Network Programming in the .NET Framework

.NET Framework (current version)

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

.NET Framework (current version)

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

.NET Framework (current version)

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 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 NetworkCredentials 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

.NET Framework (current version)

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 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 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 WebResponse.GetResponseStream method. You can use this stream like any other, as shown in the following example.

VB

```
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

.NET Framework (current version)

Applications create WebRequest instances through the 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

.NET Framework (current version)

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

VB

```
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

.NET Framework (current version)

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.

C#

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

VB

```
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.

C#

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

VB

```
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.

C#

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

VB

```
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.

C#

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

VB

```
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 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.

C#

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

VB

```
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**.

C#

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

VB

```
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.

C#

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

VB

```
response.Close()
```

VB

```
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

.NET Framework (current version)

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.

C#

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

VB

```
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.

C#

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

VB

```
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.

C#

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

VB

```
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.

C#

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

VB

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

4. Set the **ContentLength** property.

C#

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

VB

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

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

C#

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

VB

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

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

C#

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

VB

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

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

C#

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

VB

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

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

C#

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

VB

```
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.

C#

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

VB

```
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 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.

C#

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

VB

```
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**.

C#

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

VB

```
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.

C#

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

VB

```
response.Close()
```

VB

```
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

.NET Framework (current version)

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

VB

```
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

.NET Framework (current version)

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 Stream.Write and 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.

VB

```
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.

VB

```
' 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

.NET Framework (current version)

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 Stream.BeginRead to read the stream returned by the 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.

VB

```
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)

      ' 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

.NET Framework (current version)

The WebRequest and WebResponse classes throw both system exceptions (such as ArgumentException) and Web-specific exceptions (which are WebExceptions 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.
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**.

VB

```
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 SocketExceptions 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

.NET Framework (current version)

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

.NET Framework (current version)

This example shows how to register a protocol specific class that 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.

VB

```
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

.NET Framework (current version)

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

VB

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

See Also

Programming Pluggable Protocols

Deriving from WebRequest

.NET Framework (current version)

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 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 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 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 HttpWebResponse 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 RequestCanceled.

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

.NET Framework (current version)

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 HttpWebRequest.GetResponse or 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 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 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

.NET Framework (current version)

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

.NET Framework (current version)

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 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.ProtocolError. 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

.NET Framework (current version)

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

.NET Framework (current version)

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

VB

```
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()
```

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

.NET Framework (current version)

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.

VB

```
' 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`.

VB

```
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

.NET Framework (current version)

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

.NET Framework (current version)

7ce550d6-8f7c-4ea7-add8-5bc27a7b51be#tskhowtoassignuserinformationtogroupconnectionsanchor1

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.

C#

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

VB

```
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`.

C#

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

VB

```
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.

C#

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

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

VB

```
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.

C#

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

VB

```
MyWebResponse.Close()
```

Example

VB

```
' 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

.NET Framework (current version)

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

.NET Framework (current version)

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.

VB

```
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
```

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.

VB

```
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
```

See Also

TCP/UDP

Using UDP Services

.NET Framework (current version)

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.

VB

```
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
```

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.

VB

```
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
```

See Also

UdpClient
IPAddress
TCP/UDP

Sockets

.NET Framework (current version)

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
TCP/UDP
Using Application Protocols

How to: Create a Socket

.NET Framework (current version)

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.

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

VB

```
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:

VB

```
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

.NET Framework (current version)

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 IPHostEnty 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.

VB

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

VB

```
Dim ipe As 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.

VB

```
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
```

See Also

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

Using a Synchronous Client Socket

.NET Framework (current version)

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 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.

VB

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

VB

```
Dim bytes(1024) As Byte
Dim 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.

VB

```
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

.NET Framework (current version)

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 BeginConnect 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`.

VB

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

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

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.

VB

```
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
```

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.

VB

```
    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
```

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`.

VB

```
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
```

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.

VB

```
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
```

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.

VB

```
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
```

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.

VB

```
    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
```

See Also

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

Listening with Sockets

.NET Framework (current version)

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.

VB

```
Dim ipHostInfo As IPHostEntry = Dns.Resolve(Dns.GetHostName())
Dim ipAddress As IPAddress = ipHostInfo.AddressList(0)
Dim localEndPoint As 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**.

VB

```
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

.NET Framework (current version)

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.

VB

```
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()
```

See Also

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

Using an Asynchronous Server Socket

.NET Framework (current version)

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.

VB

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

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.

VB

```
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.

VB

```
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
```

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`.

VB

```
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
```

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.

VB

```
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
```

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`.

VB

```
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
```

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.

VB

```
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
```

See Also

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

Socket Code Examples

.NET Framework (current version)

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

.NET Framework (current version)

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.

VB

```
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
```

See Also

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

Synchronous Server Socket Example

.NET Framework (current version)

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.

VB

```
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
```

See Also

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

Asynchronous Client Socket Example

.NET Framework (current version)

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.

VB

```
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.
        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
```

See Also

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

Asynchronous Server Socket Example

.NET Framework (current version)

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.

VB

```
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
```

See Also

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

FTP

.NET Framework (current version)

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

.NET Framework (current version)

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

C#

```
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

.NET Framework (current version)

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

C#

```
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

.NET Framework (current version)

This sample shows how to list the directory contents of an FTP server.

C#

```
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

.NET Framework (current version)

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 ServicePointManager.MaxServicePointIdleTime property. Set 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**.
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.
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