

KAPITOLA 34

Pokročilé webové služby

V posledních dvou kapitolách jsme se podrobně zabývali fungováním webových služeb v prostředí ASP.NET. Pomocí technik, s nimiž jste se seznámili, můžete vytvářet webové služby, které vystavují data pro jiné aplikace a organizace. Na Internetu můžete používat jak webové služby, které jsou založeny na .NET, tak i služby, které .NET nepoužívají.

Náš příběh nicméně ještě nekončí. V této kapitole se naučíte, jak si rozšířit dovednosti týkající se webových služeb o specifické techniky, které jsou často důležité v situacích skutečných webových služeb.

Zaměříme se na tři hlavní oblasti:

- **Asynchronní volání webových služeb.** Volání webových služeb zabírá čas, zejména tehdy, pokud je webový server umístěn na druhém konci zeměkoule a připojen pomalým síťovým připojením. Asynchronní volání vám umožňuje dále pracovat, zatímco čekáte na odpověď.
- **Zabezpečení webových služeb.** V části IV jste se naučili, jak zabezpečit webové stránky tak, abyste zabránili v jejich používání anonymními uživateli. Podobné techniky můžete rovněž použít pro zabezpečení svých webových služeb.
- **Použití rozšíření SOAP.** Infrastruktura webových služeb je pozoruhodně rozšiřitelná díky modelu rozšíření SOAP, který vám umožní vytvářet komponenty, které se zapojují do procesu serializace a deserializace SOAP. V této kapitole si ukážeme základní rozšíření SOAP a stručně si popíšeme sadu nástrojů WSE (Web Services Enhancements), která používá rozšíření SOAP, a která poskytuje podporu několika nově vzniklým standardům.

Tyto tři oblasti vycházejí z konceptů, se kterými jste se seznámili v posledních dvou kapitolách.

POZNÁMKA Ačkoliv .NET 2.0 používá stejný model asynchronního programování jako .NET 1.x, v této kapitole si ukážeme, že asynchronní podpora se v rámci třídy proxy poněkud změnila. Techniky pro zabezpečení webových služeb a používání rozšíření SOAP se nezměnily, ačkoliv standardy webových služeb se neustále vyvíjejí a jsou poskytovány prostřednictvím nových verzí sady nástrojů WSE.

Asynchronní volání

Jak jste se dozvěděli v kapitole 32, .NET Framework chrání programátora před složitostmi volání webové služby, a to tím, že poskytuje vašim aplikacím třídu proxy. Kód, který používá třídu proxy, vypadá vždy stejně, bez ohledu na to, zdali se webová služba nachází na stejném počítači, v lokální síti, nebo na Internetu.

Přes zdánlivou podobnost je podkladové řešení, které je používáno pro volání webové služby, velice odlišné od volání funkce během procesu. Nejenom, že takové volání musí být zaobaleno do SOAP zprávy, ale musí být přenášeno sítí použitím HTTP. Kvůli charakteru Internetu se může čas, který je potřebný k zavolání webové služby, značně lišit pro jednotlivá volání. Ačkoliv váš klient nemůže nijak urychlit vyvolání webové metody, během čekání na odpověď může vykonávat další operace. Může například pokračovat ve výpočtech, ve čtení dat ze systému souborů či z databáze, nebo dokonce volat další webové služby. Ačkoliv se návrh asynchronního volání podstatně obtížněji implementuje, v určitých situacích může být značně přínosný.

Obecně má asynchronní zpracování význam v těchto dvou případech:

- Vytváříte-li Windows aplikaci. V tomto případě asynchronní volání umožní, aby uživatelské rozhraní mohlo reagovat na další požadavky uživatele.
- Provádíte-li nějakou práci zahrnující náročné výpočty, nebo pokud se potřebujete dostat k dalším zdrojům, které mají čekací dobu. Prostřednictvím asynchronního volání webové služby můžete tyto úkoly vykonávat během čekání na odpověď. Zvláštním případem je situace, kdy potřebujete zavolat několik nezávislých webových služeb. Díky asynchronnímu zpracování je můžete všechny zavolat asynchronně, čímž snížíte celkovou dobu čekání.

Je důležité si také uvědomit, kdy byste asynchronní volání neměli používat. Asynchronní volání nijak nezkrátí čas potřebný k získání odpovědi. Řečeno jinými slovy – ve většině webových aplikací, které používají webové služby, není žádný důvod k použití asynchronního volání webové služby, protože váš kód stejně musí čekat na získání odpovědi, než bude moci realizovat finální stránku a odeslat ji klientovi. Pokud však potřebujete zavolat najednou mnoho webových služeb, nebo pokud během čekání můžete provádět další úkoly, lze zkrátit celkový čas zpracování požadavku o několik milisekund. V následujících sekcích si ukážeme, jak asynchronní volání mohou zkrátit čas strávený s webovým klientem a poskytnout mnohem interaktivnější uživatelské rozhraní s klientem Windows.

