to a collection of X509Certificate or X509Certificate2 class objects. Using email as an example, the certificate collection is an instance of a X509CertificateCollection) associated with the ClientCertificates property of the SmtpClient class. The HttpWebRequest class has a similar ClientCertificates property.

The primary difference between the X509Certificate and the X509Certificate2 class is that the private key must reside in the certificate store for the X509Certificate class.

Even if certificates are added to a collection and associated with a specific SSL connection, no certificates will be sent to the server unless the server requests them. If multiple client certificates are set on a connection, the best one will be used based on an algorithm that considers the match between the list of certificate issuers provided by the server and the client certificate issuer name.

The SslStream class provides even more control over the SSL handshake. A client can specify a delegate to pick which client certificate to use.

A remote server can verify that a client certificate is valid, current, and signed by the appropriate Certificate Authority. A delegate can be added to the ServerCertificateValidationCallback to enforce certificate validation.

Client Certificate Selection

The .NET Framework selects the client certificate to present to the server in the following manner:

1. If a client certificate was presented previously to the server, the certificate is cached when first presented and is reused for subsequent client certificate requests.

2. If a delegate is present, always use the result from the delegate as the client certificate to select. Try to use a cached certificate when possible, but do not use cached anonymous credentials if the delegate has returned null and the certificate collection is not empty.

3. If this is the first challenge for a client certificate, the Framework enumerates the certificates in X509Certificate or the X509Certificate2 class objects associated with the connection, looking for a match between the list of certificate issuers provided by the server and the client certificate issuer name. The first certificate that matches is sent to the server. If no certificate matches or the certificate collection is empty, then an anonymous credential is sent to the server.

Tools for Certificate Configuration

A number of tools are available for client and server certificate configuration.

The *Winhttpcertcfg.exe* tool can be used to configure client certificates. The *Winhttpcertcfg.exe* tool is provided as one of the tools with the Windows Server 2003 Resource Kit. This tool is also available as a download as part of the Windows Server 2003 Resource Kit Tools at www.microsoft.com.

The HttpCfg.exe tool can be used to configure server certificates for the HttpListener class. The *HttpCfg.exe* tool is provided as one of the support tools for Windows Server 2003 and Windows XP Service Pack 2. *HttpCfg.exe* and the other support tools are not installed by default on either Windows Server 2003 or Windows XP. On Windows Server 2003. the support tools are installed separately from the following folder and file on the Windows Server 2003 CD-ROM:

\Support\Tools\Suptools.msi

For use with Windows XP Service Pack 2, the Windows XP Support Tools are available as a download from www.microsoft.com.

The source code to a version of the *HttpCfg.exe* tool is also provided as a sample with the Windows Server SDK. The source code to the *HttpCfg.exe* sample is installed by default with the networking samples as part of the Windows SDK under the following folder:

C:\Program Files\Microsoft SDKs\Windows\v1.0\Samples\NetDS\http\serviceconfig

In addition to these tools, the X509Certificate and X509Certificate2 classes provides methods for loading a certificate from the file system.

See Also

Security in Network Programming
Network Programming in the .NET Framework

Internet Authentication

7/29/2017 • 2 min to read • Edit Online

The System.Net classes support a variety of client authentication mechanisms, including the standard Internet authentication methods basic, digest, negotiate, NTLM, and Kerberos authentication, as well as custom methods that you can create.

Authentication credentials are stored in the NetworkCredential and CredentialCache classes, which implement the ICredentials interface. When one of these classes is queried for credentials, it returns an instance of the **NetworkCredential** class. The authentication process is managed by the AuthenticationManager class, and the actual authentication process is performed by an authentication module class that implements the IAuthenticationModule interface. You must register a custom authentication module with the **AuthenticationManager** before it can be used; modules for the basic, digest, negotiate, NTLM, and Kerberos authentication methods are registered by default.

NetworkCredential stores a set of credentials associated with a single Internet resource identified by a URI and returns them in response to any call to the GetCredential method. The **NetworkCredential** class is typically used by applications that access a limited number of Internet resources or by applications that use the same set of credentials in all cases.

The **CredentialCache** class stores a collection of credentials for various Web resources. When the GetCredential method is called, **CredentialCache** returns the proper set of credentials, as determined by the URI of the Web resource and the requested authentication scheme. Applications that use a variety of Internet resources with different authentication schemes benefit from using the **CredentialCache** class, since it stores all the credentials and provides them as requested.

When an Internet resource requests authentication, the System.Net.WebRequest.GetResponse method sends the WebRequest to the **AuthenticationManager** along with the request for credentials. The request is then authenticated according to the following process:

1. The **AuthenticationManager** calls the Authenticate method on each of the registered authentication modules in the order they were registered. The **AuthenticationManager** uses the first module that does not return **null** to carry out the authentication process. The details of the process vary depending on the type of authentication module involved.

2. When the authentication process is complete, the authentication module returns an Authorization to the **WebRequest** that contains the information needed to access the Internet resource.

Some authentication schemes can authenticate a user without first making a request for a resource. An application can save time by preauthenticating the user with the resource, thus eliminating at least one round trip to the server. Or, it can perform authentication during program startup in order to be more responsive to the user later. Authentication schemes that can use preauthentication set the PreAuthenticate property to **true**.

See Also

Basic and Digest Authentication
NTLM and Kerberos Authentication
Security in Network Programming

Basic and Digest Authentication

7/29/2017 • 1 min to read • Edit Online

The System.Net implementation of basic and digest authentication complies with RFC2617 – HTTP Authentication: Basic and Digest Authentication (available on the World Wide Web Consortium's Web site at www.w3.org).

To use basic and digest authentication, an application must provide a user name and password in the Credentials property of the WebRequest object that it uses to request data from the Internet, as shown in the following example.

```
Dim MyURI As String = "http://www.contoso.com/"
Dim WReq As WebRequest = WebRequest.Create(MyURI)
WReq.Credentials = New NetworkCredential(UserName, SecurelyStoredPassword)
```

```
String MyURI = "http://www.contoso.com/";
WebRequest WReq = WebRequest.Create(MyURI);
WReq.Credentials = new NetworkCredential(UserName, SecurelyStoredPassword);
```

Caution

Data sent with Basic and Digest Authentication is not encrypted, so the data can be seen by an adversary. Additionally, Basic Authentication credentials (user name and password) are sent in the clear and can be intercepted.

See Also

NTLM and Kerberos Authentication
Internet Authentication

NTLM and Kerberos Authentication

7/29/2017 • 1 min to read • Edit Online

Default NTLM authentication and Kerberos authentication use the Microsoft Windows NT user credentials associated with the calling application to attempt authentication with the server. When using non-default NTLM authentication, the application sets the authentication type to NTLM and uses a NetworkCredential object to pass the user name, password, and domain to the host, as shown in the following example.

```
Dim MyURI As String = "http://www.contoso.com/"
Dim WReq As WebRequest = WebRequest.Create(MyURI)
WReq.Credentials = _
    New NetworkCredential(UserName, SecurelyStoredPassword, Domain)
```

```
String MyURI = "http://www.contoso.com/";
WebRequest WReq = WebRequest.Create (MyURI);
WReq.Credentials =
    new NetworkCredential(UserName, SecurelyStoredPassword, Domain);
```

Applications that need to connect to Internet services using the credentials of the application user can do so with the user's default credentials, as shown in the following example.

```
Dim MyURI As String = "http://www.contoso.com/"
Dim WReq As WebRequest = WebRequest.Create(MyURI)
WReq.Credentials = CredentialCache.DefaultCredentials
```

```
String MyURI = "http://www.contoso.com/";
WebRequest WReq = WebRequest.Create (MyURI);
WReq.Credentials = CredentialCache.DefaultCredentials;
```

The negotiate authentication module determines whether the remote server is using NTLM or Kerberos authentication, and sends the appropriate response.

> **NOTE**
>
> NTLM authentication does not work through a proxy server.

See Also

Basic and Digest Authentication
Internet Authentication

Web and Socket Permissions

7/29/2017 • 1 min to read • Edit Online

Internet security for applications using the System.Net namespace is provided by the WebPermission and SocketPermission classes. The **WebPermission** class controls an application's right to request data from a URI or to serve a URI to the Internet. The **SocketPermission** class controls an application's right to use a Socket to accept data on a local port or to contact remote devices using a transport protocol at another address, based on the host, port number, and transport protocol of the socket.

Which permission class you use depends on your application type. Applications that use WebRequest and its descendants should use the **WebPermission** class to manage permissions. Applications that use socket-level access should use the **SocketPermission** class to manage permissions.

WebPermission and **SocketPermission** define two permissions: accept and connect. Accept grants the application the right to answer an incoming connection from another party. Connect grants the application the right to initiate a connection to another party.

For **SocketPermission** instances, accept means that an application can accept incoming connections on a local transport address; connect means that an application can connect to some remote (or local) transport address.

For **WebPermission** instances, accept means that an application can export the URI controlled by the **WebPermission** to the world; connect means that an application can access that URI (whether it is remote or local).

See Also

Security
Security in Network Programming

Best Practices for System.Net Classes

7/29/2017 • 1 min to read • Edit Online

The following recommendations will help you use the classes contained in System.Net to their best advantage:

- Use WebRequest and WebResponse whenever possible instead of type casting to descendant classes. Applications that use **WebRequest** and **WebResponse** can take advantage of new Internet protocols without needing extensive code changes.
- When writing ASP.NET applications that run on a server using the **System.Net** classes, it is often better, from a performance standpoint, to use the asynchronous methods for GetResponse and GetResponseStream.
- The number of connections opened to an Internet resource can have a significant impact on network performance and throughput. **System.Net** uses two connections per application per host by default. Setting the ConnectionLimit property in the ServicePoint for your application can increase this number for a particular host. Setting the System.Net.ServicePointManager.DefaultPersistentConnectionLimit property can increase this default for all hosts.
- When writing socket-level protocols, try to use TcpClient or UdpClient whenever possible instead of writing directly to a Socket. These two client classes encapsulate the creation of TCP and UDP sockets without requiring you to handle the details of the connection.
- When accessing sites that require credentials, use the CredentialCache class to create a cache of credentials rather than supplying them with every request. The **CredentialCache** class searches the cache to find the appropriate credential to present with a request, relieving you of the responsibility of creating and presenting credentials based on the URL.

