ГЛАВА 37

Усовершенствованные веб-службы

В предыдущих двух главах достаточно подробно рассказывалось о том, как веб-службы работают с ASP.NET. Применяя описанные в этих главах технологии, вы сможете создавать веб-службы, которые будут отображать данные для других приложений и организаций, а также сможете использовать .NET и не .NET веб-службы в Интернете.

Однако на этом дело не заканчивается. В этой главе вы сможете расширить свои навыки по созданию веб-служб, ознакомившись со специальными технологиями, которые часто играют очень важную роль в сценариях реального мира. В частности, в этой главе будут рассмотрены следующие вопросы.

- *Асинхронный вызов веб-служб.* Вызовы веб-служб занимают определенное время, особенно если веб-сервер находится на другом конце земного шара или использует медленное сетевое соединение. Применение асинхронных вызовов позволит не прерывать работу, пока ожидается ответ.
- *Обеспечение безопасности веб-служб.* В части 4 рассказывалось о том, как можно обезопасить веб-страницы, дабы исключить возможность получения к ним доступа анонимных пользователей. Некоторые из этих технологий также могут быть применены и для защиты веб-служб.
- *Использование SOAP-расширений.* Инфраструктура веб-служб является удивительно гибкой благодаря модели SOAP-расширений, которая позволяет создавать компоненты и подключать их к процессу сериализации и десериализации SOAP. В этой главе мы покажем, что собой представляет базовое SOAP-расширение, а также вкратце расскажем о наборе инструментальных средств WSE (Web Services Enhancements), который с помощью SOAP-расширений обеспечивает поддержку для новых и развивающихся стандартов.

Все эти вопросы подразумевают применение концепций, которые описывались в двух предыдущих главах.

На заметку! Хотя в версии .NET 2.0 для программирования асинхронных вызовов используется та же модель, что и в .NET 1.*x*, поддержка асинхронных вызовов в прокси-классе, как будет показано в этой главе, претерпела некоторые изменения. Технологии, применяемые для обеспечения безопасности веб-служб и работы с SOAP-расширениями, остались прежними, хотя стандарты веб-служб продолжают развиваться и предлагаются в новых версиях поставляемого отдельно набора инструментальных средств WSE.

Асинхронные вызовы

Как рассказывалось в главе 35, .NET Framework избавляет программиста от сложностей, связанных с вызовом веб-службы, создавая для приложений прокси-класс. Код прокси-класса выглядит одинаково независимо от того, где находится веб-служба — на том же самом компьютере, в локальной сети или в Интернете.

Несмотря на внешнее сходство, лежащий в основе "рабочий" код, используемый для вызова веб-службы, очень отличается от внутрипроцессного вызова функции. Этот вызов не только должен быть упакован в SOAP-сообщение, но также и передан через сеть по протоколу HTTP. Из-за природы, присущей Интернету, время, затрачиваемое на вызов веб-службы, может значительно варьироваться от вызова к вызову. Ускорить вызов метода клиент никак не может, но он хотя бы может не простаивать попусту, пока ожидает ответ, а продолжать выполнять вычислительные операции, считывать данные из файловой системы или базы данных либо даже вызывать дополнительные веб-службы. Такую асинхронную схему намного труднее реализовать, но в определенных ситуациях это может того стоить.

В целом, асинхронная обработка имеет смысл в двух случаях.

- Если создается Windows-приложение. В этом случае асинхронный вызов позволяет пользовательскому интерфейсу продолжать реагировать на действия пользователя.
- Если требуется выполнить еще какую-нибудь дорогостоящую в плане вычислений работу или имеется доступ к другим ресурсам, которые характеризуются высокой степенью латентности. Асинхронный вызов позволит выполнить эту работу во время ожидания ответа. Особым является случай, когда требуется вызвать сразу несколько независимых веб-служб. В этой ситуации вы можете вызвать их все асинхронно, что существенно сократит общее время ожидания.

Важно, чтобы вы понимали, когда *не* следует использовать асинхронную схему. Асинхронные вызовы не сокращают время, затрачиваемое на получение ответа. Другими словами, в большинстве веб-приложений, которые используют веб-службы, у вас не будет причин вызвать веб-службы асинхронно, потому что вашему коду все равно придется дожидаться ответа, прежде чем он сможет визуализировать конечную страницу и отправить ее обратно клиенту. Однако при наличии необходимости вызывать сразу несколько веб-служб одновременно или возможности выполнять другие задачи, пока ожидается ответ, асинхронные вызовы могут на несколько миллисекунд сократить общее время обработки запросов.

В следующих разделах будет показано, как асинхронные вызовы позволяют сэкономить время в случае, когда речь идет о веб-клиенте, и обеспечить более быстро реагирующий пользовательский интерфейс в случае, когда речь идет о Windows-клиенте.

На заметку! Асинхронные вызовы работают несколько по-другому в ASP.NET 2.0. Прокси-класс больше не имеет встроенных методов `Begin`*`Xxx`*`()` и `End`*`Xxx`*`()`. Однако вы можете применять альтернативный подход для уведомления об асинхронных операциях с помощью событий, который встроен в прокси-класс и подходит только для долго работающих клиентов наподобие Windows-приложений.

Асинхронные делегаты

Асинхронные потоки могут использоваться в .NET несколькими способами. Все делегаты предоставляют методы BeginInvoke() и EndInvoke(), которые позволяют запускать их в одном из потоков в пуле потоков CLR-среды. Именно эта технология, которая является очень удобной и хорошо масштабируется, и будет рассматриваться в данном разделе. Альтернативная технология состоит в применении класса System.Threading.Thread для явного создания нового потока и тем самым получения возможности полностью контролировать его приоритет и время жизни.

Как уже упоминалось, делегаты — это поддерживающие безопасность типов указатели функций, которые формируют основу для .NET-событий. Вы создаете делегата, который ссылается на определенный метод, и затем можете использовать этот метод уже через данного делегата.

Первое, что потребуется сделать — это определить делегата на уровне пространства имен (если он уже не содержится в .NET-библиотеке классов). Например, ниже показан делегат, который может указывать на любой метод, принимающий и возвращающий целое число:

```
public delegate int DoSomethingDelegate(int input);
```

Теперь давайте создадим класс, включающий соответствующий этому делегату метод:

```
public class MyClass
{
 public int DoubleNumber(int input)
 {
   return input * 2;
 }
}
```

Мы можем создать переменную делегата, указывающую на метод с такой же сигнатурой. Вот как в этом случае будет выглядеть код:

```
MyOClass myObj = new MyClass();
// Создаем делегата, который указывает на метод myObj.DoubleNumber().
DoSomethingDelegate doSomething = new DoSomethingDelegate(myObj.DoubleNumber);
// Вызываем метод myObj.DoubleNumber()через делегата.
int doubleValue = doSomething(12);
```

Интересно то, что делегаты также имеют встроенные интеллектуальные средства организации поточной обработки. Каждый раз, когда определяется делегат (такой как DoSomethingDelegat из предыдущего примера), генерируется специальный класс делегата, который затем добавляется в соответствующую сборку. (Специальный класс делегата необходим потому, что код для каждого делегата разный и зависит от сигнатуры использованного метода.) При вызове метода через делегата на самом деле вызывается метод Invoke() класса делегата.

Метод Invoke() выполняет связанный с ним метод синхронно. Однако класс делегата также включает методы для асинхронного вызова, а именно BeginInvoke() и EndInvoke(). Когда используется метод BeginInvoke(), вызов возвращается сразу же, но не включает возвращаемого значения. Это говорит о том, что метод просто ставится в очередь для запуска в другом потоке. Вызывая метод BeginInvoke(), необходимо предоставить все параметры исходного метода плюс два дополнительных необязательных параметра, отвечающих за обратный вызов и состояние. Если эти детали (которые более подробно будут рассматриваться чуть позже в этой главе) не нужны, для них следует просто передать значение null:

```
IAsyncResult async = doSomething.BeginInvoke(12, null, null);
```

Метод BeginInvoke() не предоставляет возвращаемого значения лежащего в его основе метода. Вместо этого он возвращает объект IAsyncResult, который позволяет определить, когда завершилась асинхронная операция. Для обеспечения возможности получения результатов объект IAsyncResult передается в соответствующий метод EndInvoke() делегата. Метод EndInvoke() дожидается, пока завершится выполнение операции, если оно еще не завершилось, и затем предоставляет реальное возвращаемое значение. В случае если в методе, который выполнялся асинхронно, произошли необработанные ошибки, они все проявятся в коде после вызова метода EndInvoke().

Ниже показан измененный код из предыдущего примера, который теперь вызывает делегата асинхронно:

```
MyClass myObj = new MyClass();
// Создаем делегата, который указывает на метод myObj.DoubleNumber().
DoSomethingDelegate doSomething = new DoSomethingDelegate(myObj.DoubleNumber);
// Запускаем метод myObj.DoubleNumber()в другом потоке.
IAsyncResult handle = doSomething.BeginInvoke(originalValue, null, null);
// (Делаем что-нибудь, пока метод myObj.DoubleNumber() выполняется.)
// Извлекаем результаты и ожидаем (синхронно), если они еще не готовы.
int doubleValue = doSomething.EndInvoke(handle);
```

Чтобы воспользоваться преимуществами многопоточной обработки с помощью этой технологии, вы могли бы вызвать сразу несколько методов асинхронно, используя метод BeginInvoke(), а затем, прежде чем продолжить, вызывать во всех них метод EndInvoke().

Простой асинхронный вызов

Следующий пример иллюстрирует разницу между синхронным и асинхронным кодом. Чтобы протестировать этот пример, вам придется искусственно замедлить выполнение своего кода, чтобы сымитировать условия большой нагрузки или длительные по времени задачи. Сначала добавьте в метод GetEmployees() веб-службы следующую строку, чтобы сымитировать задержку на четыре секунды:

```
System.Threading.Thread.Sleep(4000);
```

Далее создайте веб-страницу, использующую веб-службу. Эта веб-страница должна определять индивидуальный метод, имитирующий длительную по времени задачу, опять-таки, с помощью метода ThreadSleep(). Вот какой код вам следует для этого добавить в код веб-страницы:

```
private void DoSomethingSlow()
{
 System.Threading.Thread.Sleep(3000);
}
```

На странице вы должны выполнить оба эти метода. Используя простой фрагмент временного кода, вы можете сравнить синхронный подход с асинхронным подходом. В зависимости от того, на какой кнопке щелкает пользователь, вы будете выполнять две операции синхронно (одну за другой) или асинхронно в одно и то же время.

Вот как должен выглядеть код, если вы хотите, чтобы эти две задачи выполнялись синхронно:

```
protected void cmdSynchronous_Click(object sender, System.EventArgs e)
{
 // Записываем время начала.
 DateTime startTime = DateTime.Now;
 // Извлекаем данные веб-службы.
 EmployeesService proxy = new EmployeesService();
```

```
  try
  {
     GridView1.DataSource = proxy.GetEmployees();
  }
  catch (Exception err)
  {
     lblInfo.Text = "Problem contacting web service.";
     return;
  }
  GridView1.DataBind();
  // Выполняем какие-то другие длительные по времени задачи.
  DoSomethingSlow();
  // Определяем общее количество времени, которое было затрачено.
  TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
  lblInfo.Text = "Synchronous operations took " + timeTaken.TotalSeconds + " seconds.";
}
```

Чтобы воспользоваться асинхронными делегатами, вы должны определить делегата, соответствующего сигнатуре метода, который требуется вызывать асинхронно. В данном случае это метод `GetEmployees()`:

```
public delegate DataSet GetEmployeesDelegate();
```

И вот как вы могли бы запустить сначала веб-службу так, чтобы операции перекрывали друг друга (т.е. выполнялись асинхронно):

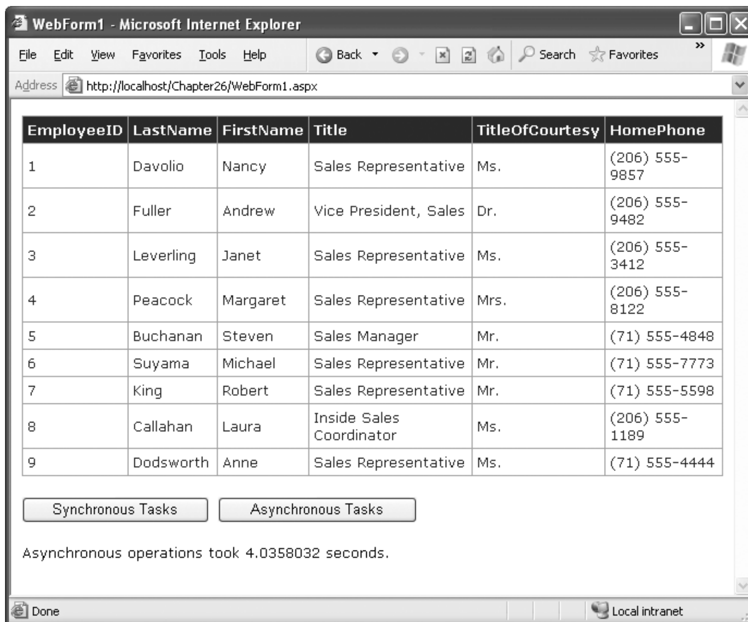
```
protected void cmdAsynchronous_Click(object sender, System.EventArgs e)
{
  // Записываем время начала.
  DateTime startTime = DateTime.Now;
  // Запускаем веб-службу в другом потоке.
  EmployeesService proxy = new EmployeesService();
  GetEmployeesDelegate async = new GetEmployeesDelegate(proxy.GetEmployees);
  IAsyncResult handle = async.BeginInvoke(null, null);
  // Выполняем какие-то другие длительные по времени задачи.
  DoSomethingSlow();

  // Извлекаем результат. Если он еще не готов, ожидаем.
  try
  {
    GridView1.DataSource = async.EndInvoke(handle);
  }
  catch (Exception err)
  {
    lblInfo.Text = "Problem contacting web service.";
    return;
  }
  GridView1.DataBind();
  // Определяем общее количество затраченного времени.
  TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
  lblInfo.Text = "Asynchronous operations took " + timeTaken.TotalSeconds +
    " seconds.";
}
```

Обратите внимание, что обработчик исключений охватывает метод `EndInvoke()`, но не метод `BeginInvoke()`. Причина проста: если во время обработки запроса возникнет какая-нибудь ошибка (либо из-за проблем в сети, либо из-за проблем на сервере), код не получит ее до тех пор, пока не будет вызван метод `EndInvoke()`.