POZNÁMKA Asynchronní volání fungují v ASP.NET 2.0 poněkud odlišně. Třída proxy již nemá vestavěné metody `BeginXxx()` a `EndXxx()`. Pro notifikace asynchronních operací založených na událostech ovšem můžete použít jiný přístup, který je vestavěn do třídy proxy, a který má význam u déle spuštěných klientů, jako jsou například aplikace Windows.

Asynchronní delegáti

Asynchronní vlákna můžete v .NET použít několika způsoby. Všichni delegáti poskytují metody `BeginInvoke()` a `EndInvoke()`, které vám umožní je spustit na jednom z vláken z fondu vláken CLR. Právě na tuto techniku, která je pohodlná a dobře rozšiřitelná, se zaměříme v této sekci. Další možností je použití třídy `System.Threading.Thread`, pomocí které explicitně vytvoříte nové vlákno, přičemž máte zcela pod kontrolou jeho prioritu a dobu života.

Jak už víte, delegáti jsou typově bezpeční ukazatelé funkce, které tvoří základ pro události .NET. Vytvořme si nyní delegáta, který se odkazuje na specifickou metodu, přičemž tuto metodu můžeme zavolat prostřednictvím delegáta.

Prvním krokem je definování delegáta na úrovni jmenného prostoru (pokud již není přítomen v knihovně třídy .NET). Zde je příklad delegáta, který může odkazovat na libovolnou metodu, která akceptuje jediný celočíselný parametr a vrací celé číslo:

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

Nyní si představte třídu, která má metodu, odpovídající tomuto delegátovi:

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

Můžete vytvořit proměnnou delegáta, která se bude odkazovat na metodu se stejnou signaturou. Zde uvádíme příslušný kód:

```
MyOClass myObj = new MyClass();
// Vytvoř delegáta, který odkazuje na metodu myObj.DoubleNumber().
DoSomethingDelegate doSomething = new DoSomethingDelegate(myObj.DoubleNumber);
// Vyvolej metodu myObj.DoubleNumber() prostřednictvím delegáta.
int doubleValue = doSomething(12);
```

Možná jste si neuvědomili, že delegáti mají rovněž vestavěny inteligentní funkce vláken. Při každém definování delegáta (jako například DoSomethingDelegate v předchozím příkladě), je vygenerována vlastní třída delegáta, která je přidána do vaší assembly. (Vlastní třída delegáta je potřebná, protože kód jednotlivých delegátů se liší v závislosti na signatuře metody, kterou jste definovali.) Když vyvoláte metodu prostřednictvím delegáta, v podstatě se spoléháte na metodu Invoke() z třídy delegáta.

Metoda Invoke() synchronně vykonává odkazovanou metodu. Třída delegáta však také obsahuje metody pro asynchronní volání – BeginInvoke() a EndInvoke(). Pokud použijete BeginInvoke(), volání se vrátí ihned, nicméně neposkytne návratovou hodnotu. Metoda je místo toho zařazena na začátek dalšího vlákna. Při vyvolání BeginInvoke() dodáváte všechny parametry původní metody a dva dodatečné parametry pro volitelné objekty zpětného volání a stavu. Pokud tyto údaje nepotřebujete, (což bude popsáno dále v této sekci), jednoduše předejte referenci null.

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

BeginInvoke() neposkytuje návratovou hodnotu výchozí metody. Místo toho vrací objekt IAAsyncResult, který můžete prozkoumat a rozhodnout se, kdy je asynchronní operace kompletní. Abyste si mohli později vyzvednout výsledky, odešlete objekt IAAsyncResult odpovídající metodě EndInvoke() delegáta. Pokud operace ještě neskončila, metoda EndInvoke() počká, až bude kompletní, a poté poskytne skutečnou návratovou hodnotu. Pokud se v metodě, kterou jste vykonali asynchronně, vyskytne neošetřená výjimka, po zavolání metody EndInvoke() se výsledky objeví na konci vašeho kódu.

Zde uvádíme předchozí příklad, který byl přepsán tak, aby asynchronně volal delegáta:

```

MyClass myObj = new MyClass();
// Vytvoř delegáta, který odkazuje na metodu myObj.DoubleNumber().
DoSomethingDelegate doSomething = new DoSomethingDelegate(myObj.DoubleNumber);
// Spust' metodu myObj.DoubleNumber() v rámci jiného vlákna.
IAsyncResult handle = doSomething.BeginInvoke(originalValue, null, null);
// (Během vykonávání myObj.DoubleNumber() dělej něco jiného.)
// Získej výsledky a (synchronně) počkej, pokud ještě nejsou hotové.
int doubleValue = doSomething.EndInvoke(handle);

```

Abyste prostřednictvím této techniky získali některé výhody multithreadingu, můžete pomocí `BeginInvoke()` asynchronně zavolat několik metod. Předtím, než budeme pokračovat, můžete pro všechny tyto metody zavolat `EndInvoke()`.