See Also

Network Programming in the .NET Framework

Accessing the Internet Through a Proxy

7/29/2017 • 1 min to read • Edit Online

If your site uses a proxy to provide access to the Internet, you must configure a proxy instance to enable your application to communicate with the Web proxy.

This section includes the following topics:

- Proxy Configuration
- Automatic Proxy Detection
- How to: Enable a WebRequest to Use a Proxy to Communicate With the Internet
- How to: Override a Global Proxy Selection

See Also

Using Application Protocols
Network Programming in the .NET Framework

Proxy Configuration

7/29/2017 • 3 min to read • Edit Online

A proxy server handles client requests for resources. A proxy can return a requested resource from its cache or forward the request to the server where the resource resides. Proxies can improve network performance by reducing the number of requests sent to remote servers. Proxies can also be used to restrict access to resources.

Adaptive Proxies

In the .NET Framework, proxies come in two varieties: adaptive and static. Adaptive proxies adjust their settings when the network configuration changes. For example, if a laptop user starts a dialup network connection, an adaptive proxy would recognize this change, discover and run its new configuration script, and adjust its settings appropriately.

Adaptive proxies are configured by a configuration script (see Automatic Proxy Detection). The script generates a set of application protocols and a proxy for each protocol.

Several options control how the configuration script is run. You can specify the following:

- How often the configuration script is downloaded and run.
- How long to wait for the script to download.
- Which credentials your system should use to access the proxy.
- Which credentials your system should use to download the configuration script.

Changes in the network environment may require that the system use a new set of proxies. If a network connection goes down or a new network connection is initialized, the system must discover the appropriate source of the configuration script in the new environment and run the new script.

The following table shows configuration options for an adaptive proxy.

ATTRIBUTE, PROPERTY, OR CONFIGURATION FILE SETTING	DESCRIPTION
`scriptDownloadInterval`	Elapsed time in seconds between script downloads.
`scriptDownloadTimeout`	Time to wait (in seconds) for the script to download.
`useDefaultCredentials` or UseDefaultCredentials	Controls whether the system uses the default network credentials to access a proxy.
`useDefaultCredentialForScriptDownload`	Controls whether the system uses the default network credentials to download the configuration script.

ATTRIBUTE, PROPERTY, OR CONFIGURATION FILE SETTING	DESCRIPTION
`usesystemdefaults`	Controls whether the static proxy settings (proxy address, bypass list, and bypass on local) should be read from the Internet Explorer proxy settings for the user. If this value is set to "true", then the static proxy settings from Internet Explorer will be used. If this value is "false" or not set, then the static proxy settings can be specified in the configuration and will override the Internet Explorer proxy settings. This value must also be set to "false" or not set for adaptive proxies to be enabled.

The following example shows a typical adaptive proxy configuration.

```
<system.net>
    <defaultProxy>
      <proxy  scriptDownloadInterval="600"
              scriptDownloadTimeout="30"
              useDefaultCredentials="true"
              usesystemdefaults="true"
      />
    </defaultProxy>
</system.net>
```

Static Proxies

Static proxies are usually configured explicitly by an application, or when a configuration file is invoked by an application or the system. Static proxies are useful in networks in which the topology changes infrequently, such as a desktop computer connected to a corporate network.

Several options control how a static proxy operates. You can specify the following:

- The address of the proxy.
- Whether the proxy should be bypassed for local addresses.
- Whether the proxy should be bypassed for a set of addresses.

The following table shows the configuration options for a static proxy.

ATTRIBUTE, PROPERTY, OR CONFIGURATION FILE SETTING	DESCRIPTION
`proxyaddress` or Address	The address of the proxy to use.
`bypassonlocal` or BypassProxyOnLocal	Controls whether the proxy is bypassed for local addresses.
`bypasslist` or BypassArrayList	Describes, with regular expressions, a set of addresses that bypass the proxy.

ATTRIBUTE, PROPERTY, OR CONFIGURATION FILE SETTING	DESCRIPTION
`usesystemdefaults`	Controls whether the static proxy settings (proxy address, bypass list, and bypass on local) should be read from the Internet Explorer proxy settings for the user. If this value is set to "true", then the static proxy settings from Internet Explorer will be used. On .NET Framework 2.0 when this value is set to "true", the Internet Explorer proxy settings are not overridden by other proxy settings in the configuration file. On .NET Framework 1.1, the Internet Explorer proxy settings can be overridden by other proxy settings in the configuration file. If this value is "false" or not set, then the static proxy settings can be specified in the configuration and will override the Internet Explorer proxy settings. This value must also be set to "false" or not set for adaptive proxies to be enabled.

The following example shows a typical static proxy configuration.

```
<system.net>
    <defaultProxy>
        <proxy  proxyaddress="http://proxy.contoso.com:3128"
                bypassonlocal="true"
        />
        <bypasslist>
            <add address="[a-z]+.blueyonderairlines.com$" />
        </bypasslist>
    </defaultProxy>
</system.net>
```

See Also

WebProxy
GlobalProxySelection
Automatic Proxy Detection

Automatic Proxy Detection

7/29/2017 • 2 min to read • Edit Online

Automatic proxy detection is a process by which a Web proxy server is identified by the system and used to send requests on behalf of the client. This feature is also known as Web Proxy Auto-Discovery (WPAD). When automatic proxy detection is enabled, the system attempts to locate a proxy configuration script that is responsible for returning the set of proxies that can be used for the request. If the proxy configuration script is found, the script is downloaded, compiled, and run on the local computer when proxy information, the request stream, or the response is obtained for a request that uses a WebProxy instance.

Automatic proxy detection is performed by the WebProxy class and can employ request-level settings, settings in configuration files, and settings specified using the Internet Explorer **Local Area Network (LAN)** dialog box.

> **NOTE**
>
> You can display the Internet Explorer **Local Area Network (LAN) Settings** dialog box by selecting **Tools** from the Internet Explorer main menu and then selecting **Internet Options**. Next, select the **Connections** tab, and click **LAN Settings**.

When automatic proxy detection is enabled, the WebProxy class attempts to locate the proxy configuration script as follows:

1. The WinINet `InternetQueryOption` function is used to locate the proxy configuration script most recently detected by Internet Explorer.
2. If the script is not located, the WebProxy class uses the Dynamic Host Configuration Protocol (DHCP) to locate the script. The DHCP server can respond either with the location (host name) of the script or with the full URL for the script.
3. If DHCP does not identify the WPAD host, DNS is queried for a host with WPAD as its name or alias.
4. If the host is not identified and the location of a proxy configuration script is specified by the Internet Explorer LAN settings or a configuration file, this location is used.

> **NOTE**
>
> Applications running as an NT Service or as part of ASP.NET use the Internet Explorer proxy server settings (if available) of the invoking user. These settings may not be available for all service applications.

Proxies are configured on a per-connectoid basis. A connectoid is an item in the network connection dialog, and can be a physical network device (a modem or Ethernet card) or a virtual interface (such as a VPN connection running over a network device). When a connectoid changes (for example, a wireless connection changes an access point, or a VPN is enabled), the proxy detection algorithm is run again.

By default, the Internet Explorer proxy settings are used to detect the proxy. If your application is running under a non-interactive account (without a convenient way to configure IE proxy settings), or if you want to use proxy settings different than the IE settings, you can configure your proxy by creating a configuration file with the <defaultProxy> Element (Network Settings) and <proxy> Element (Network Settings) elements defined.

For requests that you create, you can disable automatic proxy detection at the request level by using a null Proxy with your request, as shown in the following code example.

```
public static void DisableForMyRequest (Uri resource)
{
    WebRequest request = WebRequest.Create (resource);
    request.Proxy = null;
    WebResponse response = request.GetResponse ();
}
```

```
Public Shared Sub DisableForMyRequest(ByVal resource As Uri)
    Dim request As WebRequest = WebRequest.Create(resource)
    request.Proxy = Nothing
    Dim response As WebResponse = request.GetResponse()
    End Sub
```

Requests that do not have a proxy use your application domain's default proxy, which is available in the DefaultWebProxy property.

See Also

WebProxy
WebRequest
<system.Net> Element (Network Settings)

How to: Enable a WebRequest to Use a Proxy to Communicate With the Internet

7/29/2017 • 1 min to read • Edit Online

This example creates a global proxy instance that will enable any WebRequest to use a proxy to communicate with the Internet. The example assumes that the proxy server is named `webproxy` and that it communicates on port 80, the standard HTTP port.

Example

```
WebProxy proxyObject = new WebProxy("http://webproxy:80/");
GlobalProxySelection.Select = proxyObject;
```

```
Dim proxyObject As WebProxy = New WebProxy("http://webproxy:80/")
GlobalProxySelection.Select = proxyObject
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

See Also

Using Application Protocols
Accessing the Internet Through a Proxy

How to: Override a Global Proxy Selection

7/29/2017 • 1 min to read • Edit Online

This example sends a **WebRequest** to www.contoso.com that overrides the global proxy selection with a proxy server named `alternateproxy` on port 80.

Example

```
WebRequest req = WebRequest.Create("http://www.contoso.com/");
req.Proxy = new WebProxy("http://alternateproxy:80/");
```

```
Dim req As WebRequest = WebRequest.Create("http://www.contoso.com/")
req.Proxy = New WebProxy("http://alternateproxy:80/")
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

See Also

Using Application Protocols
Accessing the Internet Through a Proxy

NetworkInformation

7/29/2017 • 1 min to read • Edit Online

The System.Net.NetworkInformation namespace enables you to gather information about network events, changes, statistics, and properties. You can also determine whether a remote host is reachable by using the System.Net.NetworkInformation.Ping class.

Network Availability and Events

The System.Net.NetworkInformation.NetworkChange class enables you to determine whether the network address or availability has changed. To use this class, create an event handler to process the change, and associate it with a NetworkAddressChangedEventHandler or a NetworkAvailabilityChangedEventHandler. For more information, see How to: Detect Network Availability and Address Changes.

Network Statistics and Properties

You can gather network statistics and properties on an interface or protocol basis. The NetworkInterface, NetworkInterfaceType, and PhysicalAddress classes give information about a particular network interface, while the IPInterfaceProperties, IPGlobalProperties, IPGlobalStatistics, TcpStatistics, and UdpStatistics classes give information about layer 3 and layer 4 packets. For more information, see How to: Get Interface and Protocol Information.