Когда вы протестируете эти два примера, то увидите, что выполнение синхронного кода занимает 7–9 секунд, в то время как выполнение асинхронного кода — всего лишь 4–5 секунд. На рис. 37.1 показана результирующая веб-страница, в нижней части которой отображается общее затраченное на выполнение операций время.

Совет. Не забывайте о том, что выгода потоковой обработки зависит от типа операций. В данном примере все преимущества потоковой обработки проявляются потому, что операции не привязаны к центральному процессору (ЦП) — они просто дожидаются своей очереди. Это поведение похоже на то, с которым вы будете встречаться, получая доступ к внешним веб-службам или базам данных. Однако если вы попытаетесь применить потоковую обработку для одновременного выполнения двух операций, использующих ресурсы ЦП на одном и том же компьютере (и этот компьютер имеет только один ЦП), никаких преимуществ вы не получите, потому что на каждую из этих операций будет приходиться примерно по половине всех ресурсов ЦП, в результате чего время их выполнения увеличится вдвое.



EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	HomePhone
1	Davolio	Nancy	Sales Representative	Ms.	(206) 555-9857
2	Fuller	Andrew	Vice President, Sales	Dr.	(206) 555-9482
3	Leverling	Janet	Sales Representative	Ms.	(206) 555-3412
4	Peacock	Margaret	Sales Representative	Mrs.	(206) 555-8122
5	Buchanan	Steven	Sales Manager	Mr.	(71) 555-4848
6	Suyama	Michael	Sales Representative	Mr.	(71) 555-7773
7	King	Robert	Sales Representative	Mr.	(71) 555-5598
8	Callahan	Laura	Inside Sales Coordinator	Ms.	(206) 555-1189
9	Dodsworth	Anne	Sales Representative	Ms.	(71) 555-4444

Рис. 37.1. Тестирование асинхронного вызова метода

Параллельные асинхронные вызовы

Объект `IAsyncState` предлагает несколько других опций, которые могут оказаться очень полезными в случае, когда вызываются сразу несколько веб-методов одновременно. Главной из них является объект `IAsyncState.WaitHandle`, который возвращает объект `System.Threading.WaitHandle`. Используя этот объект, можно вызывать метод `WaitAll()`, чтобы подождать, пока будут выполнены все асинхронные операции. В следующем примере эта технология применяется для того, чтобы вызвать три раза подряд метод `GetEmployees()`:

```
protected void cmdMultiple_Click(object sender, System.EventArgs e)
{
 // Записываем время начала.
 DateTime startTime = DateTime.Now;
 EmployeesService proxy = new EmployeesService();
 GetEmployeesDelegate async = new GetEmployeesDelegate(proxy.GetEmployees);
 // Вызываем три метода асинхронно.
 IAsyncResult handle1 = async.BeginInvoke(null, null);
 IAsyncResult handle2 = async.BeginInvoke(null, null);
 IAsyncResult handle3 = async.BeginInvoke(null, null);
 // Создаем массив объектов WaitHandle.
 WaitHandle[] waitHandles = {handle1.AsyncWaitHandle,
 handle2.AsyncWaitHandle, handle3.AsyncWaitHandle};
 // Ожидаем, пока будут обработаны все запросы.
 WaitHandle.WaitAll(waitHandles);
 // Теперь можно извлечь результаты.
 DataSet ds1 = async.EndInvoke(handle1);
 DataSet ds2 = async.EndInvoke(handle2);
 DataSet ds3 = async.EndInvoke(handle3);
 // Объединяем все результаты в одну таблицу и отображаем ее.
 DataSet dsMerge = new DataSet();
 dsMerge.Merge(ds1);
 dsMerge.Merge(ds2);
 dsMerge.Merge(ds3);
 GridView1.DataSource = dsMerge;
 GridView1.DataBind();
 // Определяем общее количество затраченного времени.
 TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
 lblInfo.Text = "Calling three methods took " + timeTaken.TotalSeconds +
   " seconds.";
}
```

Вместо того чтобы использовать обработчик `WaitHandle`, можно было бы запустить три асинхронных вызова, вызвав метод `BeginInvoke()` три раза, и затем сразу же после этого вызвать три метода `EndInvoke()`. В этом случае код при необходимости ожидал бы. Однако использование обработчика `WaitHandle` делает код более понятным.

Также можно использовать одну из перегруженных версий метода `WaitAll()`, которая принимает значение тайм-аута. Если указанный период времени закончился, а выполнение вызовов еще не завершилось, будет выдано исключение. Однако, как правило, лучше вместо этого использовать свойство `Timeout`, которое завершит вызов, если ответ не был получен в течение указанного количества времени.

Еще один вариант — можно заставить обработчик `WaitHandle` блокировать поток до тех пор, пока не завершится обработка хотя бы какого-нибудь одного из вызовов метода, используя метод `WaitHandle.WaitAny()` с массивом объектов `WaitHandle`. Метод `WaitAny()` возвращается сразу же, как только завершится хотя бы какой-то один асинхронный вызов. Эта технология подходит для случаев, когда необходимо обработать пакет данных из разных источников, и порядок обработки значения не имеет. Она позволяет начинать обрабатывать результаты из одного метода еще до того, как завершится выполнение других методов. Однако она также усложняет код, потому что предполагает выполнение проверки свойства `IsCompleted` каждого объекта `IAsyncResult` и многократный вызов метода `WaitAny()` (как правило, в виде цикла) до тех пор, пока не завершится выполнение всех методов.

На заметку! Не забывайте о том, что .NET предоставляет даже еще больше опций потоковой обработки с классом `Thread`. Например, если необходимо вызвать ряд веб-служб в определенном порядке в виде долго работающей фоновой службы в Windows-приложении, лучше всего воспользоваться классом `Thread`. Более подробную информацию о многопоточной обработке можно найти в книгах, посвященных усовершенствованным технологиями программирования в Windows.

Реагирующие Windows-клиенты

В Windows-клиенте код потоковой обработки будет выглядеть несколько по-другому. Скорее всего, вы захотите сделать так, чтобы приложение продолжало нормально функционировать, пока выполняется операция, а когда обработка вызова завершится, отображаемые данные заменялись новыми (обновлялись).

Поддержка для такой модели встроена в прокси-класс. Чтобы понять, как она работает, не помешает взглянуть на код прокси-класса. Для каждого веб-метода в веб-службе код прокси-класса на самом деле включает два метода — синхронную версию, с которой вы встречались до сих пор, и асинхронную версию, в которой к имени метода прибавляется суффикс `Async`.

Ниже показан код для синхронной версии метода `GetEmployees()`. Атрибуты, необходимые для XML-сериализации, были опущены.

```
[SoapDocumentMethod(...)]
public DataSet GetEmployees()
{
 object[] results = this.Invoke("GetEmployees", new object[]{});
 return ((DataSet)(results[0]));
}
```

А вот как выглядит асинхронная версия этого же метода. Обратите внимание, что код на самом деле содержит две версии метода `GetEmployeesAsync()`. Единственное отличие между ними состоит в том, что одна из них принимает дополнительный параметр `userState`, которым может быть любой объект, используемый для идентификации вызова. Когда выполнение данного вызова завершится, этот объект будет возвращен в обратном вызове. Параметр `userState` особенно полезен в случаях, когда параллельно реализуются сразу несколько асинхронных веб-методов.

```
public void GetEmployeesAsync()
{
 this.GetEmployeesAsync(null);
}
public void GetEmployeesAsync(object userState)
{
 if ((this.GetEmployeesOperationCompleted == null))
 {
   this.GetEmployeesOperationCompleted = new
    System.Threading.SendOrPostCallback(this.OnGetEmployeesOperationCompleted);
 }
 this.InvokeAsync("DownloadFile", new object[]{},
 this.GetEmployeesOperationCompleted, userState);
}
```

Идея такова: вы вызываете метод `GetEmployeesAsync()`, чтобы запустить запрос. Из этого метода сразу происходит возврат, еще даже до того, как запрос будет отправлен через сеть, в результате чего прокси-класс помещается в свободный поток (точно так же, как было в примере с асинхронным делегатом) и находится там в режиме ожи-

дания до тех пор, пока не получит ответ. Как только ответ будет получен и пройдет десериализацию, .NET инициирует соответствующее событие, чтобы уведомить об этом ваше приложение. После этого вы можете извлечь результаты.

Чтобы эта система работала, прокси-класс также добавляет событие для каждого веб-метода. Это событие инициируется сразу же после того, как выполнение асинхронного метода завершится. Ниже показано событие завершения для метода `GetEmployees()`:

```
public event GetEmployeesCompletedEventHandler GetEmployeesCompleted;
```

Код прокси-класса ведет себя здесь достаточно интеллектуально — он облегчает разработчику жизнь, создавая для каждого веб-метода специальный объект `EventArgs`. Этот объект `EventArgs` представляет результат метода в виде свойства `Result`. Его класс объявляется с помощью ключевого слова `partial`, благодаря чему добавить в него код можно и в другом (не автоматически генерируемом) файле:

```
// Определяет сигнатуру события завершения.
public delegate void GetEmployeesCompletedEventHandler(object sender,
 GetEmployeesCompletedEventArgs e);
public partial class GetEmployeesCompletedEventArgs :
 System.ComponentModel.AsyncCompletedEventArgs
{
 private object[] results;
 // Конструктор является внутренним, исключаем возможность
 // создания экземпляра этого класса другими сборками.
 internal GetEmployeesEventArgs(object[] results,
   Exception exception, bool cancelled, object userState) :
   base(exception, cancelled, userState)
 {
   this.results = results;
 }

 public System.Data.DataSet Result
 {
   get
   {
        this.RaiseExceptionIfNecessary();
        return ((System.Data.DataSet)(this.results[0]));
   }
 }
}
```

Обратите внимание, что в случае, если бы на сервере произошла ошибка, она не была бы выдана до тех пор, пока вы не попытались бы извлечь свойство `Result`, потому что именно на этом этапе объект `SoapException` включается в объект `TargetInvocation`.

На заметку! В специальном объекте `EventArgs` может встречаться не только свойство `Result`. Если веб-метод принимает ссылочные или выходные параметры, они тоже будут добавлены в этот объект. Таким образом, вы сможете получить измененные значения всех ссылочных или выходных параметров, когда завершится выполнение вызова.

Помимо свойства `Result`, доступно еще несколько свойств, которые объявляются в базовом классе `AsyncCompletedEventArgs`, а именно: `Cancelled` (возвращает значение `true`, если операция была отменена до того, как она завершилась), `Error` (возвращает объект исключения, если во время обработки запроса произошла необработанная ошибка) и `UserState` (возвращает объект состояния, который предоставлялся при вызове метода).

Чтобы посмотреть, как работают эти свойства, давайте изменим код Windows-клиента, который мы создавали в главе 35, так, чтобы он использовал асинхронный вызов. Первое, что потребуется сделать — это создать в классе формы обработчик для события завершения:

```
private void GetEmployeesCompleted(object sender, GetEmployeesCompletedEventArgs e)
{ ... }
private void cmdGetEmployees_Click(object sender, System.EventArgs e)
{
 // Делаем кнопку недоступной, так что за раз сможет
 // обрабатываться только один асинхронный вызов.
 cmdGetEmployees.Enabled = false;

 // Создаем прокси.
 EmployeesService proxy = new EmployeesService();

 // Создаем делегата обратного вызова.
 proxy.GetEmployeesCompleted += new
 GetEmployeesCompletedEventHandler(this.GetEmployeesCompleted);

 // Вызываем веб-службу асинхронно.
 proxy.GetEmployeesAsync();
}
```

Сразу же после завершения операции прокси-класс инициирует это событие. Во время обработки этого события его результат можно привязать к сетке (т.е. к элементу управления DataGridView). Волноваться о маршалинге вызова в потоке пользовательского интерфейса не нужно, потому что прокси-класс делает это автоматически перед тем, как инициировать событие, что чрезвычайно удобно.

```
private void GetEmployeesCompleted(object sender, GetEmployeesCompletedEventArgs e)
{
 // Извлекаем результат.
 try
 {
   dataGridView1.DataSource = e.Result;
 }
 catch (System.Reflection.TargetInvocationException err)
 {
   MessageBox.Show("An error occurred.");
 }
}
```

Если вы запустите показанное в этом примере приложение и щелкните на кнопке Get Employees (Ввести сотрудников), кнопка станет недоступной, но приложение будет по-прежнему реагировать на ваши действия. Вы сможете изменять размер окна, щелкать на других кнопках, чтобы выполнить еще какой-нибудь код, и т.д. А когда результаты будут получены, последует обратный вызов и содержимое элемента DataGridView будет автоматически обновлено.