Jednoduché asynchronní volání

Následující příklad demonstruje rozdíly mezi synchronním a asynchronním kódem. Abyste si tento příklad mohli vyzkoušet, musíte uměle zpomalit váš kód, čímž nasimulujete podmínky velkého zatížení nebo řadu časově náročných úkolů. Ve webové službě přidejte k metodě `GetEmployees()` následující řádek, čímž vytvoříte zpoždění čtyř sekund:

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

Dále vytvořte webovou stránku, která bude používat webovou službu. Tato webová stránka definuje soukromou metodu, která simuluje časově náročný úkol, opět s použitím metody `Thread.Sleep()`. Zde uvádíme kód, který musíte přidat k webové stránce:

```

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

```

Na vaší stránce musíte vykonat obě metody. Pomocí jednoduchého fragmentu časovacího kódu můžete porovnat synchronní přístup s asynchronním. Podle toho, na které tlačítko uživatel klikne, se provedou tyto operace buď synchronně (jedna po druhé) nebo najednou asynchronně.

Zde je kód pro synchronní vyvolání obou metod:

```

protected void cmdSynchronous_Click(object sender, System.EventArgs e)
{
    // Zaznamenej čas spuštění.
    DateTime startTime = DateTime.Now;

    // Získej data webové služby.
    EmployeesService proxy = new EmployeesService();
    try
    {
        GridView1.DataSource = proxy.GetEmployees();
    }
    catch (Exception err)

```



```

{
    lblInfo.Text = "Problem contacting web service.";
    return;
}
GridView1.DataBind();
// Vykonej některé časově náročné úkoly.
DoSomethingSlow();
// Urči celkový spotřebovaný čas.
TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
lblInfo.Text = "Synchronous operations took " + timeTaken.TotalSeconds +
" seconds.";
}

```

Pro použití asynchronních delegátů musíte nadefinovat delegáta, který odpovídá signatuře metody, kterou chcete asynchronně volat. V tomto případě je to metoda `GetEmployees()`:

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

Zde je kód, kterým spustíte webovou službu jako první, takže operace se budou překrývat:

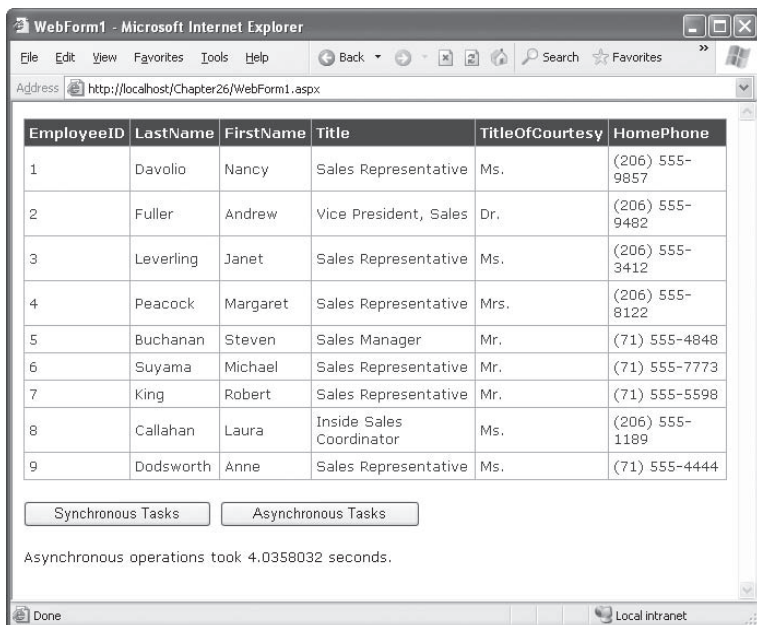
```

protected void cmdAsynchronous_Click(object sender, System.EventArgs e)
{
    // Zaznamenej čas spuštění.
    DateTime startTime = DateTime.Now;
    // Spust' webovou službu v rámci jiného vlákna.
    EmployeesService proxy = new EmployeesService();
    GetEmployeesDelegate async = new GetEmployeesDelegate(proxy.GetEmployees);
    IAsyncResult handle = async.BeginInvoke(null, null);
    // Vykonej některé časově náročné úkoly.
    DoSomethingSlow();
    // Získej výsledek. Pokud ještě není hotov, počkej.
    try
    {
        GridView1.DataSource = async.EndInvoke(handle);
    }
    catch (Exception err)
    {
        lblInfo.Text = "Problem contacting web service.";
        return;
    }
    GridView1.DataBind();
    // Urči celkový spotřebovaný čas.
    TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
    lblInfo.Text = "Asynchronous operations took " + timeTaken.TotalSeconds +
" seconds.";
}

```

Všimněte si, že ovladač výjimky zaobaluje metodu `EndInvoke()`, ale nikoliv metodu `BeginInvoke()`. Je to způsobeno tím, že pokud se během zpracování požadavku vyskytne jakákoliv výjimka (bez ohledu na to, zdali se jedná o problém sítě, nebo o výjimku na straně serveru), váš kód ji neobdrží, dokud nezavoláte metodu `EndInvoke()`.