Determine if a Remote Host is Reachable

You can use the Ping class to determine whether a Remote Host is up, on the network, and reachable. For more information, see How to: Ping a Host.

See Also

Network Programming Samples
Network Information Technology Sample
NetStat Tool Technology Sample
Ping Client Technology Sample

How to: Detect Network Availability and Address Changes

7/29/2017 • 1 min to read • Edit Online

This sample shows how to detect changes in the network address of an interface.

Example

```
using System;
using System.Net;
using System.Net.NetworkInformation;

namespace Examples.Net.AddressChanges
{
    public class NetworkingExample
    {
        public static void Main()
        {
            NetworkChange.NetworkAddressChanged += new
             NetworkAddressChangedEventHandler(AddressChangedCallback);
            Console.WriteLine("Listening for address changes. Press any key to exit.");
            Console.ReadLine();
        }
        static void AddressChangedCallback(object sender, EventArgs e)
        {

            NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();
            foreach(NetworkInterface n in adapters)
            {
                Console.WriteLine("   {0} is {1}", n.Name, n.OperationalStatus);
            }
        }
    }
}
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

How to: Get Interface and Protocol Information

7/29/2017 • 1 min to read • Edit Online

This sample shows how to read the TCP statistics of a network interface.

Example

```
public static void ShowTcpStatistics(NetworkInterfaceComponent version)
{
    IPGlobalProperties properties =
          IPGlobalProperties.GetIPGlobalProperties();
    TcpStatistics tcpstat = null;
    Console.WriteLine("");
    switch (version)
    {
        case NetworkInterfaceComponent.IPv4:
             tcpstat = properties.GetTcpIPv4Statistics();
            Console.WriteLine("TCP/IPv4 Statistics:");
            break;
        case NetworkInterfaceComponent.IPv6:
            tcpstat = properties.GetTcpIPv6Statistics();
            Console.WriteLine("TCP/IPv6 Statistics:");
            break;
        default:
            throw new ArgumentException("version");
            break;
    }
    Console.WriteLine("  Minimum Transmission Timeout. : {0}",
        tcpstat.MinimumTransmissionTimeout);
    Console.WriteLine("  Maximum Transmission Timeout. : {0}",
        tcpstat.MaximumTransmissionTimeout);

    Console.WriteLine("  Connection Data:");
    Console.WriteLine("      Current : {0}",
    tcpstat.CurrentConnections);
    Console.WriteLine("      Cumulative : {0}",
        tcpstat.CumulativeConnections);
    Console.WriteLine("      Initiated  : {0}",
        tcpstat.ConnectionsInitiated);
    Console.WriteLine("      Accepted : {0}",
        tcpstat.ConnectionsAccepted);
    Console.WriteLine("      Failed Attempts : {0}",
        tcpstat.FailedConnectionAttempts);
    Console.WriteLine("      Reset : {0}",
        tcpstat.ResetConnections);

    Console.WriteLine("");
    Console.WriteLine("  Segment Data:");
    Console.WriteLine("      Received  ................... : {0}",
        tcpstat.SegmentsReceived);
    Console.WriteLine("      Sent : {0}",
        tcpstat.SegmentsSent);
    Console.WriteLine("      Retransmitted : {0}",
        tcpstat.SegmentsResent);

    Console.WriteLine("");
}
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

How to: Ping a Host

7/29/2017 • 1 min to read • Edit Online

This sample shows how to ping a remote host.

Example

```
using System;
using System.Text;
using System.Net;
using System.Net.NetworkInformation;
using System.ComponentModel;
using System.Threading;

namespace Examples.System.Net.NetworkInformation.PingTest
{
    public class PingExample
    {
        public static void Main (string[] args)
        {
            if (args.Length == 0)
                throw new ArgumentException ("Ping needs a host or IP Address.");

            string who = args[0];
            AutoResetEvent waiter = new AutoResetEvent (false);

            Ping pingSender = new Ping ();

            // When the PingCompleted event is raised,
            // the PingCompletedCallback method is called.
            pingSender.PingCompleted += new PingCompletedEventHandler (PingCompletedCallback);

            // Create a buffer of 32 bytes of data to be transmitted.
            string data = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
            byte[] buffer = Encoding.ASCII.GetBytes (data);

            // Wait 12 seconds for a reply.
            int timeout = 12000;

            // Set options for transmission:
            // The data can go through 64 gateways or routers
            // before it is destroyed, and the data packet
            // cannot be fragmented.
            PingOptions options = new PingOptions (64, true);

            Console.WriteLine ("Time to live: {0}", options.Ttl);
            Console.WriteLine ("Don't fragment: {0}", options.DontFragment);

            // Send the ping asynchronously.
            // Use the waiter as the user token.
            // When the callback completes, it can wake up this thread.
            pingSender.SendAsync(who, timeout, buffer, options, waiter);

            // Prevent this example application from ending.
            // A real application should do something useful
            // when possible.
            waiter.WaitOne ();
            Console.WriteLine ("Ping example completed.");
        }

        public static void PingCompletedCallback (object sender, PingCompletedEventArgs e)
```

```
        {
            // If the operation was canceled, display a message to the user.
            if (e.Cancelled)
            {
                Console.WriteLine ("Ping canceled.");

                // Let the main thread resume.
                // UserToken is the AutoResetEvent object that the main thread
                // is waiting for.
                ((AutoResetEvent)e.UserState).Set ();
            }

            // If an error occurred, display the exception to the user.
            if (e.Error != null)
            {
                Console.WriteLine ("Ping failed:");
                Console.WriteLine (e.Error.ToString ());

                // Let the main thread resume.
                ((AutoResetEvent)e.UserState).Set();
            }

            PingReply reply = e.Reply;

            DisplayReply (reply);

            // Let the main thread resume.
            ((AutoResetEvent)e.UserState).Set();
        }

        public static void DisplayReply (PingReply reply)
        {
            if (reply == null)
                return;

            Console.WriteLine ("ping status: {0}", reply.Status);
            if (reply.Status == IPStatus.Success)
            {
                Console.WriteLine ("Address: {0}", reply.Address.ToString ());
                Console.WriteLine ("RoundTrip time: {0}", reply.RoundtripTime);
                Console.WriteLine ("Time to live: {0}", reply.Options.Ttl);
                Console.WriteLine ("Don't fragment: {0}", reply.Options.DontFragment);
                Console.WriteLine ("Buffer size: {0}", reply.Buffer.Length);
            }
        }
    }
}
```

Compiling the Code

This example requires:

- References to the **System.Net** namespace.

Changes to the System.Uri namespace in Version 2.0

7/29/2017 • 1 min to read • Edit Online

Several changes were made to the System.Uri class. These changes fixed incorrect behavior, enhanced usability, and enhanced security.

Obsolete and Deprecated Members

Constructors:

- All constructors that have a `dontEscape` parameter.

Methods:

- CheckSecurity
- Escape
- Canonicalize
- Parse
- IsReservedCharacter
- IsBadFileSystemCharacter
- IsExcludedCharacter
- EscapeString

Changes

- For URI schemes that are known to not have a query part (file, ftp, and others), the '?' character is always escaped and is not considered the beginning of a Query part.
- For implicit file URIs (of the form "c:\directory\file@name.txt"), the fragment character ('#') is always escaped unless full unescaping is requested or LocalPath is `true`.
- UNC hostname support was removed; the IDN specification for representing international hostnames was adopted.
- LocalPath always returns a completely unescaped string.
- ToString does not unescape an escaped '%', '?', or '#' character.
- Equals now includes the Query part in the equality check.
- Operators "==" and "!=" are overridden and linked to the Equals method.
- IsLoopback now produces consistent results.
- The URI "`file:///path`" is no longer translated into "file://path".
- "#" is now recognized as a host name terminator. That is, "http://consoto.com#fragment" is now converted to "http://contoso.com/#fragment".
- A bug when combining a base URI with a fragment has been fixed.

- A bug in HostNameType is fixed.
- A bug in NNTP parsing is fixed.
- A URI of the form HTTP:contoso.com now throws a parsing exception.
- The Framework correctly handles userinfo in a URI.
- URI path compression is fixed so that a broken URI cannot traverse the file system above the root.

See Also

System.Uri

International Resource Identifier Support in System.Uri

7/29/2017 • 2 min to read • Edit Online

The System.Uri class has been extended with International Resource Identifier (IRI) and Internationalized Domain Names (IDN) support. These enhancements are available in .NET Framework 3.5, 3.0 SP1, and 2.0 SP1.

IRI and IDN Support

Web addresses are typically expressed using Uniform Resource Identifiers (URI) that consist of a very restricted set of characters:

- Upper and lower case ASCII letters from the English alphabet.
- Digits from 0 to 9.
- A small number of other ASCII symbols.

The specifications for URIs are documented in RFC 2396 and RFC 3986 published by the Internet Engineering Task Force (IETF).

With the growth of the Internet, there is a growing need to identify resources using languages other than English. Identifiers which facilitate this need and allow non-ASCII characters (characters in the Unicode/ISO 10646 character set) are known as International Resource Identifiers (IRIs). The specifications for IRIs are documented in RFC 3987 published by IETF. Using IRIs allows a URL to contain Unicode characters.

The existing System.Uri class has been extended to provide IRI support based on RFC 3987. Current users will not see any change from the .NET Framework 2.0 behavior unless they specifically enable IRI. This ensures application compatibility with prior versions of the .NET Framework.

An application can specify whether to use Internationalized Domain Name (IDN) parsing applied to domain names and whether IRI parsing rules should be applied. This can be done in the machine.config or in the app.config file.

Enabling IDN will convert all Unicode labels in a domain name to their Punycode equivalents. Punycode names contain only ASCII characters and always start with the xn-- prefix. The reason for this is to support existing DNS servers on the Internet, since most DNS servers only support ASCII characters (see RFC 3940).

Enabling IRI and IDN affects the value of the System.Uri.DnsSafeHost property. Enabling IRI and IDN can also change the behavior of the System.Uri.Equals, System.Uri.OriginalString, System.Uri.GetComponents, and IsWellFormedOriginalString methods.