Чтобы реализовать модель отмены в этом приложении, сначала необходимо объявить прокси-класс на уровне формы так, чтобы он был доступен для всех обработчиков событий:

```
private EmployeesService proxy = new EmployeesService();
```

Далее потребуется добавить обработчик события после загрузки формы:

```
private void Form1_Load(object sender, EventArgs e)
{
  proxy.GetEmployeesCompleted += new
    GetEmployeesCompletedEventHandler(this.GetEmployeesCompleted);
}
```

В этом примере то, какой объект состояния используется, роли не играет, потому что за один раз будет выполняться только одна операция. В случае если выполняться будут сразу несколько операций, имеет смысл сгенерировать новый GUID-идентификатор (Globally Unique Identifier — глобально уникальный идентификатор) для отслеживания каждой из них.

Чтобы сделать это, сначала объявите GUID-идентификатор в классе формы:

```
private Guid requestID;
```

Затем сгенерируйте его и передайте вызову асинхронного веб-метода:

```
requestID = Guid.NewGuid();
proxy.GetEmployeesAsync(requestID);
```

Теперь все, что вам нужно — это указать этот GUID-идентификатор при вызове метода `CancelAsync()`. Вот как должен выглядеть код для кнопки отмены:

```
private void cmdCancel_Click(object sender, EventArgs e)
{
  proxy.CancelAsync(requestID);
  MessageBox.Show("Operation cancelled.");
}
```

Здесь необходимо обратить внимание на одну важную деталь. При вызове метода `CancelAsync()` инициируется событие `Completed`. В этом есть смысл, потому что завершилась (хотя и по причине программного вмешательства) длительная операция, и, возможно, пользовательский интерфейс требуется обновить. Однако, очевидно, что получить доступ к результату метода в событии завершения не получится, потому что процесс выполнения кода был прерван. Чтобы избежать такой ошибки, потребуется явно выполнить проверку на то, не случались ли во время выполнения кода отмены, как показано ниже:

```
private void GetEmployeesCompleted(object sender, GetEmployeesCompletedEventArgs e)
{
 if (!e.Cancelled)
 {
   try
   {
       dataGridView1.DataSource = e.Result;
   }
   catch (System.Reflection.TargetInvocationException err)
   {
       MessageBox.Show("An error occurred.");
   }
 }
}
```

Асинхронные службы

Мы уже рассмотрели несколько примеров, позволяющих клиентам вызывать веб-службы асинхронно. Но во всех этих примерах веб-метод все равно выполнялся синхронно от начала до конца. А что если вы хотите получить другое поведение, позволяющее клиенту инициировать длительный процесс и затем подключаться к веб-службе уже позже для сборки результатов?

К сожалению, .NET не поддерживает такую модель напрямую. Часть проблемы состоит в том, что все соединения с веб-службой должны инициироваться клиентом. На текущий момент веб-сервер не имеет возможности инициировать обратный вызов клиента для того, чтобы уведомить его о том, что задача выполнена. И даже если со временем стандарты разовьются настолько, что смогут поддерживать такую возможность,

она вряд ли будет широко применяться, что связано с архитектурными особенностями Web. Многие клиенты подключаются через прокси-серверы или брандмауэры, которые не разрешают входящие соединения или скрывают информацию о месторасположении, такую как IP-адрес. Поэтому каждое соединение должно инициироваться самим клиентом.

Конечно, определенные новаторские решения могут обеспечить некоторые из необходимых функциональных возможностей. Например, вы можете сделать так, чтобы клиент подключался к серверу периодически (через определенные промежутки времени) и проверял, выполнена ли задача. Такая схема подходит для случаев, когда веб-серверу приходится выполнять чрезвычайно длительные по времени задачи вроде визуализации сложной графики. Однако здесь по-прежнему не хватает одного ингредиента. В таких ситуациях клиент должен иметь возможность запускать веб-метод, не дожидаясь того, когда завершится выполнение этого веб-метода. ASP.NET делает такое поведение возможным, позволяя создавать *однонаправленные методы*.

Когда используются однонаправленные методы (также называемые методами типа “инициировать и забыть”), клиент отправляет сообщение запроса, но веб-служба никогда не отвечает на него. Это означает, что такой веб-метод завершается сразу же и закрывает соединение. Клиенту вообще не приходится тратить время на ожидание. Однако однонаправленные методы имеют несколько недостатков. В частности, такой веб-метод не может предоставлять возвращаемого значения или использовать ссылочный или выходной параметр. Также, если такой веб-метод выдаст необработанное исключение, оно не будет передано обратно клиенту.

Чтобы создать однонаправленный метод в основанной на XML веб-службе, просто примените атрибут `SoapDocumentMethod` к нужному веб-методу и установите для свойства `OneWay` значение `true`, как показано ниже:

```
[SoapDocumentMethod(OneWay = true)]
[WebMethod()]
public DataSet GetEmployees()
{ ... }
```

Клиенту не нужно предпринимать никаких специальных шагов, чтобы вызывать однонаправленный метод асинхронно. Такой метод всегда возвращается незамедлительно.

Конечно, вы можете и не захотеть пользоваться однонаправленными методами. Самой главной причиной, по которой вы можете принять такое решение, является то, что в таком случае у вас не получится возвратить какую-либо информацию. Например, на стороне сервера чаще всего применяется такая модель: когда сервер получает запрос от клиента, он возвращает ему какой-нибудь уникальный, генерируемый автоматически мандат. Клиент затем передает этот мандат другим методам, чтобы проверить состояние или извлечь результаты. В случае с однонаправленным методом возможности вернуть мандат или уведомить клиента о произошедшей ошибке не существует. Еще одна проблема состоит в том, что однонаправленные методы все равно используют рабочие потоки ASP.NET. Если задача выполняется очень долго и текущих задач много, другие клиенты могут лишиться возможности отправлять новые запросы, а в этом нет ничего хорошего.

Единственный практичный способ решить проблему с длительными по времени асинхронными задачами на веб-сайте с большим трафиком — это объединить веб-службы с еще одной .NET-технологией, которая называется удаленным доступом (Remoting). Например, вы могли бы создать обычный синхронный веб-метод, который бы возвращал мандат и затем вызывал какой-нибудь метод в компоненте на стороне сервера, использующем технологию удаленного доступа, после чего этот компонент начинал бы асинхронно выполнять обработку. Эта технология использования веб-службы

в качестве интерфейса для полнофункционального, работающего в непрерывном режиме серверного компонента представляет собой один из наиболее сложных подходов распределенного проектирования. Более подробную информацию об этой технологии можно найти в книгах, посвященных распределенному программированию. Конечно, если в подобной гибкости нет необходимости, эту технологию лучше вообще не применять, потому что она значительно увеличивает степень сложности проекта и прибавляет немало работы.

Обеспечение безопасности веб-служб

В идеальном мире к веб-службе можно было бы относиться как к библиотеке классов, содержащей определенные функциональные возможности, и не волноваться о написании кода для аутентификации пользователя или, другими словами, о логике поддержки безопасности. Однако чтобы создать веб-службу, основанную на системе регистрации или микро-платежей, вы должны определить, кто ее использует. И даже если вы не продаете свою логику другим веб-разработчиками, вам все равно может понадобиться использовать аутентификацию, чтобы защитить уязвимые данные и заблокировать доступ злоумышленникам, особенно если ваша веб-служба работает через Интернет.

Вы можете применять для защиты своих веб-служб ряд тех же технологий, которые применяете для защиты веб-страниц. Например, вы можете воспользоваться IIS, чтобы принудительно активизировать SSL-протокол (для этого просто укажите своим клиентам, что они должны использовать URL-адрес веб-службы, начинающийся с `https://`). Вы также можете воспользоваться IIS, чтобы применить систему аутентификации Windows, хотя в этом случае вам также придется выполнить несколько дополнительных шагов, которые описываются в следующем разделе. И, наконец, вы можете создать свою собственную специальную систему аутентификации с помощью SOAP-заголовков.

Аутентификация Windows

Система аутентификации Windows работает с веб-службой практически так же, как она работает с веб-страницей. Единственное отличие состоит в том, что веб-служба выполняется другим приложением, а не самим браузером. Из-за этого встроенная возможность отображать пользователю приглашение на ввод имени пользователя и пароля отсутствует. Поэтому эту информацию должно предоставлять приложение, которое использует данную веб-службу. Это приложение может считывать такую информацию из конфигурационного файла или базы данных или же отображать пользователю приглашение ввести эти данные перед тем, как устанавливать соединение с веб-службой.

Для примера давайте рассмотрим следующую веб-службу, которая предоставляет один единственный метод — `TestAuthenticated()`. Этот метод проверяет, прошел ли пользователь аутентификацию. Если пользователь прошел аутентификацию, он возвращает имя пользователя (в виде строки в формате `Имя_домена\Имя_пользователя` или `Имя_компьютера\Имя_пользователя`).

```
public class SecureService : System.Web.Services.WebService
{
 [WebMethod()]
 public string TestAuthenticated()
 {
   if (!User.Identity.IsAuthenticated)
   {
       return "Not authenticated.";
   }
   else
```

```
        {
            return "Authenticated as: " + User.Identity.Name;
        }
    }
}
```

веб-служба также может проверять принадлежность к роли, хотя именно эта веб-служба этого не делает.

Чтобы предоставить данные удостоверения этой службе, клиент должен изменить значение свойства NetworkCredential прокси-класса. Здесь доступны два варианта.

- Можно создать новый объект NetworkCredential и присоединить его к свойству NetworkCredential объекта прокси-класса. При создании объекта NetworkCredential необходимо будет указать имя пользователя и пароль, которые должны применяться. Такой подход работает с системой аутентификации Windows любого вида.
- Если веб-служба использует систему встроенной аутентификации Windows, можно автоматически предоставить данные удостоверения пользователя, воспользовавшись свойством DefaultCredentials класса CredentialCache и применив его к свойству NetworkCredential объекта прокси-класса.

И класс CredentialCache, и класс NetworkCredential находятся в пространстве имен System.Net. Поэтому, прежде чем продолжить, следует импортировать это пространство имен:

```
using System.Net;
```

Следующий код иллюстрирует веб-страницу с двумя текстовыми полями и двумя кнопками (рис. 37.2). Одна кнопка выполняет несанкционированный вызов, а вторая отправляет имя пользователя и пароль, которые были введены в текстовых полях.

Несанкционированный вызов не будет принят, если возможность доступа анонимных пользователей была отключена. В противном случае несанкционированный вызов будет принят, но метод TestAuthenticated() вернет строку, информирующую о том, что аутентификация не выполнялась. Санкционированный вызов будет приниматься всегда при условии предоставления удостоверения, которое соответствует действительному пользователю на веб-сервере.

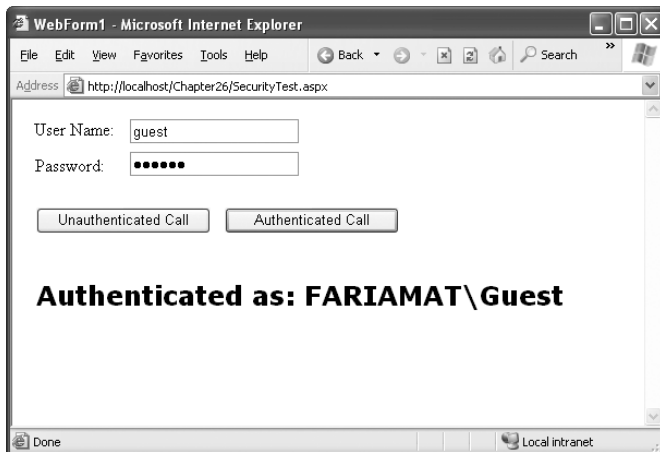


Рис. 37.2. Успешная аутентификация в веб-службе

Ниже показан полный код этой веб-страницы:

```
public partial class WindowsAuthenticationSecurityTest : Page
{
 protected void cmdUnauthenticated_Click(Object sender, EventArgs e)
 {
   SecureService proxy = new SecureService();
   try
   {
       lblInfo.Text = proxy.TestAuthenticated();
   }
   catch (Exception err)
   {
       lblInfo.Text = err.Message;
   }
 }
 protected void cmdAuthenticated_Click(Object sender, EventArgs e)
 {
   SecureService proxy = new SecureService();
   // Предоставляем какое-нибудь удостоверение пользователя для веб-службы.
   NetworkCredential credentials = new NetworkCredential(
       txtUserName.Text, txtPassword.Text);
   proxy.Credentials = credentials;
   lblInfo.Text = proxy.TestAuthenticated();
 }
}
```

Чтобы испробовать этот код, добавьте в файл web.config следующий дескриптор <location>, который ограничивает доступ к веб-службе SecureService.asmx:

```
<configuration>
 <system.web>
  <authorization>
   <allow users="*" />
  </authorization>
 ...
 </system.web>
 <location path="SecureService.asmx">
  <system.web>
   <authorization>
    <deny users="?" />
   </authorization>
  </system.web>
 </location>
</configuration>
```

При желании использовать данные удостоверения учетной записи текущего пользователя со встроенной аутентификацией Windows, лучше воспользуйтесь следующим кодом:

```
SecuredService proxy = new SecuredService();
proxy.Credentials = CredentialCache.DefaultCredentials;
lblInfo.Text = proxy.TestAuthenticated();
```

В этом примере (как и на всех веб-страницах) текущей учетной записью пользователя будет учетная запись, которую использует ASP.NET, а не учетная запись удаленного пользователя, который запрашивает данную веб-страницу. В случае применения этой технологии в Windows-приложении предоставляться будут данные учетной записи того пользователя, который работает с данным приложением.