Až budete spouštět tyto příklady, zjistíte, že synchronní kód zabere 7 až 8 sekund, zatímco asynchronní kód pouze 4 až 5 sekund. Obrázek 34-1 zobrazuje webovou stránku s uvedením času v dolní části stránky.



Obrázek 34-1. Test volání asynchronní metody.

TIP Zapamatujte si, že výhodnost threadingu závisí na typu operace. V tomto příkladě byl threading přínosem, protože dané operace nejsou vázány na procesor – pouze nečinně čekají. S podobným chováním se setkáte při kontaktování externích webových služeb nebo databází. Pokud se však pokusíte použít threading pro současné spuštění dvou úkolů, které využívají stejný procesor na jednom počítači (pokud má počítač pouze jeden procesor), threading nebude výhodný, protože oba úkoly spotřebují přibližně polovinu zdrojů procesoru a jejich vykonání bude trvat přibližně dvakrát déle. Threading je tak ideální pro webové služby, ale už je méně výhodný pro zbytek vašeho obchodního kódu.

Souběžné asynchronní volání

Objekt `IAsyncState` vám poskytuje několik dalších možností, které jsou užitečné při volání více webových metod najednou. Klíčovým je objekt `IAsyncState.WaitHandle`, který vrací objekt `System.Threading.WaitHandle`. Pomocí tohoto objektu můžete zavolat `WaitAll()` a počkat, až budou ukončeny všechny asynchronní operace. Tato technika je v následujícím příkladě použita u metody `GetEmployees()`, která je třikrát současně vyvolána:

```
protected void cmdMultiple_Click(object sender, System.EventArgs e)
{
    // Zaznamenej čas spuštění.
    DateTime startTime = DateTime.Now;
    EmployeesService proxy = new EmployeesService();
    GetEmployeesDelegate async = new GetEmployeesDelegate(proxy.GetEmployees);
    // Asynchronně vyvolej tři metody.
    IAsyncResult handle1 = async.BeginInvoke(null, null);
    IAsyncResult handle2 = async.BeginInvoke(null, null);
    IAsyncResult handle3 = async.BeginInvoke(null, null);
    // Vytvoř sadu objektů WaitHandle.
    WaitHandle[] waitHandles = {handle1.AsyncWaitHandle,
    handle2.AsyncWaitHandle, handle3.AsyncWaitHandle};
    // Počkej, až se ukončí všechna volání.
    WaitHandle.WaitAll(waitHandles);
    // Teď můžeš získat výsledky.
    DataSet ds1 = async.EndInvoke(handle1);
    DataSet ds2 = async.EndInvoke(handle2);
    DataSet ds3 = async.EndInvoke(handle3);
    // Sluč všechny výsledky do jedné tabulky a zobraz ji.
    DataSet dsMerge = new DataSet();
    dsMerge.Merge(ds1);
    dsMerge.Merge(ds2);
    dsMerge.Merge(ds3);
    GridView1.DataSource = dsMerge;
    GridView1.DataBind();
    // Urči celkový spotřebovaný čas.
    TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
    lblInfo.Text = "Calling three methods took " + timeTaken.TotalSeconds +
    " seconds.";
}
```

Místo použití služby čekání (wait handle) můžete spustit tři asynchronní volání tak, že třikrát zavoláte `BeginInvoke()` a hned poté zavoláte tři metody `EndInvoke()`. V tomto případě bude váš kód čekat (v případě potřeby). Použití služby čekání (wait handle) však kód zpřehlední.

Můžete také použít jednu z extrémně často používaných verzí metody `WaitAll()`, která akceptuje hodnotu prodlevy (timeout). Pokud nebudou volání ukončena do uplynutí doby prodlevy, vznikne výjimka. Obvykle je však lepší se raději opírat o vlastnost `Timeout` z proxy, která ukončí volání, pokud během určené doby nebude obdržena žádná odpověď.