The System.GenericUriParser class has also been extended to allow creating a customizable parser that supports IRI and IDN. The behavior of a System.GenericUriParser object is specified by passing a bitwise combination of the values available in the System.GenericUriParserOptions enumeration to the System.GenericUriParser constructor. The System.GenericUriParserOptions.IriParsing type indicates the parser supports the parsing rules specified in RFC 3987 for International Resource Identifiers (IRI). Whether IRI is actually used depends on if IRI is enabled.

The System.GenericUriParserOptions.Idn type indicates the parser supports Internationalized Domain Name (IDN) parsing (IDN) of host names. Whether IDN is actually used depends on if IDN is enabled.

Enabling IRI parsing will do normalization and character checking according to the latest IRI rules in RFC 3987. The default value is for IRI parsing to be disabled so normalization and character checking are done according to RFC 2396 and RFC 3986.

IRI and IDN processing in the System.Uri class can also be controlled using the System.Configuration.IriParsingElement and System.Configuration.IdnElement configuration setting classes. The System.Configuration.IriParsingElement setting enables or disables IRI processing in the System.Uri class. The System.Configuration.IdnElement setting enables or disables IDN processing in the Uri class. The System.Configuration.IriParsingElement setting also indirectly controls IDN. IRI processing must be enabled for IDN processing to be possible. If IRI processing is disabled, then IDN processing will be set to the default setting where the .NET Framework 2.0 behavior is used for compatibility and IDN names are not used.

The configuration setting for the System.Configuration.IriParsingElement and System.Configuration.IdnElement configuration classes will be read once when the first System.Uri class is constructed. Changes to configuration settings after that time are ignored.

See Also

System.Configuration.IdnElement
System.Configuration.IriParsingElement
System.Uri
System.Uri.DnsSafeHost

Socket Performance Enhancements in Version 3.5

7/29/2017 • 2 min to read • Edit Online

The System.Net.Sockets.Socket class has been enhanced in Version 3.5 for use by applications that use asynchronous network I/O to achieve the highest performance. A series of new classes have been added as part of a set of enhancements to the Socket class that provide an alternative asynchronous pattern that can be used by specialized high-performance socket applications. These enhancements were specifically designed for network server applications that require high performance. An application can use the enhanced asynchronous pattern exclusively, or only in targeted hot areas of their application (when receiving large amounts of data, for example).

Class Enhancements

The main feature of these enhancements is the avoidance of the repeated allocation and synchronization of objects during high-volume asynchronous socket I/O. The Begin/End design pattern currently implemented by the Socket class for asynchronous socket I/O requires a System.IAsyncResult object be allocated for each asynchronous socket operation.

In the new Socket class enhancements, asynchronous socket operations are described by reusable System.Net.Sockets.SocketAsyncEventArgs class objects allocated and maintained by the application. High-performance socket applications know best the amount of overlapped socket operations that must be sustained. The application can create as many of the SocketAsyncEventArgs objects that it needs. For example, if a server application needs to have 15 socket accept operations outstanding at all times to support incoming client connection rates, it can allocate 15 reusable SocketAsyncEventArgs objects in advance for that purpose.

The pattern for performing an asynchronous socket operation with this class consists of the following steps:

1. Allocate a new SocketAsyncEventArgs context object, or get a free one from an application pool.
2. Set properties on the context object to the operation about to be performed (the callback delegate method and data buffer, for example).
3. Call the appropriate socket method (xxxAsync) to initiate the asynchronous operation.
4. If the asynchronous socket method (xxxAsync) returns true in the callback, query the context properties for completion status.
5. If the asynchronous socket method (xxxAsync) returns false in the callback, the operation completed synchronously. The context properties may be queried for the operation result.
6. Reuse the context for another operation, put it back in the pool, or discard it.

The lifetime of the new asynchronous socket operation context object is determined by references in the application code and asynchronous I/O references. It is not necessary for the application to retain a reference to an asynchronous socket operation context object after it is submitted as a parameter to one of the asynchronous socket operation methods. It will remain referenced until the completion callback returns. However it is advantageous for the application to retain the reference to the context object so that it can be reused for a future asynchronous socket operation.

See Also

System.Net.Sockets.Socket
System.Net.Sockets.SendPacketsElement
System.Net.Sockets.SocketAsyncEventArgs

System.Net.Sockets.SocketAsyncOperation
Network Programming Samples
Socket Performance Technology Sample

Peer Name Resolution Protocol

7/29/2017 • 2 min to read • Edit Online

In peer-to-peer environments, peers use specific name resolution systems to resolve each other's network locations (addresses, protocols, and ports) from names or other types of identifiers. In the past, peer name resolution has been complicated by the inherently transient connectivity as well as other shortcomings within the Domain Name System (DNS).

The Microsoft® Windows® Peer-to-Peer Networking platform solves this problem with the Peer Name Resolution Protocol (PNRP), a secure, scalable, and dynamic name registration and name resolution protocol first developed for Windows XP and then upgraded in Windows Vista™. PNRP works very differently from traditional name resolution systems, opening up exciting new possibilities for application developers.

With PNRP, peer names can be applied to the machine, or individual applications or services on the machine. A peer name resolution includes an address, port, and possibly an extended payload. Benefits of this system include fault tolerance, no bottlenecks, and name resolutions that will never return stale addresses; making the protocol an excellent solution for locating mobile users.

In terms of security, peer names can be published as secured (protected) or unsecured (unprotected). PNRP uses public key cryptography to protect secure peer names against spoofing; both computers and services can be named with PNRP.

- The Peer Name Resolution Protocol demonstrates the following properties:
- Distributed and almost entirely serverless. Servers are only required for the bootstrapping process.
- Secure name publication without the involvement of third parties. Unlike DNS name publication, PNRP name publication is instantaneous and without financial cost.
- PNRP updates in real-time, which prevents the resolution of stale addresses.
- The resolution of names via PNRP extends beyond computers by also allowing name resolution for services.

-

The System.Net.PeerToPeer Namespace

- PNRP functionality is defined by the System.Net.PeerToPeer namespace within the .NET Framework version 3.5. It provides a set of types that can be used to register and resolve peer names with an available PNRP service.

-

- (PNRP and custom peer resolvers can be created and instantiated using the types provided in the System.ServiceModel.PeerResolvers namespace.)

-

- The basic types used to register and resolve names with an available PNRP service are as follows:

-

- Cloud: Defines the information describing an available PNRP cloud, including its scope.
- PeerName: Defines a peer name that can be used to register and subsequently resolve a peer within a cloud.
- PeerNameRecord: Defines the record in PNRP cloud that contains the registration information for a peer,

which includes the network endpoints at which the peer can be contacted.

- PeerNameRegistration: Defines the registration process for a peer name, including methods to start and stop peer name registration.
- PeerNameResolver: Defines the process for resolving a peer name to its network endpoint(s), including both synchronous and asynchronous methods for resolution.

-

-

See Also

System.ServiceModel.PeerResolvers
System.Net.PeerToPeer
Network Programming Samples
PeerToPeer Technology Sample

Peer Names and PNRP IDs

7/29/2017 • 1 min to read • Edit Online

A Peer Name represents an endpoint for communication, which can be a computer, a user, a group, a service, or anything associated with a Peer that can be resolved to an IPv6 address. The Peer Name Resolution Protocol (PNRP) takes the statistically unique Peer Name for the creation of a PNRP ID, which is used to identify cloud members.

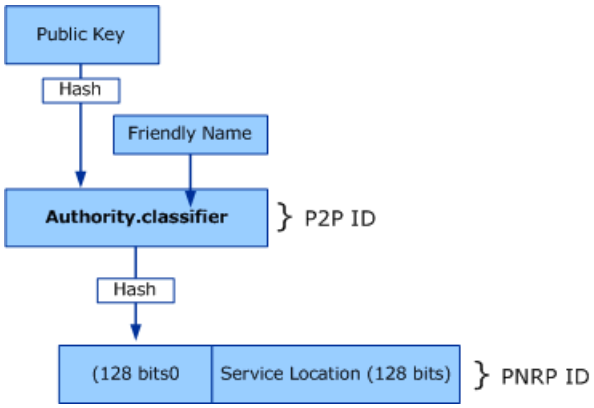
Peer Names

Peer names can be registered as unsecured or secured. Unsecured names are just text strings that are subject to spoofing, as anyone can register a duplicate unsecured name. Unsecured names are best used in private or otherwise protected networks. Secured names are protected with a certificate and a digital signature. Only the original publisher will be able to prove ownership of a secured name.

The combination of cloud and scope provides a reasonably secure environment for peers that participate in PNRP activity. However, using a secured peer name does not ensure the overall security of the networking application. Security of the application is implementation-dependent.

Secured peer names are only registered by their owner and are protected with public key cryptography. A secured peer name is considered owned by the peer entity having the corresponding private key. Ownership can be proved via the certified peer address (CPA), which is signed using the private key. A malicious user cannot forge ownership of a peer name without the corresponding private key.

PNRP IDs



PNRP IDs are composed of the following:

- The high-order 128 bits, known as the peer-to-peer (P2P) ID, are a hash of a peer name assigned to the endpoint. The peer name has the following format: *Authority.Classifier*. For secured names, *Authority* is the Secure Hash Algorithm 1 (SHA1) hash of the public key of the peer name in hexadecimal characters. For unsecured names, the *Authority* is the single character "0". *Classifier* is a string that identifies the application. No peer name classifier can be greater than 149 characters long, including the `null` terminator.
- The low-order 128 bits are used for the Service Location, which is a generated number that identifies different instances of the same P2P ID in the same cloud.

This combination of P2P ID and Service Location allows multiple PNRP IDs to be registered from a single computer.

See Also

PeerName

System.Net.PeerToPeer

Peer Name Publication and Resolution

7/29/2017 • 3 min to read • Edit Online

Publishing a Peer Name

To publish a new PNRP ID, a peer performs the following:

- Sends PNRP publication messages to its cache neighbors (the peers that have registered PNRP IDs in the lowest level of the cache) to seed their caches.
- Chooses random nodes in the cloud that are not its neighbors and sends them PNRP name resolution requests for its own P2P ID. The resulting endpoint determination process seeds the caches of random nodes in the cloud with the PNRP ID of the publishing peer.