Специальная аутентификация на основе мандатов

Аутентификация Windows прекрасно подходит для веб-служб, когда речь идет о небольшом наборе пользователей, у которых имеются учетные записи в Windows. Однако в случае, когда речь идет крупномасштабных общедоступных веб-службах, это далеко не самый лучший вариант. На веб-страницах ASP.NET в такой ситуации все пробелы заполняются посредством аутентификации с помощью форм. Тем не менее, для веб-служб аутентификация с помощью форм не подходит, потому что они не имеют возможности направлять пользователя на веб-страницу. На самом деле к ним даже может оказаться невозможным получать доступ через браузер: они могут быть доступны через какое-то Windows-приложение или даже через автоматизированную Windows-службу. Также аутентификация с помощью форм предполагает пересылку cookie-наборов, которые являются просто ненужным ограничением для веб-служб, поскольку те могут применять протоколы, не поддерживающие cookie-наборы, или работать с клиентами, которые их не ожидают.

Наиболее распространенное решение в такой ситуации — создать свою собственную систему аутентификации. В такой модели пользователь будет вызывать в веб-службе определенный веб-метод, чтобы зарегистрироваться, что предполагает предоставление данных удостоверения (таких как имя пользователя и пароль). Метод регистрации зарегистрирует сеанс пользователя и создаст для него новый уникальный мандат. С этого момента пользователь сможет повторно подключаться к данной веб-службе, просто передавая всем остальным методам этот мандат.

Хорошо спроектированная система мандатов обладает рядом преимуществ. Как и система аутентификации с помощью форм, она обеспечивает высокую степень гибкости. Она также оптимизирует производительность и гарантирует масштабируемость, потому что вы можете поместить мандат в кэш в памяти, в результате чего при последующих запросах вы сможете просто проверять мандат вместо того, чтобы снова аутентифицировать пользователя с помощью базы данных. И, наконец, она дает возможность воспользоваться преимуществами SOAP-заголовков, которые делают процесс управления мандатами и авторизации прозрачным для клиента.

С выходом версии ASP.NET 2.0 появилась возможность упростить специальную систему аутентификации. Хотя за написание кода для передачи данных удостоверения пользователя и отслеживания того, кто подключился, за счет создания и проверки мандатов, по-прежнему отвечает разработчик, для обработки аутентификации и авторизации теперь могут применяться функции диспетчера ролей и ролевой принадлежности, описанные в главах 21 и 23. В следующих разделах будет показано, как создать специальную основанную на мандатах систему аутентификации, использующую эти функции подобным образом.

Отслеживание подлинности пользователя

Чтобы использовать специальную систему безопасности, первым делом следует определиться с тем, какая информация о пользователе должна помещаться в кэш в памяти. Далее потребуется создать специальный класс, представляющий эту информацию. Этот класс может включать информацию о пользователе (имя, адрес электронной почты и т.д.) и его правах. Также он обязательно должен включать мандат.

Ниже показан пример базового кода для такого класса, который сохраняет имя пользователя и мандат:

```
public class TicketIdentity
{
 private string userName;
```

```
  public string UserName
  {
     get { return userName; }
  }
  private string ticket;
  public string Ticket
  {
     get { return ticket; }
  }
  public TicketIdentity(string userName)
  {
     this.userName = userName;
     // Создаем глобально уникальный идентификатор мандата.
     Ticket = Guid.NewGuid().ToString();
  }
}
```

На заметку! Вы наверняка заметили, что этот класс данных пользователя не реализует интерфейс `IIdentity`. Причина проста: такой подход не позволяет подключаться к модели безопасности .NET так, как он позволяет это делать, когда речь идет о создании специальной системы аутентификации для веб-страниц. По сути, проблема состоит в том, что аутентификация должна выполняться *после* того, как объект `User` уже был создан. И обойти эту проблему с помощью класса `global.asax` нельзя, потому что у обработчиков событий приложения не будет доступа к параметрам веб-метода и SOAP-заголовку, который необходим для выполнения аутентификации и авторизации.

Создав класс данных пользователя, следует создать SOAP-заголовок. Этот SOAP-заголовок будет отслеживать только мандат пользователя. Поскольку мандат представляет собой генерируемый путем случайного подбора символов GUID-идентификатор, вероятность того, что злоумышленнику удастся “угадать” значение мандата другого пользователя, очень низка.

```
public class TicketHeader : SoapHeader
{
  public string Ticket;
  public TicketHeader(string ticket)
  {
     Ticket = ticket;
  }
  public TicketHeader()
  {}
}
```

Теперь осталось только добавить переменную экземпляра для `TicketHeader` в веб-службу:

```
public class SoapSecurityService : WebService
{
  public TicketHeader Ticket;
  ...
}
```

Аутентификация пользователя

Следующее, что нужно сделать — это создать специальный веб-метод, регистрирующий пользователя. Пользователь должен предоставить этому методу данные удостоверения (такие как имя пользователя и пароль). После чего это метод извлечет информацию о данном пользователе, создаст объект `TicketIdentity` и сгенерирует мандат.

В рассматриваемом примере веб-метод Login() проверяет данные удостоверения пользователя, используя статический метод Membership.ValidateUser(). Далее на основании полученной от пользователя информации создается новый мандат, который затем сохраняется в коллекции Application в специально отведенном для этого пользователя сегменте. В это же время на основе мандата генерируется новый SOAP-заголовок, позволяющий данному пользователю получать доступ к другим методам.

Ниже показан полный код метода Login().

```
[WebMethod()]
[SoapHeader("Ticket", Direction = SoapHeaderDirection.Out)]
public void Login(string userName, string password)
{
 if (Membership.ValidateUser(username, password))
 {
   // Создаем новый мандат.
   TicketIdentity ticket = new TicketIdentity(username);
   // Добавляем этот мандат в состояние Application.
   Application[ticket.Ticket] = ticket;
   // Создаем SOAP-заголовок.
   Ticket = new TicketHeader(ticket.Ticket);
 }
 else
 {
   throw new SecurityException("Invalid credentials.");
 }
}
```

В этом примере следует обратить внимание на то, что объект TicketIdentity сохраняется в коллекции Application, которая является глобальной для всех пользователей. Однако волноваться о том, что это может привести к перезаписи одного мандата пользователя другим, не стоит, потому что мандаты индексируются с помощью GUID-идентификаторов. Каждый пользователь имеет свой собственный GUID-идентификатор мандата, а значит и отдельный сегмент в коллекции Application.

Коллекция Application обладает определенными ограничениями, к которым относится отсутствие поддержки для работы с группой веб-серверов и плохая масштабируемость при обслуживании большого количества пользователей. Также в случае перезапуска веб-приложения все мандаты будут утрачены. Чтобы улучшить это решение, можно сохранить информацию в двух местах: в объекте Cache и в конечной базе данных. В таком случае код сможет сначала проверять объект Cache и, если ему там удастся обнаружить подходящий объект TicketIdentity, он не будет отправлять запрос в базу данных. Но даже если подходящего объекта TicketIdentity там нет, необходимая информация все-таки сможет быть извлечена из базы данных. Важно понимать, что это улучшение все равно предполагает использование того же SOAP-заголовка с мандатом и того же объекта TicketIdentity. Единственное отличие связано с тем, как объект TicketIdentity сохраняется и как он извлекается между запросами.

Авторизация пользователя

Создав метод Login(), имеет смысл написать индивидуальный метод, который мог бы вызываться для проверки присутствия пользователя. Затем этот метод можно вызывать из других веб-методов в веб-службе.

Показанный ниже метод AuthorizeUser() выполняет проверку на наличие подходящего мандата и возвращает объект TicketIdentity, если ему удалось его обнаружить. Если нет, генерируется исключение, которое будет возвращено пользователю.

```
private TicketIdentity AuthorizeUser(string ticket)
{
 TicketIdentity ticketIdentity = (TicketIdentity)Application[ticket];
 if (ticket != null)
 {
   return ticketIdentity;
 }
 else
 {
   throw new SecurityException("Invalid ticket.");
 }
}
```

Более того, эта перегруженная версия метода `AuthorizeUser()` проверяет, имеется ли у пользователя мандат и является ли он членом какой-нибудь определенной роли. Проверкой ролей занимается ASP.NET-поставщик управления ролями.

```
private TicketIdentity AuthorizeUser(string ticket, string role)
{
 TicketIdentity ticketIdentity = AuthorizeUser(ticket);
 if (Roles.IsUserInRole(ticketIdentity.UserName, role))
 {
   throw new SecurityException("Insufficient permissions.");
 }
 else
 {
   return ticketIdentity;
 }
}
```

Используя эти два вспомогательных метода, вы можете создать другие методы в веб-службе, проверяющие права пользователя, прежде чем выполнять какие-либо задачи или возвращать конфиденциальную информацию.

Тестирование SOAP-системы аутентификации

Теперь осталось только создать веб-метод тестирования, использующий метод `AuthorizeUse()` для проверки наличия у пользователя необходимых прав. Ниже показан пример такого метода, который проверяет, является ли клиент администратором, прежде чем позволит ему извлечь объект `DataSet` со списком сотрудников:

```
[WebMethod()]
[SoapHeader("Ticket", Direction = SoapHeaderDirection.In)]
public DataSet GetEmployees()
{
 AuthorizeUser(Ticket.Ticket, "Administrator");
 ...
}
```

Чтобы упростить настройку процесса тестирования, в приведенный в этой главе в качестве примера код был включен веб-метод `CreateTestUser()`, который генерирует определенного пользователя и выполняет пользовательскую часть роли `Administrators`:

```
[WebMethod()]
public void CreateTestUser(string username, string password)
{
 // Удаляем пользователя, если пользователь уже существует.
 if (Membership.GetUser(username) != null)
 {
   Membership.DeleteUser(username);
 }
```

```
  // Создаем пользователя.
  Membership.CreateUser(username, password);
  // Делаем этого пользователя администратором
  // и создаем роль, если ее не существует.
  string role = "Administrator";
  if (!Roles.RoleExists(role))
  {
    Roles.CreateRole(role);
  }
  Roles.AddUserToRole(username, role);
}
```

Теперь осталось просто создать клиента, тестирующего этот код. В нашем случае это будет веб-страница, включающая два текстовых поля для ввода имени пользователя и пароля (рис. 37.3). Введенная в этих полях информация передается методу `Login()`, после чего вызывается метод `GetEmployees()`, который уже непосредственно извлекает данные. Этот метод выполняется успешно для пользователя с ролью `Administrator`, а для всех остальных пользователей возвращает ошибку.

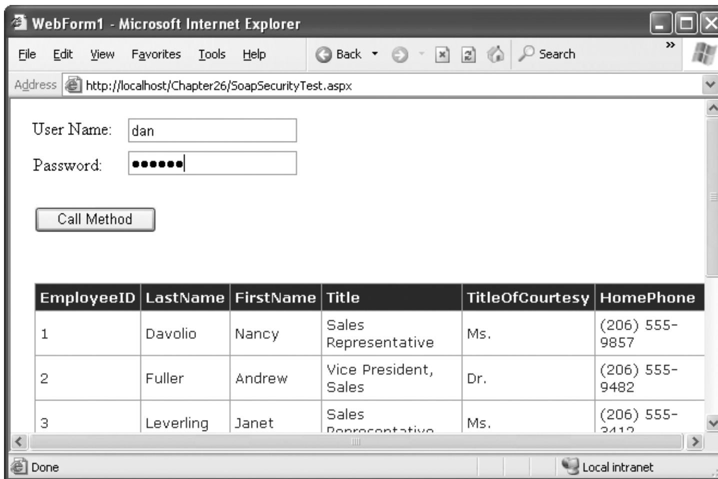


Рис. 37.3. Тестирование веб-метода, который использует авторизацию на основе мандатов

Вот как выглядит код этой веб-страницы:

```
protected void cmdCall_Click(object sender, System.EventArgs e)
{
  SoapSecurityService proxy = new SoapSecurityService();
  try
  {
    proxy.Login(txtUserName.Text, txtPassword.Text);
    GridView1.DataSource = proxy.GetEmployees();
    GridView1.DataBind();
  }
  catch (Exception err)
  {
    lblInfo.Text = err.Message;
  }
}
```

Вся прелесть в том, что клиенту даже не нужно знать об управление мандатами. А все потому, что метод `Login()` генерирует мандат, а обслуживает его прокси-класс. Если клиент использует тот же самый экземпляр прокси-класса, автоматически предоставляется то же самое значение мандата и пользователь проходит аутентификацию.

Вы можете сделать еще много чего, чтобы улучшить эту систему аутентификации. Например, вы можете сделать так, чтобы вместе с объектом `TicketIdentity` записывались дополнительные детали, такие как время создания и последнего использования мандата и сетевой адрес пользователя, который владеет данным мандатом, а значит и предусмотреть дополнительные проверки в методе `AuthorizeUser()`. Еще вы можете сделать так, чтобы у мандатов после длительного неиспользования истекал срок действия или так, чтобы мандат отклонялся, если у клиента изменился IP-адрес.

