Publish Events Defensively

by Juval Löwy

Technology Toolbox

- ☑ VB.NET
- ☑ C#
- ☐ SQL Server 2000
- ☑ ASP.NET
- ☐ XML
- ☐ VB6

Q: Publish Events Defensively

When I use a delegate to publish events, the subscribers raise exceptions occasionally that abort the publishing sequence. I also get another exception if there are no subscribers. Is there a way to continue publishing events defensively, even in the face of exceptions?

A:

You can use quite a few techniques to publish events defensively. When you publish events in C#, trying to fire an event on a delegate that has no subscribers in its internal list is a common source of exceptions. If no client subscribes to the event, the delegate's target list is empty, and the delegate's value is set to null automatically. As a result, an exception is thrown when the publisher tries to access a nulled delegate. In C#, the publisher should always check an event delegate to see if it's null before attempting to publish:

```
public class MySource
{
   public event
      EventHandler
      MyEvent;
   public void FireEvent()
   {
      if(MyEvent != null)
         MyEvent(this,
         EventArgs.Empty);
   }
}
```

In VB.NET, you needn't check the delegate's value, because the RaiseEvent statement can accept an empty delegate without throwing an exception:

```
Public Class MySource
   Public Event MyEvent As EventHandler
   Public Sub FireEvent()
      RaiseEvent MyEvent(Me,
         EventArgs.Empty)
   End Sub
End Class
```

Exceptions that the subscribers throw are the second source of exceptions. Some subscribers might encounter an exception in their handling of the event, not handle it, and cause the publisher to crash. For this reason, you should always publish inside a try/catch block (see Listing 1).

The code in Listing 1 aborts publishing the event if one of the subscribers throws an excep-

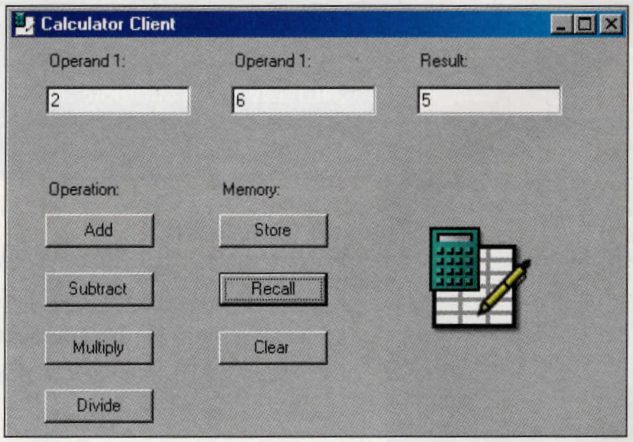

Figure 1 Create a Calculator Client. The calculator client simply exercises the various Web methods in the calculator Web service. You use the same textbox both for the arithmetic operations and for the memory content.

Go Online!

Use these Locator+ codes at www.visualstudiomagazine.com to go directly to these resources.

Download

VS0305QA Download the code for this article. It includes the EventsHelper class and the Web services state demo.

Discuss

VS0305QA_D Discuss this article in the C# forum.

Read More

VS0305QA_T Read this article online.
VS0204JL_T "Tame .NET Events" by Juval Löwy
NM0101JL_T "Web Services Hurdle the Firewall" by Juval Löwy
VS02ENAN_T ASP.NET, "Write More Powerful Web Services," by Francesco Balena

C# • Publish Defensively

```
public class MySource
{
  public event EventHandler MyEvent;
  public void FireEvent()
  {
    try
    {
      if(MyEvent != null)
        MyEvent(this,EventArgs.Empty);
    }
    catch
    {
      //handle exceptions
    }
  }
}
```

Listing 1 Always publish events within a try/catch block to handle exceptions that the subscribers throw. In C#, you also need to check the delegate for null (no subscribers).

tion. However, you want to continue publishing sometimes, even if one of the subscribers throws an exception. To do this, you need to iterate manually over the internal delegate list that the delegate maintains and catch any exceptions that the individual delegates in the list throw. You access the internal list by using a special method called GetInvocationList(), which every delegate supports:

```
public virtual Delegate[]
  GetInvocationList();
```