-

PNRP version 2 nodes do not publish PNRP IDs if they are only resolving other P2P IDs. The HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\PeerNet\PNRP\IPV6-Global\SearchOnly=1 registry value (REG_DWORD type) specifies that peers only use PNRP for name resolution, never for name publication. This registry value can also be configured through Group Policy.

Resolving a Peer Name

Locating other peers in a PNRP network or cloud is a process comprised of two phases:

1. Endpoint Determination
2. PNRP ID Resolution

In the endpoint determination phase, a peer that is attempting to resolve the PNRP ID of a service on another computer determines the IPv6 address of that remote peer. The remote peer is the one that published, or is associated with, the PNRP ID of the computer or service.

After confirming that the remote endpoint has been registered into the PNRP cloud, the requesting peer in the PNRP ID resolution phase sends a request to that peer endpoint for the PNRP ID of the desired service. The endpoint sends a reply confirming the PNRP ID of the service, a comment, and up to 4 kilobytes of additional information that the requesting peer can use for future communication. For example, if the desired endpoint is a gaming server, the additional peer name record data can contain information about the game, the level of play, and the current number of players.

In the endpoint determination phase, PNRP uses an iterative process for locating the node that published the PNRP ID, in which the node performing the resolution is responsible for contacting nodes that are successively closer to the target PNRP ID.

To perform name resolution in PNRP, the peer examines the entries in its own cache for an entry that matches the target PNRP ID. If found, the peer sends a PNRP Request message to the peer and waits for a response. If an entry for the PNRP ID is not found, the peer sends a PNRP Request message to the peer that corresponds to the entry that has a PNRP ID that most closely matches the target PNRP ID. The node that receives the PNRP Request message examines its own cache and does the following:

- If the PNRP ID is found, the requested endpoint peer replies directly to the requesting peer.
- If the PNRP ID is not found and a PNRP ID in the cache is closer to the target PNRP ID, the requested peer sends a response to the requesting peer containing the IPv6 address of the peer that represents the entry

with a PNRP ID that more closely matches the target PNRP ID. Using the IP address in the response, the requesting node sends another PNRP Request message to the IPv6 address to respond or examine its cache.

- If the PNRP ID is not found and there is no PNRP ID in its cache that is closer to the target PNRP ID, the requested peer sends the requesting peer a response that indicates this condition. The requesting peer then chooses the next-closest PNRP ID.

-

The requesting peer continues this process with successive iterations, eventually locating the node that registered the PNRP ID.

Within the System.Net.PeerToPeer namespace, there is a many-to-many relationship between the PeerName records that contain endpoints and PNRP clouds or meshes in which they communicate. When there are duplicate or stale entries, or multiple nodes with the same peer name, PNRP nodes can obtain current information using the PeerNameResolver class. The PeerNameResolver methods use a single peer name to simplify the perspective to one peer-to-many peer name records and the same one peer to many clouds. This is similar to a query performed using a relational-table join. Upon successful completion, the Resolver object returns a PeerNameRecordCollection for the specified peer name. For example, a peer name would occur in all the peer name records in the collection, ordered by cloud. These are the instances of the peer name whose supporting data may be requested by a PNRP-based application.

See Also

System.Net.PeerToPeer

PNRP Clouds

7/29/2017 • 1 min to read • Edit Online

A PNRP "cloud" represents a set of nodes that can communicate with each other through the network. The term "cloud" is synonymous with "peer mesh" and "peer-to-peer graph".

Communication between nodes should never cross from one cloud to another. A Cloud instance is uniquely identified by its name, which is case-sensitive. A single peer or node may be connected to more than one cloud.

Clouds are tied very closely to network interfaces. On a multi-homed machine with two network cards attached to different subnets, three clouds will be returned: one for each of the link local addresses per interface, and a single global scope cloud.

PNRP uses three cloud "scopes", in which a scope is a grouping of computers that are able to find each other:

- The global cloud corresponds to the global IPv6 address scope and global addresses and represents all the computers on the entire IPv6 Internet. There is only a single global cloud.
- The link-local cloud corresponds to the link-local IPv6 address scope and link-local addresses. A link-local cloud is for a specific link, which is typically the same as the locally attached subnet. There can be multiple link-local clouds.

A third cloud, the site-specific cloud, corresponds to the site IPv6 address scope and site-local addresses. This cloud has been deprecated, although it is still supported in PNRP.

Clouds

PNRP clouds are represented by instances of the Cloud class. Groups of clouds used a peer are represented by instances of the enumerable CloudCollection class. Collections of PNRP clouds known to the current peer can be obtained by calling the static GetAvailableClouds method.

Individual clouds have unique names, represented as a 256 character Unicode string. These names, along with the above-mentioned scope, are used to construct unique instances of the Cloud class. These instances can be serialized and reconstructed for persistent usage.

Once a Cloud instance is created or obtained, peer names can be registered with it to create a mesh of known peers.

See Also

Cloud
Peer Name Resolution Protocol

PNRP Caches

7/29/2017 • 3 min to read • Edit Online

Peer Name Resolution Protocol (PNRP) caches are local collections of algorithmically selected peer endpoints maintained on the peer.

PNRP Cache Initialization

To initialize the PNRP cache, or Peer Name Record Collection, when a peer node starts up, a node can use the following methods:

- Persistent cache entries that were present when the node was shut down are loaded from hard disk storage.
- If an application uses the P2P collaboration infrastructure, collaboration information is available in the Contact Manager for that node.

Scaling Peer Name Resolution with a Multi-Level Cache

To keep the sizes of the PNRP caches small, peer nodes use a multi-level cache, in which each level contains a maximum number of entries. Each level in the cache represents a one tenth smaller portion of the PNRP ID number space (2^{256}). The lowest level in the cache contains a locally registered PNRP ID and other PNRP IDs that are numerically close to it. As a level of the cache is filled with a maximum of 20 entries, a new lower level is created. The maximum number of levels in the cache is on the order of log10(Total number of PNRP IDs in the cloud). For example, for a global cloud with 100 million PNRP IDs, there are no more than 8 (=log10(100,000,000)) levels in the cache and a similar number of hops to resolve a PNRP ID during name resolution. This mechanism allows for a distributed hash table for which an arbitrary PNRP ID can be resolved by forwarding PNRP Request messages to the next-closest peer until the peer with the corresponding CPA is found.

To ensure that resolution can complete, each time a node adds an entry to the lowest level of its cache, it floods a copy of the entry to all the nodes within the last level of the cache.

The cache entries are refreshed over time. Cache entries that are stale are removed from the cache. The result is that the distributed hash table of PNRP IDs is based on active endpoints, unlike DNS in which address records and the DNS protocol provide no guarantee that the node associated with the address is actively on the network.

Other PNRP Caches

Another persistent data store is the local cache. In addition to the other objects needed for PNRP activity, it may include the records associated with a PNRP cloud or collaboration session that is securely published and synchronized between all the members of the cloud. This replicated store represents the view of the group data, which should be the same for all group members. Technically, these objects are not records per se, but rather application, presence, and object data destined for a local cache. Use of the PNRP cloud ensures that objects are propagated to all nodes in the collaboration session or PNRP cloud. Record replication between cloud members uses SSL to provide encryption and data integrity.

When a peer joins a cloud, they do not automatically receive local cache data from the host peer to which they attach; they have to subscribe to the host peer to receive updates in application, presence, and object data. After the initial synchronization, peers periodically resynchronize their replicated stores to ensure that all group members consistently have the same view. The collaboration session or applications within the collaboration session may also perform the same function.

After a collaboration session has begun for a cloud, applications can register peers and begin publishing their

information using the security defined by the cloud scope. When a peer joins a cloud, the security mechanisms for the cloud are applied to the peer, giving it a scope in which to participate. Its records can then be published securely within the scope of the cloud. Note that cloud scope may not be the same as collaboration application scope.

Peers can register interest in receiving objects from other peers. When an object is updated, the collaboration application is notified and the new object is passed to all subscribers of the application. For example, a peer in a group chat application can register interest in receiving application information, which will send it all chat records as application data. This allows it to monitor chat activity within the cloud.

See Also

System.Net.PeerToPeer

PNRP in Application Development

7/29/2017 • 1 min to read • Edit Online

In Windows Vista, networking applications can access name publication and resolution functions through a simplified PNRP application programming interface (API).

Implementing the Peer Name Resolution Protocol

With the simplified PNRP API, clouds are not explicitly specified to register the name and addresses; the PNRP component automatically determines the appropriate clouds to join and the addresses to publish within the clouds.

For highly simplified PNRP name resolution in Windows Vista, PNRP names are now integrated into the getaddrinfo() Windows Sockets function. To use PNRP to resolve a name to an IPv6 address, applications can use the getaddrinfo() function to resolve the Fully Qualified Domain Name (FQDN) name.prnp.net, in which name is peer name being resolved. The pnrp.net domain is a reserved domain in Windows Vista for PNRP name resolution.

Message passing between PeerToPeer applications is still handled by underlying architectures such as PeerChannel and WCF Large Data and Streaming.

See Also

System.Net.PeerToPeer

Peer-to-Peer Collaboration

7/29/2017 • 2 min to read • Edit Online

Peer-to-peer networking is the utilization of the relatively powerful computers (personal computers) that exist at the edge of the Internet for more than just client-based computing tasks. The modern personal computer (PC) has a very fast processor, vast memory, and a large hard disk, none of which are being fully utilized when performing common computing tasks such as e-mail and Web browsing. The modern PC can easily act as both a client and server (a peer) for many types of applications.

- The Peer-to-Peer Collaboration Infrastructure is a simplified implementation of the Microsoft Windows Peer-to-Peer Infrastructure that leverages the People Near Me service in Windows Vista and later platforms. It is best used for peer-enabled applications within a subnet for which the People Near Me service operates, although it can service internet endpoints or contacts as well. It incorporates the common Contact Manager that is used by Live Messenger and other Live-aware applications to determine contact endpoints, availability, and presence.