SOAP-расширения

веб-службы ASP.NET предоставляют высокоуровневый доступ к протоколу SOAP. Как вы видели, вам не нужно много знать о SOAP для того, чтобы создавать и вызывать веб-службы. Однако если вы хорошо разбираетесь в стандарте SOAP и не прочь “повозиться” с низкоуровневым доступом к SOAP-сообщениям, .NET предоставляет вам им такую возможность тоже. В этом разделе вы научитесь перехватывать SOAP-сообщения и выполнять с ними различные манипуляции.

На заметку! Даже если вы не хотите выполнять никаких манипуляций с SOAP-сообщениями, вам все равно не помешает узнать побольше о SOAP-расширениях, потому что они являются частью инфраструктуры, которая поддерживает набор инструментальных средств WSE, описываемый чуть позже в этой главе, в разделе “Набор инструментальных средств Web Services Enhancements”.

SOAP-расширения представляют собой механизм расширяемости. Они позволяют сторонним разработчикам создавать компоненты, которые вставляются в модель веб-службы и предоставляют какие-нибудь другие службы. Например, вы могли бы создать SOAP-расширение для выборочного шифрования или сжатия частей SOAP-сообщения перед его отправкой с клиента. Конечно, в этом случае вам также потребовалось бы запустить соответствующее SOAP-расширение на сервере для дешифровки или восстановления сообщения после его получения, но перед десериализацией.

Чтобы создать SOAP-расширение, следует подготовить класс, порожденный от класса `System.Web.Services.Protocols.SoapExtension`. Класс `SoapExtension` включает метод `ProcessMessage()`, который инициируется автоматически по мере того, как SOAP-сообщение проходит ряд этапов. Например, в случае запуска SOAP-сообщения на веб-сервере это будут четыре таких этапа.

- Этап **`SoapMessageStage.BeforeDeserialize`** происходит сразу же после того, как веб-сервер получает SOAP-сообщение с запросом.
- Этап **`MessageStage.AfterDeserialize`** происходит после того, как необработанное SOAP-сообщение преобразуется в .NET-типы данных, но как раз перед тем, как начинает выполняться код веб-метода.
- Этап **`SoapMessageState.BeforeSerialize`** происходит после запуска кода веб-метода, но перед преобразованием возвращаемого значения в SOAP-сообщение.
- Этап **`SoapMessageStage.AfterSerialize`** происходит после сериализации возвращаемых данных в SOAP-сообщение ответа, но перед его отправкой клиентскому приложению.

На каждом этапе могут извлекаться различные фрагменты информации о SOAP-сообщении. Например, на этапе `BeforeDeserialize` и этапе `AfterSerialize` можно извлечь весь текст SOAP-сообщения.

SOAP-расширение также может быть реализовано и на стороне клиента. В этом случае SOAP-сообщение будет проходить те же четыре этапа, только вот получать его, выполнять его десериализацию и обрабатывать его будет прокси-класс, а не веб-служба. Весь этот процесс показан на рис. 37.4.

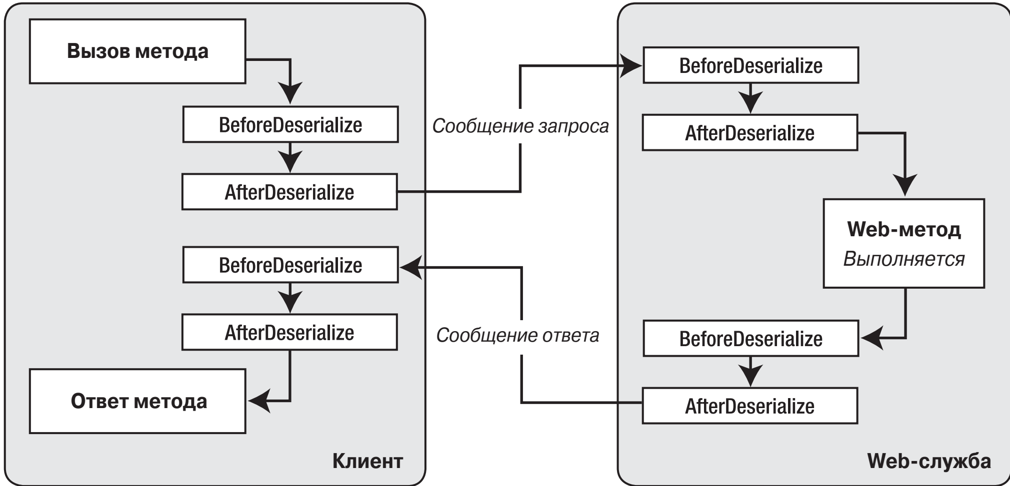


Рис. 37.4. Обработка SOAP-сообщения в клиентском приложении и на сервере

Создать SOAP-расширения производственного качества довольно непросто. Здесь возникает ряд серьезных препятствий.

- SOAP-расширения, как правило, должны выполняться как на стороне сервера, так и на стороне клиента. Это означает, что вам придется волноваться о распространении еще одного компонента и возможности управления им, что может быть чрезвычайно сложно сделать в большой распределенной системе.
- SOAP-расширения снижают степень совместимости веб-служб. Сторонние клиенты могут не знать, что им необходимо установить и запустить SOAP-расширение, чтобы использовать данную службу, поскольку такая информация не включается в WSDL-документ. Если клиенты работают с платформами, отличными от .NET, они не смогут использовать созданный вами класс `SoapExtension`, поэтому не исключено, что вам придется искать еще какой-нибудь способ для расширения конвейера обработки SOAP-сообщений, что, в зависимости от среды, может оказаться либо сложной, либо вообще не решаемой задачей.
- Интернет богат примерами SOAP-расширений, которые применяют шифрование некачественным и небезопасным образом. Основными недостатками этих расширений является небезопасное управление ключами, отсутствие возможности выполнять обмен ключами и низкая производительность из-за того, что вместо симметричного шифрования применяется ассиметричное шифрование. Более удачный вариант — использовать протокол SSL через IIS, который способен обрабатывать любые исключительные ситуации.

В большинстве случаев создание SOAP-расширений — это задача, выполнение которой лучше доверить разработчикам в Microsoft, занимающимся вопросами и про-

блемами создания “рабочего” кода уровня предприятия. На самом деле Microsoft реализует несколько более новых, менее развитых стандартов веб-служб с помощью SOAP-расширений в компоненте WSE, который можно загрузить бесплатно и о котором более подробно будет рассказывать чуть позже в этой главе, в разделе “Набор инструментальных средств Web Services Enhancements”.

На заметку! SOAP-расширения работают только через протокол SOAP. При тестировании веб-метода с помощью отображаемой в браузере тестовой страницы они вызываться не будут. Также SOAP-сообщения не будут работать, если для атрибута `BufferResponse` атрибута `WebMethod` установить значение `false`. В таком случае SOAP-расширение лишается возможности работать с SOAP-данными, потому что ASP.NET начинает их отсылать незамедлительно, еще до того, как они будут сгенерированы полностью.

Создание SOAP-расширения

В следующем примере будет показано SOAP-расширение, которое записывает SOAP-сообщения в журнал событий Windows. Проблемы, возникающие во время обработки SOAP-сообщений, будут заноситься в журнал событий в виде ошибок.

Совет. Некоторые разработчики применяют SOAP-расширения для трассировки. Однако, как уже рассказывалось в главе 36, это делать вовсе не обязательно: утилита трассировки, поставляемая вместе с набором инструментальных средств Microsoft SOAP Toolkit, подходит для этого гораздо больше. Одно из преимуществ SOAP-регистратора, демонстрируемого в этом примере, состоит в том, что вы могли бы его использовать, чтобы перехватывать и надолго сохранять SOAP-сообщения, даже если утилита трассировки не запущена. Также для того, чтобы сообщения пересылались через утилиту трассировки, изменять порт в коде клиента не нужно.

Все SOAP-расширения состоят из двух ингредиентов: специального класса, унаследованного от `System.Web.Services.Protocols.SoapExtension`, и специального атрибута, который вы применяете к веб-методу, чтобы указать, что использоваться должно ваше SOAP-расширение. Специальный атрибут является более простым из этих двух ингредиентов.

Атрибут `SoapExtension`

Атрибут `SoapExtension` позволяет связывать специальные SOAP-расширения с методами в веб-классе. При создании атрибут `SoapExtension` наследуется от класса `System.Web.Services.Protocols.SoapExtensionAttribute`, как показано ниже:

```
[AttributeUsage(AttributeTargets.Method)]
public class SoapLogAttribute:
System.Web.Services.Protocols.SoapExtensionAttribute
{ ... }
```

Обратите внимание, что класс атрибута включает еще один атрибут — `AttributeUsage`, который указывает, где вы можете использовать свой специальный атрибут. Атрибуты SOAP-расширений (`SoapExtension`) всегда применяются к индивидуальным объявлениям методов, во многом подобно атрибутам `SoapHeader` и `WebMethod`. Поэтому вы должны использовать атрибут `AttributeTargetsMethod`, дабы исключить возможность их применения пользователем к какой-нибудь другой конструкции кода (такой как объявление класса, например). Используйте атрибут `AttributeUsage` всякий раз, когда вам необходимо создать специальный атрибут, причем не только в сценариях с веб-службами.

Каждый атрибут SoapExtension должен перекрывать два абстрактных свойства: Priority и ExtensionType. Свойство Priority определяет порядок, в котором должны применяться SOAP-расширения, если было сконфигурировано несколько расширений. Однако в более простых расширениях, таких как расширение, демонстрируемое в данном примере, в нем нет необходимости. Свойство ExtensionType возвращает объект Type, который представляет специальный класс SoapExtension, и позволяет .NET присоединить SOAP-расширение к методу. В рассматриваемом примере класс SOAP-расширения имеет имя SoapLog, хотя его код еще не приводился.

```
private int priority;
public override int Priority
{
 get { return priority; }
 set { priority = value; }
}
public override Type ExtensionType
{
 get { return typeof(SoapLog); }
}
```

Помимо этого вы можете добавить свойства, которые будут предоставлять дополнительные фрагменты информации инициализации для вашего SOAP-расширения. В показанном ниже примере добавляется свойство Name, сохраняющее строку с именем источника, которая будет использоваться при записи данных в журнал событий, и свойство Level, которое определяет, какие сообщения будут регистрироваться. Если значение свойства Level равняется 1, расширение SoapLog будет регистрировать только сообщения об ошибках. Если значение свойства Level равняется 2 или больше, расширение SoapLog будет записывать сообщения всех типов. Если значение свойства Level равняется 3 или больше, расширение SoapLog будет добавлять в каждое сообщение дополнительную информацию, отображающую этап, на котором была создана данная запись журнала.

```
private string name = "SoapLog" ;
public string Name
{
 get { return name;}
 set { name = value; }
}
private int level = 1;
public int Level
{
 get { return level;}
 set { level = value; }
}
```

Теперь осталось только применить этот специальный атрибут к какому-то веб-методу и установить значения для свойств Name и Level. В приведенном ниже примере в качестве имени источника данных для регистрации указывается EmployeesService.GetEmployeesCount, а в качестве уровня регистрации (Level) — значение 3.

```
[SoapLog(Name="EmployeesService.GetEmployeesCount", Level=3)]
[WebMethod()]
public int GetEmployeesCount()
{ ... }
```

Теперь при каждом вызове метода GetEmployeesCount() с помощью SOAP-сообщения создается, инициализируется и инициируется класс SoapLog. SOAP-сообщение запроса он может обработать до того, как его получит метод GetEmployeesCount(),

а SOAP-сообщение ответа — после того как метод GetEmployeesCount() возвратит результат.

Класс SoapExtension

Код класса SoapExtension, который мы будем использовать для регистрации сообщений, является достаточно длинным, хотя большая его часть представляет собой базовый шаблон, который применяется в каждом SOAP-расширении. Мы будем рассматривать его по частям.

Первая деталь, на которую следует обратить внимание — это то, что данный класс порождается от абстрактного базового класса SoapExtension, что является обязательным требованием. Класс SoapExtension предоставляет много методов, которые должны перекрываться, среди них.

- *Метод GetInitializer() и метод Initialize().* Эти методы передают начальную информацию SOAP-расширению, когда оно создается впервые.
- *Метод ProcessMessage().* Здесь как раз и происходит фактическая обработка, позволяющая SOAP-расширению просмотреть (и изменить) необработанное SOAP-сообщение.
- *Метод ChainStream().* Этот метод представляет собой базовую часть инфраструктуры, которую должна предоставлять каждая веб-служба. Он позволяет получать доступ к SOAP-потоку, не причиняя вреда другим расширениям.

А вот как выглядит определение этого класса:

```
public class SoapLog : System.Web.Services.Protocols.SoapExtension
{ ... }
```

ASP.NET вызывает метод GetInitializer(), когда ваше расширение впервые используется для какого-нибудь определенного веб-метода, и тем самым предоставляет вам шанс инициализировать и сохранить данные, которые будут необходимы при обработке SOAP-сообщений. Вы сохраняете эти данные, передавая их обратно в виде возвращаемого значения из метода GetInitializer().

После вызова метода GetInitializer() вы получаете одну очень важную деталь информации — специальный атрибут, который был применен к соответствующему веб-методу. В случае с расширением SoapLog это будет экземпляр класса SoapLogAttribute, который предоставляет свойство Name и свойство Level. Чтобы сохранить эту информацию для использования в будущем, вы можете вернуть этот атрибут из метода GetInitializer(), как показано ниже:

```
public override object GetInitializer(LogicalMethodInfo methodInfo,
   SoapExtensionAttribute attribute)
{
 return attribute;
}
```

На самом деле метод GetInitializer() имеет две версии. Вызывается только какая-то одна из них, что зависит того, как конфигурировалось SOAP-расширение: с помощью атрибута (как в рассматриваемом примере) или с помощью конфигурационного файла. Если с помощью конфигурационного файла, оно (SOAP-расширение) будет автоматически запускаться для каждого метода каждой веб-службы.

