# **Service Channel-Level Programming**

* 03/29/2017
* 4 minutes to read

This topic describes how to write a Windows Communication Foundation (WCF) service application without using the [System.ServiceModel.ServiceHost](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.servicehost) and its associated object model.

## **Receiving Messages**

To be ready to receive and process messages, the following steps are required:

Create a binding.

Build a channel listener.

Open the channel listener.

Read the request and send a reply.

Close all channel objects.

#### **Creating a Binding**

The first step in listening for and receiving messages is creating a binding. WCF ships with several built-in or system-provided bindings that can be used directly by instantiating one of them. In addition, you can also create your own custom binding by instantiating a CustomBinding class which is what the code in listing 1 does.

The code example below creates an instance of [System.ServiceModel.Channels.CustomBinding](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.custombinding) and adds an [System.ServiceModel.Channels.HttpTransportBindingElement](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.httptransportbindingelement) to its Elements collection which is a collection of binding elements that are used to build the channel stack. In this example, because the elements collection has only the [HttpTransportBindingElement](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.httptransportbindingelement), the resulting channel stack has only the HTTP transport channel.

#### **Building a ChannelListener**

After creating a binding, we call [Binding.BuildChannelListener](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.binding.buildchannellistener) to build the channel listener where the type parameter is the channel shape to create. In this example we are using [System.ServiceModel.Channels.IReplyChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ireplychannel) because we want to listen for incoming messages in a request/reply message exchange pattern.

[IReplyChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ireplychannel) is used for receiving request messages and sending back reply messages. Calling [IReplyChannel.ReceiveRequest](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ireplychannel.receiverequest) returns an [System.ServiceModel.Channels.IRequestChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.irequestchannel), which can be used to receive the request message and to send back a reply message.

When creating the listener, we pass the network address on which it listens, in this case http://localhost:8080/channelapp. In general, each transport channel supports one or possibly several address schemes, for example, the HTTP transport supports both http and https schemes.

We also pass an empty [System.ServiceModel.Channels.BindingParameterCollection](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.bindingparametercollection) when creating the listener. A binding parameter is a mechanism to pass parameters that control how the listener should be built. In our example, we are not using any such parameters so we pass an empty collection.

#### **Listening for Incoming Messages**

We then call [ICommunicationObject.Open](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.icommunicationobject.open) on the listener and start accepting channels. The behavior of [IChannelListener<TChannel>.AcceptChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ichannellistener-1.acceptchannel) depends on whether the transport is connection-oriented or connection-less. For connection-oriented transports, [AcceptChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ichannellistener-1.acceptchannel) blocks until a new connection request comes in at which point it returns a new channel that represents that new connection. For connection-less transports, such as HTTP, [AcceptChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ichannellistener-1.acceptchannel) returns immediately with the one and only channel that the transport listener creates.

In this example, the listener returns a channel that implements [IReplyChannel](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ireplychannel). To receive messages on this channel we first call [ICommunicationObject.Open](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.icommunicationobject.open) on it to place it in a state ready for communication. We then call [ReceiveRequest](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ireplychannel.receiverequest) which blocks until a message arrives.

#### **Reading the Request and Sending a Reply**

When [ReceiveRequest](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.ireplychannel.receiverequest) returns a [RequestContext](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.requestcontext), we get the received message using its [RequestMessage](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.requestcontext.requestmessage) property. We write out the message’s action and body content, (which we assume is a string).

To send a reply, we create a new reply message in this case passing back the string data we received in the request. We then call [Reply](https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.channels.requestcontext.reply) to send the reply message.

#### **Closing Objects**

To avoid leaking resources, it is very important to close objects used in communications when they are no longer required. In this example we close the request message, the request context, the channel and the listener.

The following code example shows a basic service in which a channel listener receives only one message. A real service keeps accepting channels and receiving messages until the service exits.

C#Copy

using System;using System.ServiceModel.Channels;namespace ProgrammingChannels

{class Service

{static void RunService()

{

//Step1: Create a custom binding with just TCP.

BindingElement[] bindingElements = new BindingElement[2];

bindingElements[0] = new TextMessageEncodingBindingElement();

bindingElements[1] = new HttpTransportBindingElement();

CustomBinding binding = new CustomBinding(bindingElements);

//Step2: Use the binding to build the channel listener.

IChannelListener<IReplyChannel> listener =

binding.BuildChannelListener<IReplyChannel>(

new Uri("http://localhost:8080/channelapp"),

new BindingParameterCollection());

//Step3: Listening for messages.

listener.Open();

Console.WriteLine(

"Listening for incoming channel connections");

//Wait for and accept incoming connections.

IReplyChannel channel = listener.AcceptChannel();

Console.WriteLine("Channel accepted. Listening for messages");

//Open the accepted channel.

channel.Open();

//Wait for and receive a message from the channel.

RequestContext request= channel.ReceiveRequest();

//Step4: Reading the request message.

Message message = request.RequestMessage;

Console.WriteLine("Message received");

Console.WriteLine("Message action: {0}",

message.Headers.Action);

string data=message.GetBody<string>();

Console.WriteLine("Message content: {0}",data);

//Send a reply.

Message replymessage=Message.CreateMessage(

binding.MessageVersion,

"http://contoso.com/someotheraction",

data);

request.Reply(replymessage);

//Step5: Closing objects.

//Do not forget to close the message.

message.Close();

//Do not forget to close RequestContext.

request.Close();

//Do not forget to close channels.

channel.Close();

//Do not forget to close listeners.

listener.Close();

}public static void Main()

{

Service.RunService();

Console.WriteLine("Press enter to exit");

Console.ReadLine();

}

}

}

<https://docs.microsoft.com/en-us/dotnet/framework/wcf/extending/service-channel-level-programming>