-

Collaboration Applications

A typical peer-to-peer collaboration application is comprised of the following steps:

- Peer determines the identity of a peer who is interested in hosting a collaboration session
- A request to host a session is sent, somehow, and the host peer agrees to manage collaboration activity.
- The host invites contacts on the subnet (including the requestor) to a session.
- All peers who want to collaborate may add the host to their contact managers.
- Most peers will send invitation responses, whether accepted or declined, back to the host peer in a timely fashion.
- All peers who want to collaborate will subscribe to the host peer.
- While the peers are performing their initial collaboration activity, the host peer may add remote peers to its contact manager. It also processes all invitation responses to determine who has accepted, who has declined, and who has not answered. It may cancel invitations to those who have not answered, or perform some other activity.
- At this point, the host peer can start a collaboration session with all invited peers, or register an application with the collaboration infrastructure. P2P applications use the Peer-to-Peer Collaboration Infrastructure and the System.Net.PeerToPeer.Collaboration namespace to coordinate communications for games, bulletin boards, conferencing, and other serverless presence applications.

-

Peer-to-Peer Networking Security

In an Active Directory domain, domain controllers provide authentication services using Kerberos. In a serverless peer environment, the peers must provide their own authentication. For Peer-to-Peer Networking, any node can act as a CA, removing the requirement of a root certificate in each peer's trusted root store. Authentication is provided using self-signed certificates, formatted as X.509 certificates. These are certificates that are created by each peer, which generates the public key/private key pair and the certificate that is signed using the private key. The self-

signed certificate is used for authentication and to provide information about the peer entity. Like X.509 authentication, peer networking authentication relies upon a chain of certificates tracing back to a public key that is trusted.

See Also

System.Net.PeerToPeer.Collaboration
About the System.Net.PeerToPeer.Collaboration Namespace

About the System.Net.PeerToPeer.Collaboration Namespace

7/29/2017 • 1 min to read • Edit Online

The System.Net.PeerToPeer.Collaboration namespace provides classes and APIs that are used to implement peer collaboration activities using the Peer-to-Peer Collaboration Infrastructure.

Classes

The main classes used in the implementation of a Peer-to-Peer Collaboration activity are:

- The ContactManager, which can be used to store peer contacts.
- The PeerApplication in which to collaborate, such as a game, chat client, or conferencing solution.
- The peers that will be collaborating in an activity. These peers can be represented as PeerContact, PeerNearMe, or PeerEndPoint objects.
- The static PeerCollaboration class itself, which specifies which applications are available and which peers are participating in them.

The Invite methods are used to invite peers to a collaboration session. A calling peer can subscribe to another peer for events that signal updates to application, object, or presence information affiliated with the collaboration session. Presence classes specify whether a Peer is available for collaboration, and the PeerScope class is used to specify how much participation is allowed for a peer: Internet (global), NearMe, (subnet) or None.

A collaboration session is comprised of four steps:

- Discovery. Discover or publish applications, peers, and presence information. For instance, find other people on the local subnet that have the same games installed.
- Invitation. Send and accept secure invitations for remote peer(s) to start or join PeerCollaboration sessions.
- Contact Management. Add discovered peers as a contact to a ContactManager.
- Communication. When communication is established, use the System.Net APIs, the System.Net.PeerToPeer API, or the Windows Communication Foundation Peer Channel classes for multiparty communications.

For example, the host peer starts a collaboration session, and utilizes the CreateContact method to add a remote peer and one of its local peers to the Contact Manager of the host peer. The three users will then participate in their own private collaboration session.

Typical P2P applications are: conference calls for collaborative note-taking or whiteboarding, serverless chat applications, interactive advertisements, and online gaming sessions.

See Also

System.Net.PeerToPeer.Collaboration

Peer-to-Peer Networking Scenarios

7/29/2017 • 3 min to read • Edit Online

Peer-to-peer networking enables or enhances the following scenarios:

Real-Time Communications (RTC)

- Serverless Instant Messaging

RTC exists today. Computer users can chat and have voice or video conversations with their peers today. However, many of the existing programs and their communications protocols rely on servers to function. If you are participating in an ad-hoc wireless network or are a part of an isolated network, you are unable to use these RTC facilities. Peer-to-peer technology allows the extension of RTC technologies to these additional networking environments.

- Real-time matchmaking and gameplay

Similar to RTC, real-time game play exists today. There are many Web-based game sites that cater to the gaming community via the Internet. They offer the ability to find other gamers with similar interests and play a game together. The problem is that the game sites exist only on the Internet and are geared toward the avid gamer who wants to play against the best gamers in the world. These sites track and provide the statistics to help in the process. However, these sites do not allow a gamer to set up an ad-hoc game among friends in a variety of networking environments. Peer-to-peer networking can provide this capability.

Collaboration

- Project workspaces solving a goal

Shared workspace applications allow for the creation of ad-hoc workgroups and then allow the workgroup owners to populate the shared workspace with the tools and content that will allow the group to solve a problem. This could include message boards, productivity tools, and files.

- Sharing files with others

A subset of project workspace sharing is the ability to share files. Although this ability exists today with the current version of Windows, it can be enhanced through peer-to-peer networking to make file content available in an easy and friendly way. Allowing easy access to the incredible wealth of content at the edge of the Internet or in ad-hoc computing environments increases the value of network computing.

- Sharing experiences

With wireless connectivity becoming more prevalent, peer-to-peer networking allows you to be online in a group of peers and to be able to share your experiences (such as a sunset, a rock concert, or a vacation cruise) while they are occurring.

Content Distribution

- Text messages

Peer-to-peer networking can allow for the dissemination of text-based information in the form of files or messages to a large group of users. An example is a news list.

-

- Audio and video

Peer-to-peer networking can also allow for the dissemination of audio or video information to a large group of users, such as a large concert or company meeting. To distribute the content today, you must configure high-capacity servers to collect and distribute the load to hundreds or thousands of users. With peer-to-peer networking, only a handful of peers would actually get their content from the centralized servers. These peers would flood this information out to a few more people who send it to others, and so on. The load of distributing the content is distributed to the peers in the cloud. A peer that wants to receive the content would find the closest distributing peer and get the content from them.

-

- Distribution of product updates

Peer-to-peer networking can also provide an efficient mechanism to distribute software such as product updates (security updates and service packs). A peer that has a connection to a software distribution server can obtain the product update and propagate it to the other members of its group.

-

Distributed Processing

- Division and distribution of a task

A large computing task can first be divided into separate smaller computing tasks well suited to the computing resources of a peer. A peer could do the dividing of the large computing task. Then, peer-to-peer networking can distribute the individual tasks to the separate peers in the group. Each peer performs its computing task and reports its result back to a centralized accumulation point.

- Aggregation of computer resources

Another way to utilize peer-to-peer networking for distributed processing is to run programs on each peer that run during idle processor times and are part of a larger computing task that is coordinated by a central server. By aggregating the processors of multiple computers, peer-to-peer networking can turn a group of peer computers into a large parallel processor for large computing tasks.

-

See Also

System.Net.PeerToPeer.Collaboration

Changes to NTLM authentication for HttpWebRequest in Version 3.5 SP1

7/29/2017 • 4 min to read • Edit Online

Security changes were made in .NET Framework version 3.5 SP1 and later that affect how integrated Windows authentication is handled by the HttpWebRequest, HttpListener, NegotiateStream, and related classes in the System.Net namespace. These changes can affect applications that use these classes to make web requests and receive responses where integrated Windows authentication based on NTLM is used. This change can impact web servers and client applications that are configured to use integrated Windows authentication.

Overview

The design of integrated Windows authentication allows for some credential responses to be universal, meaning they can be re-used or forwarded. If this particular design feature is not needed, then the authentication protocols should carry target specific information as well as channel specific information. Services can then provide extended protection to ensure that credential responses contain service specific information such as a Service Principal Name (SPN). With this information in the credential exchanges, services are able to better protect against malicious use of credential responses that might have been improperly obtained.

Multiple components in the System.Net and System.Net.Security namespaces perform integrated Windows authentication on behalf of a calling application. This section describes changes to System.Net components to add extended protection in their use of integrated Windows authentication.

Changes

The NTLM authentication process used with integrated Windows authentication includes a challenge issued by the destination computer and sent back to the client computer. When a computer receives a challenge it generated itself, the authentication will fail unless the connection is a loop back connection (IPv4 address 127.0.0.1, for example).

When accessing a service running on an internal Web server, it is common to access the service using a URL similar to http://contoso/service or https://contoso/service. The name "contoso" is often not the computer name of the computer on which the service is deployed. The System.Net and related namespaces support using Active Directory, DNS, NetBIOS, the local computer's hosts file (typically WINDOWS\system32\drivers\etc\hosts, for example), or the local computer's lmhosts file (typically WINDOWS\system32\drivers\etc\lmhosts, for example) to resolve names to addresses. The name "contoso" is resolved so that requests sent to "contoso" are sent to the appropriate server computer.

When configured for large deployments, it is also common for a single virtual server name to be given to the deployment with the underlying machine names never used by client applications and end users. For example, you might call the server www.contoso.com, but on an internal network simply use "contoso". This name is called the Host header in the client web request. As specified by the HTTP protocol, the Host request-header field specifies the Internet host and port number of the resource being requested. This information is obtained from the original URI given by the user or referring resource (generally an HTTP URL). On .NET Framework version 4, this information can also be set by the client using the new Host property.

The AuthenticationManager class controls the managed authentication components ("modules") that are used by WebRequest derivative classes and the WebClient class. The AuthenticationManager class provides a property that exposes a System.Net.AuthenticationManager.CustomTargetNameDictionary object, indexed by URI string, for applications to supply a custom SPN string to be used during authentication.

Version 3.5 SP1 now defaults to specifying the host name used in the request URL in the SPN in the NTLM (NT LAN Manager) authentication exchange when the CustomTargetNameDictionary property is not set. The host name used in the request URL may be different from the Host header specified in the System.Net.HttpRequestHeader in the client request. The host name used in the request URL may be different from the actual host name of the server, the machine name of the server, the computer's IP address, or the loopback address. In these cases, Windows will fail the authentication request. To address the issue, we need to notify Windows that the host name used in the request URL in the client request ("contoso", for example) is actually an alternate name for the local computer.

There are several possible methods for a server application to work around this change. The recommended approach is to map the host name used in the request URL to the `BackConnectionHostNames` key in the registry on the server. The `BackConnectionHostNames` registry key is normally used to map a host name to a loopback address. The steps are listed below.