Obsluhu čekání můžete také nařídit, aby blokovala vlákno, dokud nebude ukončeno některé volání metody, a to pomocí statické metody `WaitHandle.WaitAny()` se sadou objektů `WaitHandle`. Metoda `WaitAny()` se vrací, jakmile je ukončeno nejméně jedno asynchronní volání. Tato technika se vám může hodit, jestliže potřebujete zpracovat data z odlišných zdrojů, přičemž pořadí jejich zpracování není důležité. Umožní vám začít zpracovávat výsledky jedné metody ještě před ukončením dalších. Nicméně to zkomplikuje váš kód, protože budete muset testovat vlastnost `IsCompleted` každého objektu `IAsyncResult` a dokud nebudou ukončeny všechny metody, vícekrát volat `WaitAny()` (obvykle ve smyčce).

POZNÁMKA Zapamatujte si, že .NET vám poskytuje ještě další možnosti threadingu prostřednictvím třídy Thread. Potřebujete-li například zavolat sérii webových služeb ve specifickém pořadí jako součást služby, která je dlouhodobě spuštěna na pozadí aplikace Windows, třída Thread nabízí nejlepší řešení. Více informací o multithreadingu naleznete v některé z publikací zabývajících se pokročilými technikami programování pro Windows.

Interaktivní klienti Windows

U klienta Windows je kód threadingu, který použijete, poněkud odlišný. V typickém případě zřejmě budete chtít, aby aplikace mohla neomezeně běžet, zatímco je vykonávána určitá operace. Po ukončení volání pak budete chtít jednoduše obnovit zobrazení s aktualizovanými informacemi.

Podpora tohoto chování je vestavěna do třídy proxy. Abyste lépe porozuměli, jak to funguje, je dobré se podívat na kód třídy proxy. Třída proxy v podstatě pro každou webovou metodu vaší webové služby vkládá dvě metody – synchronní verzi, kterou jste již viděli, a asynchronní verzi, která k metodě přidává příponu Async.

Zde je kód synchronní verze metody GetEmployees(). Atributy pro serializaci XML byly vynechány.

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

A zde je asynchronní verze stejné metody. Všimněte si, že kód v podstatě obsahuje dvě verze metody GetEmployeesAsync(). Jediný rozdíl je v tom, že jedna z nich akceptuje dodatečný parametr userState, kterým může být jakýkoliv objekt, který používáte pro identifikaci volání. Až bude volání později dokončeno, získáte tento objekt v rámci zpětného volání. Parametr userState je užitečný zejména v případě, kdy se současně vyskytne několik asynchronních webových metod.

```
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);
}
```

Základní myšlenkou je, že pro spuštění požadavku můžete vyvolat `GetEmployeesAsync()`. Tato metoda se vrací ihned, ještě předtím, než je požadavek zaslán po síti, a třída proxy čeká na volné vlákno (podobně jako v případě asynchronního delegáta), dokud neobdrží odpověď. Jakmile je odpověď získána a deserializována, .NET spustí událost, která to oznámí vaší aplikaci. Poté můžete získat výsledky.

Aby tento systém fungoval, třída proxy také přidává událost pro každou webovou metodu. Tato událost je odpálena po dokončení asynchronní metody. Zde je dokončovací událost metody `GetEmployees()`:

```
public event GetEmployeesCompletedEventHandler GetEmployeesCompleted;
```

Kód třídy proxy je v tomto případě docela chytrý – zjednodušuje vám život vytvořením vlastního objektu `EventArgs` pro každou webovou metodu. Tento objekt `EventArgs` zpřístupní výsledek metody jako vlastnost `Result`. Třída je deklarována klíčovým slovem `partial`, takže k ní můžete přidat kód v jiném souboru (který není vygenerován automaticky).

```
// Definuje signaturu dokončovací události.
public delegate void GetEmployeesCompletedEventHandler(object sender,
GetEmployeesCompletedEventArgs e);
public partial class GetEmployeesCompletedEventArgs :
System.ComponentModel.AsyncCompletedEventArgs
{
    private object[] results;
    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]));
        }
    }
}
```

Všimněte si, že pokud na straně serveru vznikne chyba, nebude oznámena, dokud se nepokusíte získat vlastnost `Result`, kdy bude `SoapException` zaobalena do `TargetInvocationException`.

POZNÁMKA `Result` není jediná vlastnost, kterou můžete najít ve vlastním objektu `EventArgs`. Budou také přidány parametry `ref` nebo `out`, pokud je vaše webová metoda akceptuje. Po dokončení volání můžete tímto získat modifikované hodnoty všech parametrů `ref` nebo `out`.

Kromě vlastnosti `Result` můžete také použít několik vlastností, které jsou deklarovány v základní třídě `AsyncCompletedEventArgs`. Jsou to `Cancelled` (vrací `true`, jestliže byla operace zrušena předtím, než byla dokon-

čena; s touto funkcí se již brzy seznámíte), Error (vrací objekt výjimky, pokud se během požadavku vyskytla neošetřená výjimka) a UserState (vrací objekt stavu, který jste dodali během volání dané metody).