Даже если вы не планируете использовать конфигурационный файл для инициализации SOAP-сообщения, вы все равно должны реализовать вторую версию метода GetInistializer(). В данном случае имеет смысл вернуть новый экземпляр SoapLogAttribute для того, чтобы используемые по умолчанию параметры Name и Level были доступны позже:

```
public override object GetInitializer(Type obj)
{
 return new SoapLogAttribute();
}
```

Метод GetInitializer() вызывается, только когда ваше SOAP-расширение впервые выполняется для какого-нибудь метода. Однако каждый раз, когда этот метод вызывается, обязательно инициируется метод Initialize(). Если из метода GetInitializer() возвращался какой-нибудь объект, ASP.NET передает его методу Initialize() всякий раз, когда тот вызывается. В случае с расширением SoapLog метод Initialize() прекрасно подходит для извлечения информации об имени источника данных для регистрации (Name) и уровне регистрации (Level) и ее сохранения в переменных экземпляра для того, чтобы она была доступной в остальной части процесса обработки SOAP-сообщений. (Сохранять эту информацию в методе GetInitialize() не имеет смысла, потому что этот метод не будет вызываться каждый раз, когда выполняется SOAP-расширение.)

```
private int level;
private string name;
public override void Initialize(object initializer)
{
 name = ((SoapLogAttribute)initializer).Name;
 level = ((SoapLogAttribute)initializer).Level;
}
```

Наибольшую часть работы в расширении выполняет метод ProcessMessage(), который ASP.NET вызывает на разных этапах процесса сериализации. Объект SoapMessage передается именно методу ProcessMessage(), поэтому данный метод может применяться для извлечения информации о сообщении вроде того, на какой стадии обработки находится конкретное сообщение, или какой текст в нем содержится. Расширение SoapLog считывает сообщение целиком только на этапе AfterSerialize и на этапе BeforeSerialize, потому что это единственные этапы, на которых могут быть считаны все XML-данные SOAP-сообщения. Однако если уровень регистрации равен 3 или выше, базовая запись в журнале будет создаваться на этапах BeforeSerialize и AfterSerialize и фиксировать просто название стадии.

Ниже показан полный код метода ProcccessMessage():

```
public override void ProcessMessage(SoapMessage message)
{
 switch (message.Stage)
 {
  case System.Web.Services.Protocols.SoapMessageStage.BeforeSerialize:
   if (level > 2 )
        WriteToLog(message.Stage.ToString(), EventLogEntryType.Information);
   break;
  case System.Web.Services.Protocols.SoapMessageStage.AfterSerialize:
   LogOutputMessage(message);
   break;
  case System.Web.Services.Protocols.SoapMessageStage.BeforeDeserialize:
   LogInputMessage(message);
   break;
  case System.Web.Services.Protocols.SoapMessageStage.AfterDeserialize:
   if (level > 2 )
        WriteToLog(message.Stage.ToString(), EventLogEntryType.Information);
   break;
  }
}
```

Метод ProcessMessage() не содержит фактического кода регистрации. Вместо этого он вызывает другие методы, такие как WriteLogLog(), LogOutputMessage() и LogInputMessage(). Метод WriteLogLog() — это конечный этап, на котором с помощью класса System.Diagnostics.EventLog создается журнальная запись. Если необходимо, данный код создает новый журнал событий и новый источник данных для этого журнала, используя имя, которое было указано в свойстве Name атрибута специального расширения.

Вот как будет выглядеть код метода WriteToLog():

```
private void WriteToLog(string message, EventLogEntryType type)
{
 // Создаем новый журнал Web Service Log, используя имя
 // источника событий, которое было указано в атрибуте.
 EventLog log;
 if (!EventLog.SourceExists(name))
   EventLog.CreateEventSource(name, "Web Service Log");
 log = new EventLog();
 log.Source = name;
 log.WriteEntry(message, type);
}
```

Когда SOAP-сообщение находится на этапе BeforeSerialize или AfterDeserialize, сразу вызывается метод WriteToLog() и записывается название этапа. Когда SOAP-сообщение находится на этапе AfterSerialize или BeforeDeserialize, потребуется выполнить еще кое-какую работу, чтобы извлечь SOAP-сообщение.

Прежде чем вы сможете создать эти методы, вам понадобится еще один ингредиент — метод CopyStream(). Это связано с тем, что XML-данные SOAP-сообщения содержатся в потоке. Поток имеет указатель, который обозначает текущую позицию в потоке. Проблема состоит в том, что при считывании данных сообщения из потока (например, с целью их записи в журнал), этот указатель передвигается. Это означает, что при считывании потока, которому предстоит десериализация, расширение SoapLog переместит этот указатель в самый конец данного потока. Для того чтобы ASP.NET смог правильно выполнить десериализацию SOAP-сообщения, указатель должен быть перемещен обратно в начало потока. Если не выполнить этот шаг, во время десериализации произойдет ошибка.

Чтобы упростить этот процесс, вы можете использовать индивидуальный метод CopyStream(). Этот метод копирует содержимое одного потока в другой. После выполнения этого метода указатели обеих потоков будут находиться в конце.

```
private void CopyStream(Stream fromstream, Stream tostream)
{
 StreamReader reader = new StreamReader(fromstream);
 StreamWriter writer = new StreamWriter(tostream);
 writer.WriteLine(reader.ReadToEnd());
 writer.Flush();
}
```

Еще один ингредиент, который вам нужен — это метод ChainStream(), который ASP.NET вызывает перед началом процесса сериализации или десериализации. Ваше SOAP-расширение может перекрывать этот метод, вставляя себя в конвейер обработки и выполняя операции кэширования ссылки на исходный поток и создания нового потока в памяти, затем возвращаемого следующему расширению в цепочке.

```
private Stream oldStream;
private Stream newStream;
```

```
public override Stream ChainStream(Stream stream)
{
 oldStream = stream;
 newStream = new MemoryStream();
 return newStream;
}
```

Конечно, это только часть истории. Другие методы могут либо считывать данные из старого потока, либо записывать данные в новый поток, что зависит от того, на какой стадии обработки находится сообщение. Делается это путем вызова метода `CopyStream()`. Реализовав эту несколько запутанную схему, вы получите следующий результат: SOAP-расширения смогут изменять SOAP-поток, не перезаписывая изменения друг друга. В целом методы `ChainStream()` и `CopyStream()` — это базовые компоненты архитектуры SOAP-расширения, которые будут выглядеть одинаково в любом SOAP-расширении, с которым вы будете встречаться.

Методы `LogInputMessage()` и `LogOutputMessage()` отвечают за извлечение информации о сообщении и ее регистрацию в журнале. Оба метода используют метод `CopyStream()`. При десериализации входной поток содержит подлежащие десериализации XML-данные, и указатель расположен в начале потока. Метод `LogInputMessage()` копирует входной поток в буфер потоков в памяти и регистрирует его содержимое. Затем он перемещает указатель в начало находящегося в буфере памяти потока для того, чтобы следующее расширение могло получить к нему доступ.

```
private void LogInputMessage(SoapMessage message)
{
 CopyStream(oldStream, newStream);
 message.Stream.Seek(0, SeekOrigin.Begin);
 LogMessage(message, newStream);
 message.Stream.Seek(0, SeekOrigin.Begin);
}
```

При сериализации сериализатор выполняет запись в поток в памяти, созданный в методе `ChainStream()`. Когда после сериализации вызывается функция `LogOutputMessage()`, указатель находится в конце этого потока. Функция `LogOutputMessage()` перемещает его в начало потока для того, чтобы расширение могло зарегистрировать его содержимое. Перед возвратом содержимое потока в памяти копируется в выходной поток, после чего указатель перемещается обратно в конец обоих потоков.

```
private void LogOutputMessage(SoapMessage message)
{
 message.Stream.Seek(0, SeekOrigin.Begin);
 LogMessage(message, newStream);
 message.Stream.Seek(0, SeekOrigin.Begin);
 CopyStream(newStream, oldStream);
}
```

Переместив указатель потока в нужную позицию, функции `LogInputMessage()` и `LogOutputMessage()` извлекают данные сообщения из SOAP-потока и создают на основе этой информации соответствующую запись в журнале. Помимо этого они также проверяют, не содержит ли SOAP-сообщение какой-нибудь ошибки. Если содержит, они записывают сообщение в журнал событий как ошибку.

```
private void LogMessage(SoapMessage message, Stream stream)
{
 StreamReader reader = new StreamReader(stream);
 eventMessage = reader.ReadToEnd();
 string eventMessage;
```

```
  if (level > 2)
    eventMessage = message.Stage.ToString() +"\n" + eventMessage;
  if (eventMessage.IndexOf("<soap:Fault>") > 0)
  {
    // В теле SOAP-сообщения содержится ошибка.
    if (level > 0)
        WriteToLog(eventMessage, EventLogEntryType.Error);
  }
  else
  {
    // В теле SOAP-сообщения содержится информационное сообщение.
    if (level > 1)
        WriteToLog(eventMessage, EventLogEntryType.Information);
  }
}
```

Этот фрагмент завершает код расширения SoapLog.

Использование расширения SoapLog

Чтобы протестировать расширение SoapLog, примените атрибут SoapLogAttribute к веб-методу, как показано ниже:

```
[SoapLog(Name="EmployeesService.GetEmployeesLogged", Level=3)]
[WebMethod()]
public int GetEmployeesLogged()
{ ... }
```

Далее создайте клиентское приложение, вызывающее этот метод. Когда вы запустите клиентское приложение и вызовете этот метод, расширение SoapLog запустится и создаст соответствующие записи в журнале событий.

На заметку! Чтобы расширение SoapLog успешно выполняло запись в журнал событий, рабочий процесс ASP.NET (в роли которого, как правило, выступает учетная запись ASP.NET) должен иметь права на доступ к журналу событий Windows. В противном случае никакие записи создаваться не будут. Обратите внимание, что в случае, если SOAP-расширение выйдет из строя или сгенерирует исключение на каком-нибудь этапе, оно будет просто проигнорировано. Ни код клиента, ни методы не будут об этом уведомлены.

При желании убедиться в том, что записи появляются, запустите программу просмотра событий Event Viewer (выбрав в меню Start (Пуск) команду Programs⇨Administrative Tools⇨Event Viewer (Программы⇨Инструменты администрирования⇨Программа просмотра событий)). Отыщите журнал под названием Web Service Log (Журнал веб-службы). На рис. 37.5 показаны записи, которые будут отображаться в этом журнале после двукратного вызова метода GetEmployeesCount() с указанием для свойства Level значения 3.

Просмотреть ту или иную запись можно, просто дважды щелкнув на ней. После этого появится диалоговое окно Event Properties (Свойства события), в поле которого будет отображаться полная версия сообщения журнала событий вместе с XML-данными из SOAP-сообщения, как показано на рис. 37.6.

Расширение SoapLog представляет собой довольно удобное средство при разработке или мониторинге веб-служб. Однако его следует использовать с умом. Даже когда отслеживается один единственный метод, количество записей в журнале событий может быстро достигнуть нескольких тысяч. По мере заполнения журнала событий, старые сообщения автоматически удаляются. Эти свойства можно сконфигурировать, щелкнув правой кнопкой мыши на журнале событий в окне Event Viewer (Программа просмотра событий) и выбрав в появившемся контекстном меню команду Properties (Свойства).

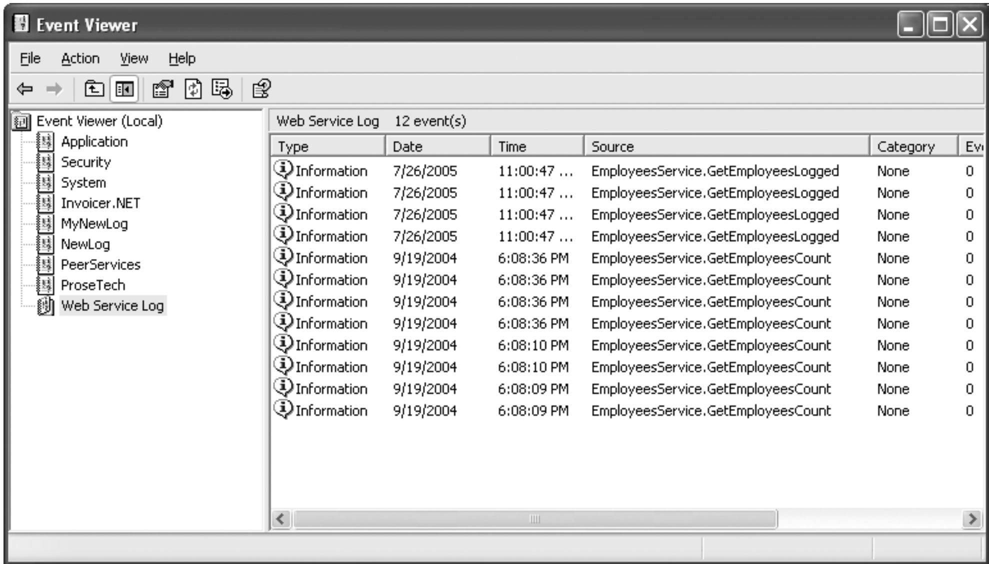


Рис. 37.5. Записи в журнале событий для SOAP-расширения

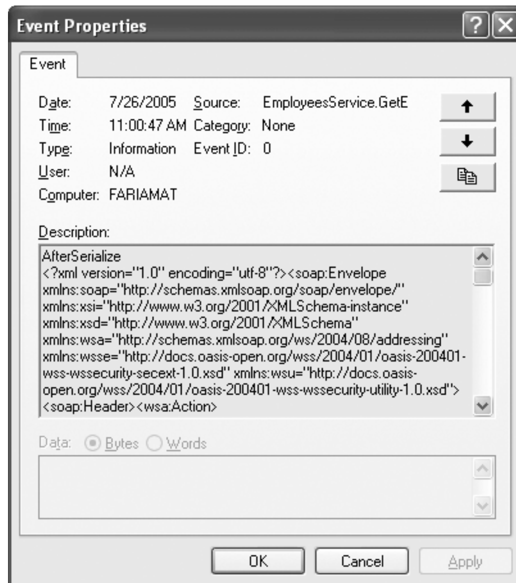


Рис. 37.6. SOAP-сообщение в записи журнала событий

Набор инструментальных средств Web Services Enhancements

С появлением еще самой первой версии .NET стандарты веб-служб начали стремительно развиваться. На самом деле разработчики по сей день продолжают разрабатывать все новые и новые стандарты, тестируют, корректируют и стандартизируют их. В будущих версиях .NET многие из этих новых стандартов будут включены в библиотеку классов. Поскольку эти стандарты являются достаточно новыми и требуют доработки, они пока еще непригодны для использования. Однако вы можете загрузить другое средство от Microsoft — бесплатный набор инструментальных средств WSE — предоставляющее поддержку для массы новых, существующих на сегодняшний день стандартов веб-служб.