To specify the host names that are mapped to the loopback address and can connect to Web sites on a local computer, follow these steps:

1. Click Start, click Run, type regedit, and then click OK.
2. In Registry Editor, locate and then click the following registry key:

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\MSV1_0`

3. Right-click MSV1_0, point to New, and then click Multi-String Value.
4. Type `BackConnectionHostNames`, and then press ENTER.
5. Right-click `BackConnectionHostNames`, and then click Modify.
6. In the Value data box, type the host name or the host names for the sites (the host name used in the request URL) that are on the local computer, and then click OK.
7. Quit Registry Editor, and then restart the IISAdmin service and run IISReset.

A less secure work around is to disable the loop back check, as described in http://support.microsoft.com/kb/896861. This disables the protection against reflection attacks. So it is better to constrain the set of alternate names to only those you expect the machine to actually use.

See Also

System.Net.AuthenticationManager.CustomTargetNameDictionary
System.Net.HttpRequestHeader
System.Net.HttpWebRequest.Host

Integrated Windows Authentication with Extended Protection

7/29/2017 • 11 min to read • Edit Online

Enhancements were made that affect how integrated Windows authentication is handled by the HttpWebRequest, HttpListener, SmtpClient, SslStream, NegotiateStream, and related classes in the System.Net and related namespaces. Support was added for extended protection to enhance security.

These changes can affect applications that use these classes to make web requests and receive responses where integrated Windows authentication is used. This change can also impact web servers and client applications that are configured to use integrated Windows authentication.

These changes can also affect applications that use these classes to make other types of requests and receive responses where integrated Windows authentication is used.

The changes to support extended protection are available only for applications on Windows 7 and Windows Server 2008 R2. The extended protection features are not available on earlier versions of Windows.

Overview

The design of integrated Windows authentication allows for some credential challenge responses to be universal, meaning they can be re-used or forwarded. The challenge responses should be constructed at a minimum with target specific information and preferably also with some channel specific information. Services can then provide extended protection to ensure that credential challenge responses contain service specific information such as a Service Principal Name (SPN). With this information in the credential exchanges, services are able to better protect against malicious use of credential challenge responses that might have been improperly used.

The extended protection design is an enhancement to authentication protocols designed to mitigate authentication relay attacks. It revolves around the concept of channel and service binding information.

The overall objectives are the following:

1. If the client is updated to support the extended protection, applications should supply a channel binding and service binding information to all supported authentication protocols. Channel binding information can only be supplied when there is a channel (TLS) to bind to. Service binding information should always be supplied.
2. Updated servers which are properly configured may verify the channel and service binding information when it is present in the client authentication token and reject the authentication attempt if the channel bindings do not match. Depending on the deployment scenario, servers may verify channel binding, service binding or both.
3. Updated servers have the ability to accept or reject down-level client requests that do not contain the channel binding information based on policy.

Information used by extended protection consists of one or both of the following two parts:

1. A Channel Binding Token or CBT.
2. Service Binding information in the form of a Service Principal Name or SPN.

Service Binding information is an indication of a client's intent to authenticate to a particular service endpoint. It is communicated from client to server with the following properties:

- The SPN value must be available to the server performing client authentication in clear text form.
- The value of the SPN is public.
- The SPN must be cryptographically protected in transit such that a man-in-the-middle attack cannot insert, remove or modify its value.

A CBT is a property of the outer secure channel (such as TLS) used to tie (bind) it to a conversation over an inner, client-authenticated channel. The CBT must have the following properties (also defined by IETF RFC 5056):

- When an outer channel exists, the value of the CBT must be a property identifying either the outer channel or the server endpoint, independently arrived at by both client and server sides of a conversation.
- Value of the CBT sent by the client must not be something an attacker can influence.
- No guarantees are made about secrecy of the CBT value. This does not however mean that the value of the service binding as well as channel binding information can always be examined by any other but the server performing authentication, as the protocol carrying the CBT may be encrypting it.
- The CBT must be cryptographically integrity protected in transit such that an attacker cannot insert, remove or modify its value.

Channel binding is accomplished by the client transferring the SPN and the CBT to the server in a tamperproof fashion. The server validates the channel binding information in accordance with its policy and rejects authentication attempts for which it does not believe itself to have been the intended target. This way, the two channels become cryptographically bound together.

To preserve compatibility with existing clients and applications, a server may be configured to allow authentication attempts by clients that do not yet support extended protection. This is referred to as a "partially hardened" configuration, in contrast to a "fully hardened" configuration.

Multiple components in the System.Net and System.Net.Security namespaces perform integrated Windows authentication on behalf of a calling application. This section describes changes to System.Net components to add extended protection in their use of integrated Windows authentication.

Extended protection is currently supported on Windows 7. A mechanism is provided so an application can determine if the operating system supports extended protection.

Changes to Support Extended Protection

The authentication process used with integrated Windows authentication, depending on the authentication protocol used, often includes a challenge issued by the destination computer and sent back to the client computer. Extended protection adds new features to this authentication process

The System.Security.Authentication.ExtendedProtection namespace provides support for authentication using extended protection for applications. The ChannelBinding class in this namespace represents a channel binding. The ExtendedProtectionPolicy class in this namespace represents the extended protection policy used by the server to validate incoming client connections. Other class members are used with extended protection.

For server applications, these classes include the following:

A ExtendedProtectionPolicy that has the following elements:

- An OSSupportsExtendedProtection property that indicates whether the operating system supports integrated windows authentication with extended protection.
- A PolicyEnforcement value that indicates when the extended protection policy should be enforced.
- A ProtectionScenario value that indicates the deployment scenario. This influences how extended protection

is checked.

- An optional ServiceNameCollection that contains the custom SPN list that is used to match against the SPN provided by the client as the intended target of the authentication.
- An optional ChannelBinding that contains a custom channel binding to use for validation. This scenario is not a common case

The System.Security.Authentication.ExtendedProtection.Configuration namespace provides support for configuration of authentication using extended protection for applications.

A number of feature changes were made to support extended protection in the existing System.Net namespace. These changes include the following:

- A new TransportContext class added to the System.Net namespace that represents a transport context.
- New EndGetRequestStream and GetRequestStream overload methods in the HttpWebRequest class that allow retrieving the TransportContext to support extended protection for client applications.
- Additions to the HttpListener and HttpListenerRequest classes to support server applications.

A feature change was made to support extended protection for SMTP client applications in the existing System.Net.Mail namespace:

- A TargetName property in the SmtpClient class that represents the SPN to use for authentication when using extended protection for SMTP client applications.

A number of feature changes were made to support extended protection in the existing System.Net.Security namespace. These changes include the following:

- New BeginAuthenticateAsClient and AuthenticateAsClient overload methods in the NegotiateStream class that allow passing a CBT to support extended protection for client applications.
- New BeginAuthenticateAsServer and AuthenticateAsServer overload methods in the NegotiateStream class that allow passing an ExtendedProtectionPolicy to support extended protection for server applications.
- A new TransportContext property in the SslStream class to support extended protection for client and server applications.

A SmtpNetworkElement property was added to support configuration of extended protection for SMTP clients in the System.Net.Security namespace.

Extended Protection for Client Applications

Extended protection support for most client applications happens automatically. The HttpWebRequest and SmtpClient classes support extended protection whenever the underlying version of Windows supports extended protection. An HttpWebRequest instance sends an SPN constructed from the Uri. By default, an SmtpClient instance sends an SPN constructed from the host name of the SMTP mail server.

For custom authentication, client applications can use the System.Net.HttpWebRequest.EndGetRequestStream(IAsyncResult, TransportContext) or System.Net.HttpWebRequest.GetRequestStream(TransportContext) methods in the HttpWebRequest class that allow retrieving the TransportContext and the CBT using the GetChannelBinding method.

The SPN to use for integrated Windows authentication sent by an HttpWebRequest instance to a given service can be overridden by setting the CustomTargetNameDictionary property.

The TargetName property can be used to set a custom SPN to use for integrated Windows authentication for the SMTP connection.

Extended Protection for Server Applications

HttpListener automatically provides mechanisms for validating service bindings when performing HTTP authentication.

The most secure scenario is to enable extended protection for HTTPS:// prefixes. In this case, set System.Net.HttpListener.ExtendedProtectionPolicy to an ExtendedProtectionPolicy with PolicyEnforcement set to WhenSupported or Always, and ProtectionScenario set to TransportSelected A value of WhenSupported puts HttpListener in partially hardened mode, while Always corresponds to fully hardened mode.

In this configuration when a request is made to the server through an outer secure channel, the outer channel is queried for a channel binding. This channel binding is passed to the authentication SSPI calls, which validate that the channel binding in the authentication blob matches. There are three possible outcomes:

1. The server's underlying operating system does not support extended protection. The request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

2. The SSPI call fails indicating that either the client specified a channel binding that did not match the expected value retrieved from the outer channel or the client failed to supply a channel binding when the extended protection policy on the server was configured for Always. In both cases, the request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

3. The client specifies the correct channel binding or is allowed to connect without specifying a channel binding since the extended protection policy on the server is configured with WhenSupported The request is returned to the application for processing. No service name check is performed automatically. An application may choose to perform its own service name validation using the ServiceName property, but under these circumstances it is redundant.

If an application makes its own SSPI calls to perform authentication based on blobs passed back and forth within the body of an HTTP request and wishes to support channel binding, it needs to retrieve the expected channel binding from the outer secure channel using HttpListener in order to pass it to native Win32 AcceptSecurityContext function. To do this, use the TransportContext property and call GetChannelBinding method to retrieve the CBT. Only endpoint bindings are supported. If anything other Endpoint is specified, a NotSupportedException will be thrown. If the underlying operating system supports channel binding, the GetChannelBinding method will return a ChannelBindingSafeHandle wrapping a pointer to a channel binding suitable for passing to AcceptSecurityContext function as the pvBuffer member of a SecBuffer structure passed in the `pInput` parameter. The Size property contains the length, in bytes, of the channel binding. If the underlying operating system does not support channel bindings, the function will return `null`.

Another possible scenario is to enable extended protection for HTTP:// prefixes when proxies are not used. In this case, set System.Net.HttpListener.ExtendedProtectionPolicy to an ExtendedProtectionPolicy with PolicyEnforcement set to WhenSupported or Always, and ProtectionScenario set to TransportSelected A value of WhenSupported puts HttpListener in partially hardened mode, while Always corresponds to fully hardened mode.