Chcete-li toto chování vyzkoušet, můžete modifikovat klienta Windows, kterého jsme vytvořili v kapitole 32, aby používal asynchronní volání. Prvním krokem je vytvoření ovladače události pro dokončovací událost, a to ve formě třídy:

```
private void GetEmployeesCompleted(object sender,
GetEmployeesCompletedEventArgs e)
{ ... }
```

Když uživatel klikne na tlačítko Get Employees, kód použije metodu GetEmployeesAsync() ke spuštění procesu. Nejprve však musí k události GetEmployeesCompleted připojit ovladač události. Zde je kód, který k tomu potřebujete:

```
private void cmdGetEmployees_Click(object sender, System.EventArgs e)
{
    // Deaktivuj tlačítko tak, aby bylo povoleno pouze jedno asynchronní
    // volání najednou.
    cmdGetEmployees.Enabled = false;
    EmployeesService proxy = new EmployeesService();
    proxy.GetEmployeesCompleted += new
    GetEmployeesCompletedEventHandler(this.GetEmployeesCompleted);
    proxy.GetEmployeesAsync();
}
```

Když je operace ukončena, třída proxy odpálí událost. Když tuto událost ošetříte, můžete svázat výsledek přímo s mřížkou. Nemusíte si dělat starosti se seřazením vašeho volání k vláknu uživatelského rozhraní, protože toto je automaticky ošetřeno třídou proxy ještě předtím, než je událost vyvolána, což je velmi užitečné.

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

Pokud spustíte tento příklad a kliknete na tlačítko Get Employees, bude toto tlačítko deaktivováno, nicméně aplikace bude pořád reagovat. Můžete přetahovat okno aplikace, měnit jeho velikost či klikat na další tlačítka, čímž se vykoná další kód. Jakmile budou získány výsledky, bude spuštěno zpětné volání a DataGridView bude automaticky obnoven.

Je možné použít ještě jeden trik. Třída proxy obsahuje vestavěnou podporu pro stornování. Abyste ji mohli používat, musíte při vyvolání asynchronní verze metody třídy proxy doplnit objekt stavu. Pak jednoduše

zavoláte metodu `CancelAsync()` třídy proxy a doplníte stejný objekt stavu. Zbytek procesu je už řízen automaticky.

Pokud chcete model stornování implementovat do již existující aplikace, musíte nejprve deklarovat třídu proxy na úrovni formuláře, aby byla dostupná všem vašim ovladačům událostí:

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

Poté, když se formulář načte, přidejte ovladač události tak, aby byl připojen pouze jednou:

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

V tomto případě není objekt stavu, který používáte, důležitý, protože bude najednou probíhat pouze jediná operace. Jestliže provádíte současně více operací, má smysl vygenerovat nový GUID pro sledování jednotlivých operací.

GUID nejprve deklaruje ve třídě formuláře:

```
private Guid requestID;
```

Poté ho vygenerujete a doplníte do asynchronního volání webové metody:

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

Teď vám stačí při volání metody `CancelAsync()` použít stejný GUID. Zde je kód pro stornovací tlačítko:

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

Zde se musíme zmínit o jedné důležité věci. Jakmile zavoláte `CancelAsync()`, je odpálena dokončená událost. To dává smysl, protože byla ukončena dlouho běžící operace (ačkoliv na základě programové intervence) a vy možná budete muset aktualizovat uživatelské rozhraní. Je samozřejmé, že se nemůžete dostat k výsledku metody v rámci dokončené události, protože kód byl přerušen. Abyste předešli nějaké chybě, je potřeba explicitně otestovat stornování, jak vidíte zde:

```
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.");
        }
    }
}

```

Asynchronní služby

Doposud jste viděli několik příkladů, které umožňovaly klientům asynchronně volat webové služby. Ve všech těchto příkladech ovšem webová metoda běží od začátku do konce synchronně. Co kdybyste chtěli dosáhnout jiného chování, které by klientovi umožňovalo spustit dlouho běžící proces, a poté se později připojit a vyzvednout si výsledky?

Bohužel .NET tento model nepodporuje přímo. Problém tkví částečně v tom, že veškerá komunikace webové služby musí být inicializována klientem. Webový server v současné době nemůže nijak inicializovat zpětné volání klienta, aby mu oznámil, že úkol byl splněn. Přestože jsou vyvíjeny standardy, které mají tuto mezeru zaplnit, není pravděpodobné, že by tato řešení začala být běžně používána, a to kvůli povaze architektury webu. Mnoho klientů se připojuje přes proxy servery nebo firewally, které nepovolí příchozí spojení, nebo které skrývají informace o umístění, jako je například IP adresa. Výsledkem je, že klient musí inicializovat každé spojení.