На заметку! В целом ситуация выглядит так: стандарты SOAP, WSDL и UDDI являются частью .NET, а все недавние касающиеся их улучшения (такие как SOAP 1.2) включены в версию .NET 2.0. Однако стандарты, которые базируются на этих трех основных (вроде стандартов для SOAP-шифрования, безопасности и т.д.), пока что не входят в состав .NET и доступны исключительно в WSE.

Загрузить набор инструментальных средств WSE (или просто почитать о нем) можно по адресу `http://msdn.microsoft.com/webservices/building/wse`. WSE предоставляет сборку библиотеки классов с набором полезных классов. "За кулисами" WSE использует SOAP-расширения для настройки сообщений веб-служб. Это означает, что ваш код взаимодействует с набором вспомогательных объектов, а в это время WSE "за кулисами" реализует соответствующие SOAP-стандарты. Применение набора WSE имеет два недостатка. Во-первых, этот набор инструментальных средств продолжает изменяться. Версия, которая работает с Visual Studio 2005 (версия WSE 3.0), не совместима с предыдущими версиями. Во-вторых, набор WSE должен использоваться как на клиенте, так и на сервере. Другими словами, если вы хотите использовать веб-службу, в которой для поддержки новой функциональной возможности применяется WSE, ваше клиентское приложение тоже должно использовать WSE (если это .NET-клиент) или другой набор инструментальных средств, который поддерживает такие же стандарты (если это отличный от .NET клиент).

Многие из доступных в WSE функциональных возможностей реализуются посредством специального класса `SoapContext`. Ваш код взаимодействует с объектом `SoapContext`. Затем, когда вы отправляете сообщение, различные фильтры (SOAP-расширения) проверяют значения свойств этого объекта `SoapContext` и, если требуется, создают SOAP-заголовки для исходящего сообщения. Этот процесс показан на рис. 37.7.

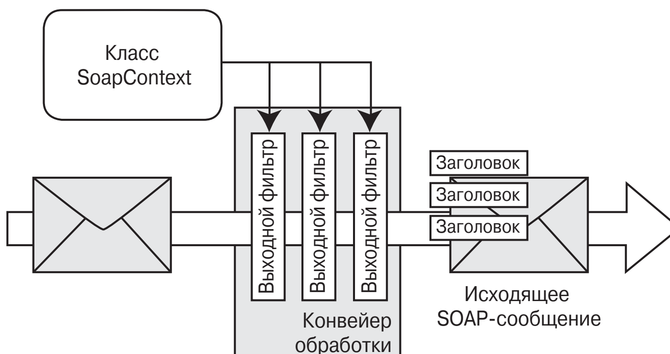


Рис. 37.7. Обработка исходящего SOAP-сообщения

В фильтрах интересно то, что они могут применяться только тогда, когда в них есть необходимость. Например, если вам не нужен заголовок безопасности, вы можете оставить фильтр для заголовков безопасности за пределами конвейера обработки, просто соответствующим образом настроив конфигурационные параметры. Кроме того, поскольку задачей каждого фильтра является добавление конкретного заголовка, одновременно может применяться абсолютно любое (как большое, так и небольшое) количество фильтров.

Когда SOAP-сообщение получено, начинает выполняться обратный процесс. В этом случае фильтры выполняют поиск определенных SOAP-заголовков и используют содержащуюся в них информацию для конфигурирования объекта `SoapContext`. Эта модель обработки показана на рис. 37.8.

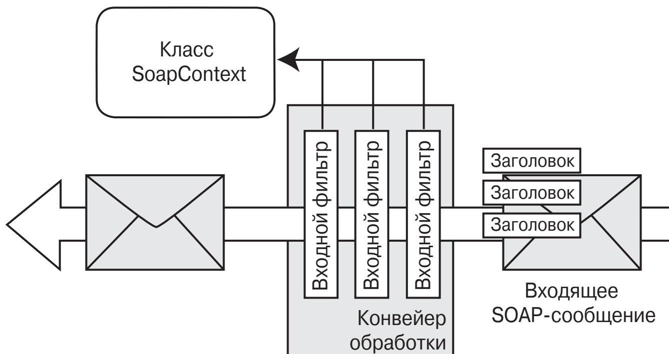


Рис. 37.8. Обработка входящего SOAP-сообщения

Итак, какие же стандарты поддерживаются набором WSE? Полный список поддерживаемых WSE стандартов можно найти в прилагаемой к этому набору документации, но в целом WSE включает поддержку для шифрования аутентификационных данных и сообщений, установки доверительных отношений, маршрутизации сообщений и функциональной совместимости. WSE также позволяет использовать SOAP-сообщения для установки прямой связи через TCP-соединение (никакого протокола HTTP для этого не требуется). В следующих разделах мы покажем, как WSE используется в простом сценарии обеспечения безопасности. Полную информацию о наборе WSE и способах его применения можно найти в соответствующих книгах или в онлайновой документации, поставляемой вместе с этим набором.

Инсталляция WSE

Прежде чем двигаться дальше, вы должны загрузить и установить набор WSE. Когда запустите программу установки, в качестве типа установки выберите Visual Studio Developer, если у вас установлена версия Visual Studio 2003, потому что такой тип установки позволит работать с проектами (рис. 37.9).

Чтобы использовать WSE в проекте, выполните следующие действия: в Visual Studio щелкните правой кнопки мыши на имени нужного проекта в окне проводника Solution Explorer и выберите расположенную в нижней части контекстного меню команду WSE Settings (Настройки WSE). Появится диалоговое окно, в котором будут отображаться два флажка. Если вы создаете веб-службу, отметьте оба этих флажках, как показано на рис. 37.10. Если вы создаете клиентское приложение, отметьте только флажок Enable This Project for Web Services Enhancements (Разрешить использовать в этом проекте набор Web Services Enhancements).

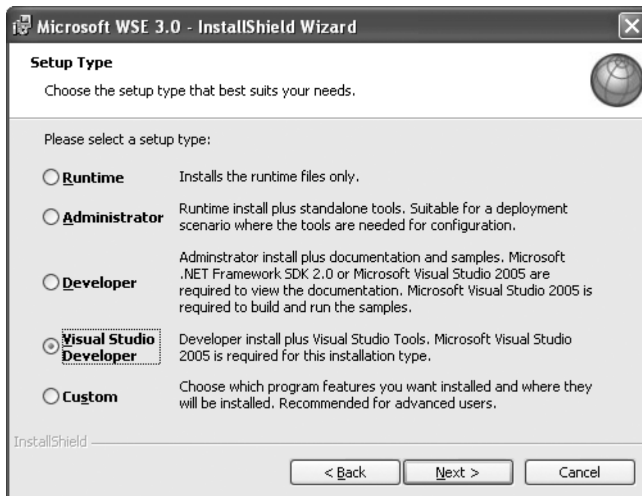


Рис. 37.9. Инсталляция WSE

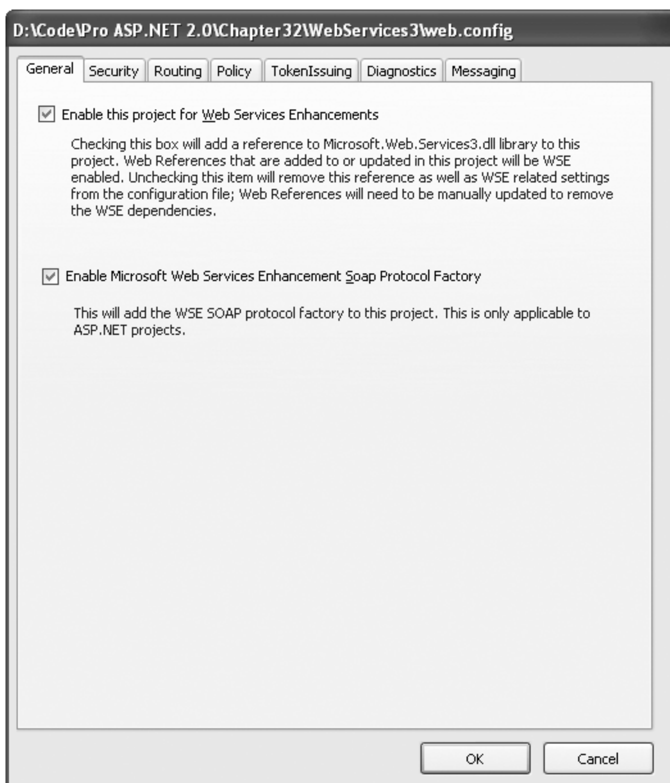


Рис. 37.10. Включение возможности использовать WSE для проекта

Когда отмечается первый флажок (Enable This Project for Web Services Enhancements (Разрешить использовать в этом проекте набор Web Services Enhancements)), Visual Studio автоматически добавляет ссылку в сборку `Microsoft.Web.Services2.dll` и изменяет файл `web.config` приложения, добавляя в него обработчик конфигурации WSE. Кроме того, любые создаваемые с этого момента веб-ссылки будут включать в прокси-классе поддержку для WSE. Однако веб-ссылки, которые были созданы до этого момента, не будут включать поддержку для WSE до тех пор, пока вы их не обновите.

Когда отмечается второй флажок (Enable Microsoft Web Services Enhancements SOAP Extensions (Разрешить использовать SOAP-расширения из набора Microsoft Web Services Enhancements)), Visual Studio изменяет файл `web.config`, чтобы зарегистрировать SOAP-расширение, которое добавляет поддержку для ваших веб-служб. Эта опция является обязательной только для веб-служб ASP.NET, использующих WSE.

Выполнение аутентификации с помощью WSE

Многие из классов WSE поддерживают стандарты поддержки безопасности. Одним из наиболее простых и последовательных из этих классов является класс `UsernameToken`, который представляет данные удостоверения пользователя.

Информация класса `UsernameToken` добавляется в сообщение в виде SOAP-заголовка. Однако она добавляется способом, отвечающим стандарту WS-Security, который может оказаться достаточно трудно реализовать собственными силами. Выгода состоит в том, что, реализуя такую общую модель безопасности, вы можете добавить аутентификацию, не разрабатывая никакого оригинального подхода, который мог бы усложнить применение вашей веб-службы в сценариях с разными платформами и сторонними приложениями. Также, скорее всего, ASP.NET и WS-Security способны обеспечить подход, более безопасный и надежный, нежели тот, который может разработать разработчик или организация (не затрачивая много времени и усилий).

На текущий момент инфраструктура системы безопасности в WSE имеет несколько пробелов и не выполняет полностью своих обещаний. Однако ее легко использовать и расширять. Чтобы понять, как она работает, лучше всего рассмотреть простой пример.

Определение данных удостоверения в клиентском приложении

В следующем примере мы покажем, как можно использовать WSE для выполнения безопасного вызова веб-службы `EmployeesService`. Первое, что потребуется сделать — это добавить веб-ссылку. При добавлении ссылки на веб-службу `EmployeesService` в допускающем использование WSE проекте (т.е. для которого был отмечен флажок Enable This Project for Web Services Enhancements (Разрешить использовать в этом проекте набор Web Services Enhancements)), Visual Studio фактически создаст *два* прокси-класса. Первый прокси-класс будет иметь такое же имя, как и класс веб-службы, и не будет ничем отличаться от классов веб-служб, генерируемых в не поддерживающих использование WSE проектах. Второй прокси-класс будет иметь имя, к которому будет прикреплен суффикс *`WSE`*. Этот класс порожден от класса `Microsoft.Web.Services3.WebServicesClientProtocol` и включает поддержку функциональных возможностей WSE (в данном случае — добавления маркеров WS-Security).