GetInvocationList() returns a collection of delegates you can iterate over (see Listing 2). You use a foreach statement to traverse the delegate collection, trying to deliver the event manually to each subscriber. If a particular subscriber throws an exception, then you catch it and continue to the next subscriber. The problem with the publishing code in Listing 2 is that it's not generic; it's specific to that particular delegate definition, and you must duplicate it in every case where you want fault isolation between the publisher and the subscribers. However, you can write a generic helper class that publishes to any delegate, passes any argument collection, and catches potential exceptions. The EventsHelper class provides the static Fire() method (see Listing 3 and download the source code from the *VSM* Web site; see the Go Online box for details):

```
public class EventsHelper
{
  public static void Fire(Delegate
    del,params object[] args);
}
```

You use Fire to fire any type of event defensively. Implementing EventsHelper involves two key elements. The first is its ability to invoke any delegate. You do this by using the DynamicInvoke() method that every delegate provides:

```
public object DynamicInvoke(object[]
  args);
```

DynamicInvoke() invokes the delegate, passing it a collection of arguments. The second key in implementing EventsHelper is passing it an open-ended number of objects as arguments for the subscribers. You do this by using the C# param parameter modifier (ParamArray in VB.NET), which allows simple in-lining of objects as parameters.

As a result, using EventsHelper is elegant and straightforward: Simply pass it the delegate to invoke, and the parameters. For example, suppose you have the delegate SomeDelegate:

```
public delegate void SomeDelegate(int
  num,string str);
```

This is the publishing code for SomeDelegate:

```
public class MySource
{
  public event SomeDelegate SomeEvent;
```

C# • Catch Exceptions and Keep Publishing

```
public class MySource
{
  public event EventHandler MyEvent;
  public void FireEvent()
  {
    if(MyEvent == null)
    {
      return;
    }
    Delegate[] delegates =
      MyEvent.GetInvocationList();
    foreach(Delegate del in delegates)
    {
      EventHandler sink = (EventHandler)del;
      try
      {
        sink(this,EventArgs.Empty);
      }
      catch{}
    }
  }
}
```

Listing 2 You can publish continuously in the face of exceptions that the subscribers throw. By iterating over the delegate's internal list of invocation targets, you can invoke them individually, catch any exception, and publish to the next subscriber.

C# • Publish to Any Delegate

```
public class EventsHelper
{
  public static void Fire(Delegate del,params
    object[] args)
  {
    if(del == null)
    {
      return;
    }
    Delegate[] delegates =
      del.GetInvocationList();
    foreach(Delegate sink in delegates)
    {
      try
      {
        sink.DynamicInvoke(args);
      }
      catch{}
    }
  }
}
```

Listing 3 You can use EventsHelper to publish any collection of arguments to any delegate, without propagating exceptions to the publisher.

C# • Build a State-Aware Web Service

```
public class CalculatorEx : WebService
{
  [WebMethod(EnableSession=true)]
  public int MemoryRecall()
  {
    int memory = 0;
    object state = Session["Calculator
      Memory"];
    if(state != null)
    {
      memory = (int)state;
    }
    return memory;
  }
  [WebMethod(EnableSession=true)]
  public void MemoryStore(int num)
  {
    Session["Calculator Memory"] = num;
  }
  [WebMethod(EnableSession=true)]
  public void MemoryReset()
  {
    Session["Calculator Memory"] = 0;
  }
  [WebMethod]
  public int Add(int num1,int num2)
  {
    return num1 + num2;
  }
  /* Rest of the arithmetic operation method */
}
```

Listing 4 This Web service stores its state in the base WebService class' Session property. The Session property provides an indexer to store and load the state. You also must enable session management in each Web method that requires access to the session object.

```
  public void FireEvent(int num,
    string str)
  {
    EventsHelper.Fire(
      SomeEvent,num,str);
  }
}
```

Q: Manage Web Services State

How do I maintain state in my Web service if .NET instantiates a new Web service object for every Web method call?

A:

Web services are single-call objects: For every Web method call, .NET instantiates a new object and lets it handle the call. As a result, if you want to have state in your Web service object, you must be state-aware—that is, manage your Web service state proactively. You retrieve the state from some repository at the beginning of a Web method; you work on the state during the method; and you must save the state just before returning from the Web method. You can store the state anywhere you like—in a file, on another machine, in a database, and so on.