Určitá inovativní řešení vám pochopitelně mohou poskytnout část požadované funkcionality. Například klient se může připojovat v nějakých časových intervalech, prozkoumat server a zjistit, zdali je úkol dokončen. Toto chování můžete použít například tehdy, jestliže webový server provádí mimořádně časově náročné úkoly, jako je například realizace komplexní grafiky. Pořád vám však chybí jeden dílek do skládačky. V těchto situacích potřebujete nějaký způsob, kterým klient spustí webovou metodu, aniž by musel čekat, až bude metoda vykonána. ASP.NET toto umožňuje prostřednictvím tzv. jednosměrných metod (one-way methods).

Pomocí těchto metod (které se také nazývají jako "odpal a zapomeň", "fire-and-forget") klient odešle zprávu o požadavku, nicméně webová služba klientovi vůbec neodpoví. To znamená, že webová metoda se ihned vrátí a ukončí spojení. Klient nemusí čekat. Jednosměrné metody ovšem mají i několik nevýhod. Webová metoda nemůže poskytovat návratovou hodnotu nebo používat parametry ref a out. Dále – pokud v rámci webové metody vznikne neošetřená výjimka, nebude zaslána zpět klientovi.

Chcete-li vytvořit XML metodu webové služby ve stylu fire-and-forget, musíte u příslušné webové metody použít atribut SoapDocumentMethod a nastavit vlastnost OneWay na hodnotu true:

```

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

```

Je to kvůli tomu, že klient nemusí podniknout žádné speciální kroky k asynchronnímu vyvolání jednosměrné metody. Metoda bude totiž ihned vrácena.

Samozřejmě – je možné, že jednosměrné metody nebudete chtít vůbec použít. Jejich nejdůležitějším omezením je, že nemohou vrátit žádné informace. Například běžný asynchronní způsob chování na straně serveru je ten, že poté, co klient předloží požadavek, server vrátí klientovi nějaký typ unikátního, automaticky generovaného lístku. Klient pak může tento lístek předložit dalším metodám, aby prověřil stav nebo získal výsledky. U jednosměrné metody není možné vrátit lístek klientovi, a ani není možné mu oznámit výskyt nějaké chyby.

Dalším problémem je, že ASP.NET u jednosměrné metody stále používá tzv. pracovní vlákna (worker threads). Pokud je úkol extrémně náročný, a pokud současně probíhá zpracování jiných úkolů, další klienti možná nebudou schopni zadat nové požadavky, což není ideální.

Jediným praktickým způsobem, jak zpracovat dlouhodobě běžící asynchronní úkoly u webové stránky s hustým provozem, je zkombinovat webové služby s další technologií .NET – s tzv. vzdáleným voláním (remoting). Například si vytvoříte obyčejnou synchronní webovou metodu, která vrací lístek, a poté pomocí vzdáleného volání zavoláte metodu serverové komponenty. Komponenta vzdáleného volání (remoting component) se pak může asynchronně pustit do zpracování úkolu. Tento styl použití webové služby je mnohem charakterističtější znakem distribuovaného návrhu. Chcete-li se dozvědět více, musíte si prostudovat některou z publikací věnovaných distribuovanému programování, například knihu Microsoft .NET Distributed Applications (Microsoft Press, 2003) nebo publikaci zabývající se vzdáleným voláním .NET, například knihu Advanced .NET Remoting (Apress, 2002). Pokud takovou flexibilitu nepotřebujete, měli byste se jí raději úplně vyhnout, protože její implementace je značně složitá.

Zabezpečení webových služeb

V ideálním světě byste mohli webovou službu pojmout jako knihovnu třídy s určitými schopnostmi a nemuseli byste si dělat starosti se zajišťováním autentizace uživatelů nebo s logikou zabezpečení. Vytváříte-li však webové služby, které jsou založeny na přihlašování nebo které jsou spojeny s mikroplatbami, musíte specifikovat, kdo je bude používat. A dokonce i v případě, kdy neprodáváte schopnosti vaší webové služby skupině nedočkavých webových vývojářů, i tehdy potřebujete použít autentizaci, abyste ochránili citlivá data a zabránili zlomyslným uživatelům v přístupu, zejména tehdy, pokud je vaše webová služba vystavena na Internetu.

K ochraně webových služeb můžete použít některou z technik používaných k ochraně webových stránek. Můžete například použít IIS s podporou SSL (své klienty jednoduše nasměrujte na adresu webové služby, která začíná předponou https://). IIS můžete rovněž použít pro autentizaci Windows, ačkoliv je pravda, že musíte provést ještě několik dalších kroků, o kterých budeme hovořit v následující sekci. Další možností je vytvořit si svůj vlastní systém autentizace pomocí SOAP záhlaví.