A default list of allowed service names is created based on the prefixes which have been registered with the HttpListener. This default list can be examined through the DefaultServiceNames property. If this list is not comprehensive, an application can specify a custom service name collection in the constructor for the ExtendedProtectionPolicy class which will be used instead of the default service name list.

In this configuration, when a request is made to the server without an outer secure channel authentication proceeds normally without a channel binding check. If the authentication succeeds, the context is queried for the service name that the client provided and validated against the list of acceptable service names. There are four possible outcomes:

1. The server's underlying operating system does not support extended protection. The request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

2. The client's underlying operating system does not support extended protection. In the WhenSupported configuration, the authentication attempt will succeed and the request will be returned to the application. In the Always configuration, the authentication attempt will fail. The request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

3. The client's underlying operating system supports extended protection, but the application did not specify a service binding. The request will not be exposed to the application, and an unauthorized (401) response will be returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

4. The client specified a service binding. The service binding is compared to the list of allowed service bindings. If it matches, the request is returned to the application. Otherwise, the request will not be exposed to the application, and an unauthorized (401) response will be automatically returned to the client. A message will be logged to the HttpListener trace source specifying the reason for the failure.

If this simple approach using an allowed list of acceptable service names is insufficient, an application may provide its own service name validation by querying the ServiceName property. In cases 1 and 2 above, the property will return `null`. In case 3, it will return an empty string. In case 4, the service name specified by the client will be returned.

These extended protection features can also be used by server applications for authentication with other types of requests and when trusted proxies are used.

See Also

System.Security.Authentication.ExtendedProtection
System.Security.Authentication.ExtendedProtection.Configuration

NAT Traversal using IPv6 and Teredo

7/29/2017 • 4 min to read • Edit Online

Enhancements were made that provide support for Network Address Translation (NAT) traversal. These changes are designed for use with IPv6 and Teredo, but they are also applicable to other IP tunneling technologies. These enhancements affect classes in the System.Net and related namespaces.

These changes can affect client and server applications that plan to use IP tunneling technologies.

The changes to support NAT traversal are available only for applications using .NET Framework version 4. These features are not available on earlier versions of the .NET Framework.

Overview

The Internet Protocol version 4 (IPv4) defined an IPv4 address as 32 bits long. As a result, IPv4 supports approximately 4 billion unique IP addresses (2^32). As the number of computers and network devices on the Internet expanded in the 1990s, the limits of the IPv4 address space became apparent.

One of several techniques used to extend the lifetime of IPv4 has been to deploy NAT to allow a single unique public IP address to represent a large number of private IP addresses (private Intranet). The private IP addresses behind the NAT device share the single public IPv4 address. The NAT device may be a dedicated hardware device (an inexpensive Wireless Access Point and router, for example) or a computer running a service to provide NAT. A device or service for this public IP address translates IP network packets between the public Internet and the private Intranet.

This scheme works well for client applications running on the private Intranet that send requests to other IP addresses (usually servers) on the Internet. The NAT device or server can keep a mapping of client requests so when a response is returned it knows where to send the response. But this scheme poses problems for applications running in the private Intranet behind the NAT device that want to provide services, listen for packets, and respond. This is particularly the case for peer-to-peer applications.

The IPv6 protocol defined an IPv4 address as 128 bits long. As a result, IPv6 supports very a large IP address space of 3.2 x 10^38 unique addresses (2^128). With an address space of this size, it is possible for every device connected to the Internet to be given a unique address. But there are problems. Much of the world is still using only IPv4. In particular, many of the existing routers and wireless access points used by small companies, organizations, and households do not support IPv6. Also some Internet service providers that serve these customers either do not support or have not configured support for IPv6.

Several IPv6 transition technologies have been developed to tunnel IPv6 addresses in an IPv4 packet. These technologies include 6to4, ISATAP, and Teredo tunnels that provide address assignment and host-to-host automatic tunneling for unicast IPv6 traffic when IPv6 hosts must traverse IP4 networks to reach other IPv6 networks. IPv6 packets are sent tunneled as IPv4 packets. Several tunneling techniques are being used that allow NAT traversal for IPv6 addresses through a NAT device.

Teredo is one of the IPv6 transition technologies which brings IPv6 connectivity to IPv4 networks. Teredo is documented in RFC 4380 published by the Internet Engineering Task Force (IETF). Windows XP SP2 and later provide support for a virtual Teredo adapter which can provide a public IPv6 address in the range 2001:0::/32. This IPv6 address can be used to listen for incoming connections from the Internet and can be provided to IPv6 enabled clients that wish to connect to the listening service. This frees an application from worrying about how to address a computer behind a NAT device, since the application can just connect to it using its IPv6 Teredo address.

Enhancements to Support NAT Traversal and Teredo

Enhancements are added to the System.Net, System.Net.NetworkInformation, and System.Net.Sockets namespaces for supporting NAT traversal using IPv6 and Teredo.

Several methods are added to the System.Net.NetworkInformation.IPGlobalProperties class to get the list of unicast IP addresses on the host. The BeginGetUnicastAddresses method begins an asynchronous request to retrieve the stable unicast IP address table on the local computer. The EndGetUnicastAddresses method ends a pending asynchronous request to retrieve the stable unicast IP address table on the local computer. The GetUnicastAddresses method is a synchronous request to retrieve the stable unicast IP address table on the local computer, waiting until the address table stabilizes if necessary.

The System.Net.IPAddress.IsIPv6Teredo property can be used to determine if an IPAddress is an IPv6 Teredo address.

Using these new IPGlobalProperties class methods in combination with the IsIPv6Teredo property allows an application to easily find the Teredo address. An application normally only needs to know the local Teredo address if it is communicating this information to remote applications. For example, a peer-to-peer application might send all of its IPv6 addresses to a matchmaking server which can then forward them to others peers to enable direct communication.

An application should normally set its listening service to listen on System.Net.IPAddress.IPv6Any rather than on the local Teredo address. So if a remote client or peer has a direct IPv6 route to the host of the listening service, the client or peer can connect directly using IPv6 and not have to use Teredo to tunnel packets.

For TCP applications, the System.Net.Sockets.TcpListener class has an AllowNatTraversal method to enable NAT traversal. For UDP applications, the System.Net.Sockets.UdpClient class has an AllowNatTraversal method to enable NAT traversal.

For applications that use the System.Net.Sockets.Socket and related classes, the GetSocketOption and SetSocketOption methods can be used with the System.Net.Sockets.SocketOptionName.IPProtectionLevel socket option to query, enable, or disable NAT traversal.

See Also

System.Net.IPAddress.IsIPv6Teredo
System.Net.NetworkInformation.IPGlobalProperties.BeginGetUnicastAddresses
System.Net.NetworkInformation.IPGlobalProperties.EndGetUnicastAddresses
System.Net.NetworkInformation.IPGlobalProperties.GetUnicastAddresses
System.Net.Sockets.IPProtectionLevel
System.Net.Sockets.Socket.SetIPProtectionLevel
System.Net.Sockets.TcpListener.AllowNatTraversal
System.Net.Sockets.UdpClient.AllowNatTraversal

Network Isolation for Windows Store Apps

7/29/2017 • 1 min to read • Edit Online

Classes in the System.Net, System.Net.Http, and System.Net.Http.Headers namespaces can be used to develop Windows Store apps or desktop apps. When used in a Windows Store app, classes in these namespaces are affected by network isolation, part of the application security model used by the Windows 8. The appropriate network capabilities must be enabled in the app manifest for a Windows Store app for the system to allow network access.

Checklist for Network Isolation

Use this checklist to be sure that network isolation is configured for your Windows Store app.

1. Determine the direction of network access requests needed by the app. This can be either outbound client-initiated requests or inbound unsolicited requests or it could be a combination of both of these network request types.
2. Determine the type of network resources that that app will communicate with. An app may need to communicate with trusted resources on a Home or Work network. An app might need to communicate with resources on the Internet. An app might need access to both types of network resources.
3. Configure the minimum-required networking isolation capabilities in the app manifest.
4. Deploy and run your app to test it using the network isolation tools provided for troubleshooting.

For more detailed information on how to configure network capabilities and isolation tools used for troubleshooting network isolation, see How to configure network isolation capabilities in the Windows 8.x Store developer documentation.

See Also

Connecting to a web service
Guidelines and checklist for network isolation
Quickstart: Connecting using HttpClient
How to use HttpClient handlers
How to secure HttpClient connections
HttpClient Sample

Network Programming Samples

7/29/2017 • 1 min to read • Edit Online

This section contains descriptions and links to downloadable network programming samples that use classes in the System.Net, System.Net.Cache, System.Net.Configuration, System.Net.Mail, System.Net.Mime, System.Net.NetworkInformation, System.Net.PeerToPeer, System.Net.Security, System.Net.Sockets, and related namespaces.

In This Section

Download Progress Indicator Technology Sample
Shows how to display the progress of a file download.

FTP Client Technology Sample
Shows how to upload and download files to and from an FTP server.

HttpListener Technology Sample
Shows how to process HTTP requests from within an application.

HttpListener ASPX Host Application Sample
Demonstrates how to use the features of the HttpListener class to create an HTTP server that routes calls to a hosted ASP.NET application.

Mailer Technology Sample
Shows how to send email messages from a client application.

NetStat Tool Technology Sample
Demonstrates the NCLNetStat network information tool.

Network Information Technology Sample
Shows how to monitor and display network information.

Ping Client Technology Sample
Demonstrates a client application that can ping a remote host.

WebClient Technology Sample
Demonstrates how to perform common operations, such as the upload or download of files or data.

Secure Streams Sample
Shows how to use a secure stream to communicate between a client and a server.

IPv6 Sockets Sample
Demonstrates how to use sockets when IPv6 is enabled.

FTP Explorer Technology Sample
Demonstrates how to list the contents of an FTP server.

Socket Performance Technology Sample
Shows how to use enhancements in the Socket class to build a server application that uses asynchronous network I/O to achieve the highest performance.

PeerToPeer Technology Sample
Shows how to use the new classes in the System.Net.PeerToPeer namespace to register and publish a peer name and then resolve a peer name.

Reference

System.Net

System.Net.NetworkInformation

System.Net.PeerToPeer

System.Net.Sockets

See Also

Network Programming in the .NET Framework
Network Programming How-to Topics
Networking Samples for .NET