ASP.NET provides built-in support for Web services state management by storing the state in a Session state variable. This is the same support that's available for Web forms. You specify the session storage in the Web service configuration file (Web.config). The available options are in-memory on the same machine where the Web service runs, on a designated machine, or in SQL Server. In-memory is the easiest option during development. A dedicated state machine caters to scalability when you use a server farm. SQL Server storage is available for cases when the Web service initiates a transaction, and it's useful in both single-machine and Web farm scenarios. The great thing about ASP.NET's state management is that you can develop it one way and deploy it in another, merely by changing the entries in the configuration file.

ASP.NET distinguishes automatically between different sessions by sending the client a cookie. Each distinct session has a different copy of the session variables. A Web form accesses the session storage through the Page class' Session property (of type HttpSessionState). To access the session object from a Web service, you can derive from the class WebService and access its Session property. Another option is to access the Session property of the current HTTP context:

```
HttpSessionState Session =
  HttpContext.Current.Session;
```

You must also enable session management support for your Web service explicitly. The HttpSessionState object provides an indexer that accepts an object as a key and an object as a value. You use the indexer for both state storage and retrieval.

For example, suppose you want to develop a calculator Web service that provides the four basic arithmetic operations for integers, as well as the capability (similar to a pocket calculator's memory feature) to store a number in memory on the server, to recall the number from memory, and to reset the memory (see Listing 4 and download the source code). The CalculatorEx Web service in Listing 4 derives from WebService so that you can access its Session property. You must set the WebMethod attribute's EnableSession property to true in every method that requires access to the session object, because session-state support is disabled by default. The ability to enable session state support on an individual-method basis caters to performance, because you don't need to pay for its overhead in methods that don't require it.

Use the indexer to store an object in the session state:

```
[WebMethod(EnableSession=true)]
public void MemoryStore(int num)
{
  Session["Calculator Memory"] =
    num;
}
```

You must decide on the key object; a string is fine for this example. Simply provide the key back to the indexer to load the state:

```
[WebMethod(EnableSession=true)]
public int MemoryRecall()
{
  int memory = 0;
  object state =
    Session["Calculator Memory"];
  if(state != null)
  {
    memory = (int)state;
  }
```

```
    return memory;
}
```

The session state returns null if it doesn't contain the specified key.

You can use the VS.NET-generated test page to build and run the Web service. If your browser supports session cookies (it likely does, unless you disabled it), the memory-feature methods will store, recall, and reset the memory. However, Web services are likely to be accessed by another machine, without any browser involved. You need to take an additional step on the client side to enable session state in this case. Create a new WinForms client, then add buttons and textboxes to exercise the calculator Web methods (see Figure 1). Next, add a reference to the Web service. If you run the client, you'll see that the arithmetic methods (such as Add) work fine, but the memory-management methods don't work. The reason is that by default, the Web service wrapper VS.NET creates doesn't support session cookies sent from the service. You must add that support manually. The Web service wrapper class derives indirectly from the HttpWebClientProtocol class, which has the CookieContainer property of the type System.Net.CookieContainer. By default, CookieContainer isn't initialized (set to null), so the cookie sent from the Web service isn't stored on the client side. You must create a cookie container (sometimes called a cookie jar) and associate it with the CookieContainer property. Open the Reference.cs file on the client side and create a new container in the wrapper class constructor:

```
public class CalculatorEx :
   SoapHttpClientProtocol
{
   public CalculatorEx()
   {
      CookieContainer = new
         System.Net.CookieContainer();
      Url =
         "http://localhost/
         CalculatorEx/
      CalculatorEx.asmx";
   }
   /* Rest of CalculatorEx */
}
```

If you test the client, you'll see the memory feature works now. VSM

Additional Resources

Programming .NET Components by Juval Löwy [O'Reilly & Associates, 2003, ISBN: 0596003471]

Juval Löwy is a software architect and the principal of IDesign, a consulting and training company focused on .NET design and .NET migration. Juval is Microsoft's regional director for the Silicon Valley, working with Microsoft on helping the industry adopt .NET. This article derives from his latest book, *Programming .NET Components* (O'Reilly & Associates). Juval speaks frequently at software-development conferences. Contact him at www.idesign.net.