Таким образом, чтобы использовать WS-Security с веб-службой `EmployeesService`, необходимо создать допускающий применение WSE проект, добавить или обновить веб-ссылку и затем изменить код так, чтобы в нем использовался прокси-класс `EmployeeServiceWse`. Выполнив эти шаги, можно добавить новый объект `UsernameToken` с данными удостоверения с помощью такого кода:

```
// Создаем прокси.
EmployeesServiceWse proxy = new EmployeesServiceWse();
// Добавляем маркер WS-Security.
proxy.RequestSoapContext.Security.Tokens.Add(
new UsernameToken(userName, password, PasswordOption.SendPlainText));
// Привязываем результаты.
GridView1.DataSource = proxy.GetEmployees().Tables[0];
```

В этом коде видно, что маркер доступа добавляется как объект UsernameToken. Он вставляется в коллекцию Security.Tokens объекта RequestSoapContext, являющегося объектом SoapContext, который представляет подготавливаемое к отправке сообщение запроса.

Чтобы использовать этот код в таком виде, каком он здесь показан, необходимо импортировать следующие пространства имен:

```
using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;
```

На заметку! Обратите внимание на наличие в пространствах имен WSE цифры 3, которая обозначает третью версию набора WSE. Это связано с тем, что третья версия не обеспечивает обратной совместимости с предыдущими двумя. Во избежание конфликта с частично обновленными приложениями, WSE-классы разделяются на отдельные пространства имен по версии. Это пример одного из недостатков работы с новыми (еще развивающимися) стандартами веб-служб.

Конструктор класса UsernameToken принимает три параметра: строку с именем пользователя, строку с паролем и опцию хеширования. К сожалению, при желании сделать так, чтобы использовался стандартный поставщик аутентификации в WSE (который применяет аутентификацию Windows), выбрать можно *только* опцию PasswordOption.SendPlain.Text. В этом случае код будет совершенно незащищенным, и будет уязвимым для сетевого шпионажа, если только запрос не будет пересылаться через SSL-соединение.

Хотя в рассматриваемом примере в запрос добавляются только две дополнительных детали, SOAP-сообщение становится намного более сложным из-за структуры, которую использует стандарт WS-Security. Он определяет дополнительные детали, такие как Expiration Data (Дата истечения срока) (применяется для предотвращения атак типа воспроизведения) и nonce (случайное значение, которое может включаться в хеш-значение для усиления защиты). Ниже показан несколько сокращенный пример SOAP-сообщения запроса.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
 mlns:wsse="http://docs.oasis-open.org/wss/2004/01/..."
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/...">
 <soap:Header>
 <wsa:Action>http://www.apress.com/ProASP.NET/GetEmployees</wsa:Action>
 <wsa:MessageID>uuid:5b1bc235-7f81-40c4-ac1e-e4ea81ade319</wsa:MessageID>
 <wsa:ReplyTo>
  <wsa:Address>http://schemas.xmlsoap.org/ws/2004/03/...</wsa:Address>
 </wsa:ReplyTo>
 <wsa:To>http://localhost:8080/WebServices3/EmployeesService.asmx</wsa:To>
 <wsse:Security soap:mustUnderstand="1">
 <wsu:Timestamp wsu:Id="Timestamp-dc0d8d9a-e385-438f-9ff1-2cb0b699c90f">
  <wsu:Created>2004-09-21T01:49:33Z</wsu:Created>
  <wsu:Expires>2004-09-21T01:54:33Z</wsu:Expires>
 </wsu:Timestamp>
```

```
 <wsse:UsernameToken xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/..."
  wsu:Id="SecurityToken-8b663245-30ac-4178-b2c8-724f43fc27be">
  <wsse:Username>guest</wsse:Username>
  <wsse:Password
  Type="http://docs.oasis- open.org/wss/2004/01/...">secret</wsse:Password>
  <wsse:Nonce>9m8UofSBhw+XWIqfO83NiQ==</wsse:Nonce>
  <wsu:Created>2004-09-21T01:49:33Z</wsu:Created>
 </wsse:UsernameToken>
 </wsse:Security>
 </soap:Header>
 <soap:Body>
  <GetEmployees xmlns="http://www.apress.com/ProASP.NET/" />
 </soap:Body>
</soap:Envelope>
```

Считывание данных удостоверения в веб-службе

Поддерживающая WSE служба исследует предоставленный маркер и сразу же проверяет его достоверность. Стандартный (используемый по умолчанию) поставщик аутентификации, включенный в набор WSE, реализует аутентификацию Windows, а это означает, что он извлекает имя пользователя и пароль из SOAP-заголовка и использует эту информацию для подключения пользователя к веб-службе через учетную запись Windows. Если маркер не отображается ни на какую действительную учетную запись Windows, клиенту возвращается сообщение об ошибке. Однако если маркеры не предоставляются, не происходят и ошибки. Вы должны сами позаботиться о том, чтобы на веб-сервере выполнялась проверка этого условия, если хотите ограничить доступ к каким-то определенным веб-методам.

К сожалению, интегрировать набор WSE так, чтобы он мог использовать объект `User`, не получится. Вместо этого маркеры придется извлекать из текущего контекста. WSE предоставляет `RequestSoapContext`. Используя свойство `RequestSoapContext.Current`, можно извлечь экземпляр класса `SoapContext`, который представляет последнее полученное сообщение, а затем проверить коллекцию `SoapContext.Security.Tokens`.

Чтобы упростить эту задачу, не помешает создать индивидуальный метод, подобный тому, который показан ниже. Этот метод проверяет, существует ли маркер, и если нет — генерирует исключение. В противном случае он возвращает имя пользователя.

```
private string GetUsernameToken()
{
 // Хотя маркеров может быть и много, только
 // один из них будет маркером UsernameToken.
 foreach (UsernameToken token in RequestSoapContext.Current.Security.Tokens)
 {
  return token.Username;
 }
 throw new SecurityException("Missing security token");
}
```

Этот метод `GetUsernameToken()` можно было бы вызвать в начале веб-метода для гарантии действия системы безопасности. В целом это хороший подход для обеспечения безопасности. Однако важно помнить о его ограничениях. Во-первых, он не поддерживает хеширование или шифрование данных удостоверения пользователя. А также он не поддерживает более совершенные протоколы аутентификации Windows, такие как аутентификация типа Digest и встроенная аутентификация Windows. Кроме того, клиент всегда должен предоставлять пароль и имя пользователя. Клиент не имеет возможности автоматически предоставлять данные удостоверения текущего пользователя,

как демонстрировалось ранее в этой главе в примере с объектом CredentialCache. На самом деле свойство Credentials прокси-класса игнорируется полностью.

К счастью, ваши возможности не ограничиваются применением только имеющей низкую степень масштабируемости версии аутентификации Windows, предоставляемой используемой по умолчанию службой аутентификации WSE. Вы также можете создавать и свою собственную логику аутентификации, о чем будет рассказываться уже в следующем разделе.

Специальная аутентификация

Создавая свой собственный класс аутентификации, вы можете выполнять аутентификацию с помощью любого источника данных, включая XML-файл и базу данных. Чтобы создать свой собственный аутентификатор, потребуется просто создать класс, порожденный от класса UsernameTokenManager и переопределяющий метод AuthenticateToken(). В этом методе код должен отыскивать пользователя, пытающегося пройти аутентификацию, и возвращать пароль для этого пользователя. После этого ASP.NET сравнит этот пароль с данными удостоверения пользователя и решит, успешно ли прошла аутентификация.

Создание этого класса представляет собой достаточно простой процесс. Ниже показан пример такого класса, который просто возвращает два жестко закодированных пароля для двух пользователей. Этот пример позволяет быстро и легко протестировать такой класс, хотя в реальном мире для извлечения этой же информации, возможно, использовался бы код ADO.NET.

```
public class CustomAuthenticator : UsernameTokenManager
{
 protected override string AuthenticateToken(UsernameToken token)
 {
  string username = token.Username;
  if (username == "dan")
    return "secret";
  else if (username == "jenny")
    return "opensesame";
  else
    return "";
 }
}
```

Вам не стоит выполнять проверку пароля самостоятельно из-за того, что выбор типа сравнения зависит от того, как кодируются данные удостоверения. Например, если они передаются в виде открытого текста, вам следует выполнить простое строковое сравнение. Если они хешируются, необходимо создать новое хеш-значение для пароля, используя тот же самый стандартизированный алгоритм, и сделать так, чтобы в нем учитывались все те же самые данные (включая случайное значение nonce из сообщения клиента). Однако WSE может выполнять эту связанную со сравнением задачу автоматически, что значительно упрощает вашу логику. Единственная проблема состоит в том, что вы должны сохранять пароль пользователя на веб-сервере в извлекаемом виде. Если вы сохраняете только хеш-значения паролей в базе данных, вы не сможете передать исходный пароль ASP.NET и ASP.NET не сможет воссоздать хеш-значение удостоверения, необходимое ему для аутентификации пользователя.

Когда класс аутентификации уже создан, вы должны указать WSE на необходимость использования этого класса для аутентификации маркеров доступа пользователей, зарегистрировав его в файле web.config. Чтобы сделать это, щелкните правой кнопкой мыши на имени проекта в окне проводника Solution Explorer и выберите в контекстном

меню команду WSE Settings (Настройки WSE). Далее в появившемся диалоговом окне перейдите на вкладку Security (Безопасность), как показано на рис. 37.11.

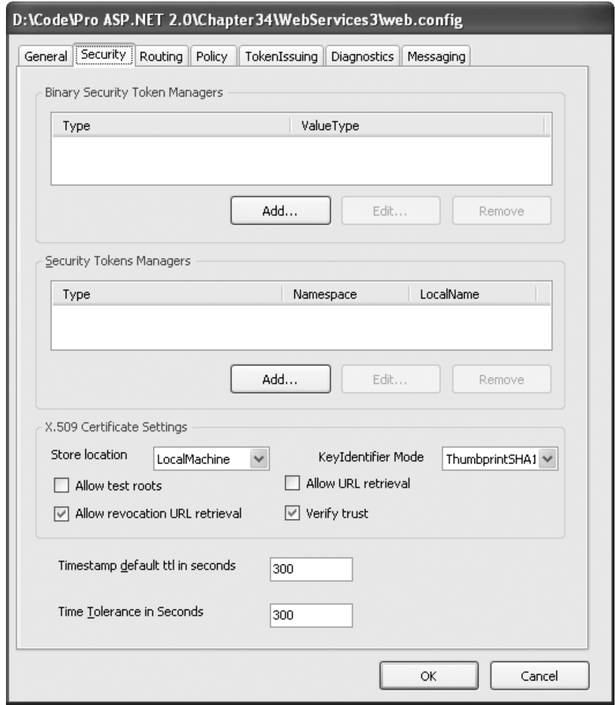


Рис. 37.11. Параметры безопасности

В разделе Security Tokens Managers (Диспетчеры маркеров доступа) щелкните на кнопке Add (Добавить). Появится диалоговое окно SecurityToken Manager (Диспетчер `SecurityToken`), показанное на рис. 37.12.

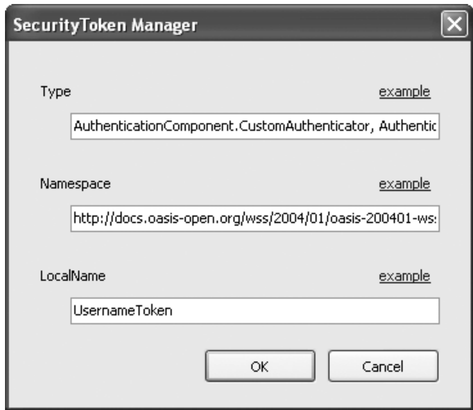


Рис. 37.12. Конфигурирование нового диспетчера `UsernameTokenManager`

В этом диалоговом окне будут отображаться три поля.

- *Type (Тип).* Введите в этом поле полностью квалифицированное имя класса, поставьте после него запятую и введите имя сборки (без расширения `.dll`). Например, если проект называется `MyProject`, а класс аутентификации — `CustomAuthenticator`, введите здесь следующее значение: `MyProject.CustomAuthenticator, MyProject`.
- *Namespace (Пространство имен).* Введите в этом поле следующую жестко закодированную строку: `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd`.
- *LocalName (Локальное имя).* Введите в этом поле следующую жестко закодированную строку — `UsernameToken`.

Клиент создается, по сути, тем же самым способом, независимо от того, как аутентификация выполняется на сервере. Однако в этом случае при создании специального класса аутентификации для использования могут быть выбраны и хешированные пароли, как показано ниже:

```
// Создаем прокси.
EmployeesServiceWse proxy = new EmployeesServiceWse();

// Добавляем маркер WS-Security.
proxy.RequestSoapContext.Security.Tokens.Add(
  new UsernameToken("dan", "secret", PasswordOption.SendHashed));

// Привязываем результаты.
GridView1.DataSource = proxy.GetEmployees().Tables[0];
GridView1.DataBind();
```

Вся прелесть состоит в том, что вручную хешировать или шифровать пароль пользователя не нужно. WSE хеширует его автоматически на основе ваших инструкций и выполняет хеш-сравнение на стороне сервера. WSE даже использует случайное значение `nonce` для предотвращения атак типа воспроизведения.

Резюме

В этой главе был рассмотрен целый ряд усовершенствованных SOAP-технологий, которые позволяют асинхронно вызывать веб-службы, усиливать безопасность и использовать SOAP-расширения. Очевидно, что стандарты веб-служб продолжают развиваться, поэтому следующие версии .NET Framework наверняка уже будут включать какие-то новые стандарты и какие-нибудь другие, чрезвычайно мощные возможности.