Autentizace Windows

U webových služeb funguje autentizace Windows velmi podobně jako u webové stránky. Rozdíl je v tom, že webová služba je vykonávána jinou aplikací, nikoliv přímo prohlížečem. Z tohoto důvodu zde není k dispozici žádný vestavěný způsob, jak vyzvat uživatele, aby zadal své uživatelské jméno a heslo. Tyto informace totiž musí doplnit samotná aplikace, která používá webovou službu. Tato aplikace může přečíst tyto informace z konfiguračního souboru, z databáze, nebo může vyzvat uživatele k doplnění těchto informací ještě před kontaktováním webové služby.

Představte si například následující webovou službu, která poskytuje jedinou metodu TestAuthenticated(). Tato metoda kontroluje, zdali je uživatel autentizován. Pokud autentizován je, vrací uživatelské jméno jako řetězec ve tvaru NázevDomény\UživatelskéJméno nebo NázevPočítač\UživatelskéJméno.

```
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;
    }
}
}

```

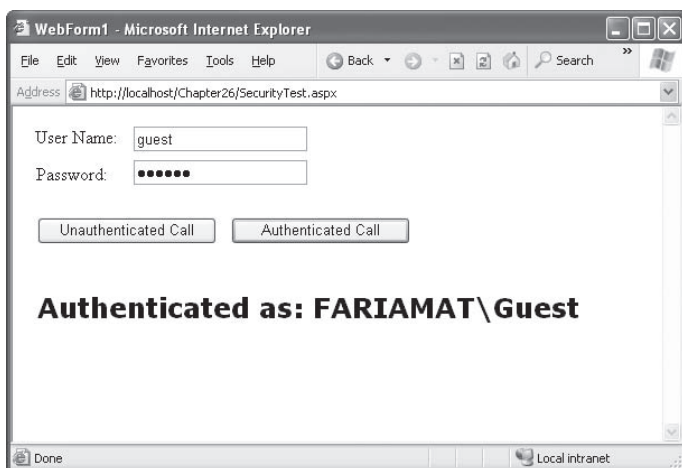
Webová služba může také prozkoumat roli členství, ačkoli tato webová služba tento krok neprovádí. Pokud chce uživatel předložit této službě své přihlašovací doklady, musí klient modifikovat vlastnost `NetworkCredential` třídy `proxy`. Máte dvě následující volby:

- Můžete vytvořit nový objekt `NetworkCredential` a připojit jej k vlastnosti `NetworkCredential` objektu `proxy`. Po vytvoření objektu `NetworkCredential` musíte specifikovat uživatelské jméno a heslo, které chcete používat. Tento přístup funguje u všech typů autentizace Windows.
- Pokud webová služba používá integrovanou autentizaci Windows, můžete automaticky předložit přihlašovací doklady aktuálního uživatele pomocí statické vlastnosti `DefaultCredentials` z třídy `CredentialCache` a aplikovat je pro vlastnost `NetworkCredential` objektu `proxy`.

Obě třídy, `CredentialCache` a `NetworkCredential`, se nacházejí ve jmenném prostoru `System.Net`, takže předtím než budete pokračovat, měli byste tento jmenný prostor importovat:

```
using System.Net;
```

Následující kód představuje webovou stránku se dvěma textovými rámečky a dvěma tlačítky (viz obrázek 34-2). Jedno tlačítko provádí neautentizované volání, druhé tlačítko odesílá uživatelské jméno a heslo, které byly vloženy do dvou textových polí formuláře.



Obrázek 34-2. Úspěšná autentizace prostřednictvím webové služby.

Neautentizované volání selže, pokud jste zablokovali anonymní uživatele. V opačném případě bude neautentizované volání úspěšné, nicméně metoda `TestAuthenticated()` vrátí řetězec informující o tom, že autentizace

nebyla provedena. Autentizované volání bude úspěšné vždy, když vložíte přihlašovací doklady, které korespondují s platným uživatelem na webovém serveru.

Zde uvádíme kompletní kód webové stránky:

```
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();
        // Doplně některé ověřovací informace uživatele pro webovou službu.
        NetworkCredential credentials = new NetworkCredential(
            txtUserName.Text, txtPassword.Text);
        proxy.Credentials = credentials;
        lblInfo.Text = proxy.TestAuthenticated();
    }
}
```

Pokud si to chcete vyzkoušet, přidejte následující element <location> do souboru web.config, čímž omezíte přístup k webové službě 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>
```

Pokud chcete použít přihlašovací doklady aktuálně přihlášeného účtu v integrované autentizaci Windows, měli byste použít tento kód:

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

V tomto příkladě (podobně jako u všech webových stránek) bude aktuálním uživatelským účtem ten účet, který používá ASP.NET, a nikoliv účet vzdáleného uživatele, který požádá o webovou stránku. Pokud stejnou techniku používáte v aplikaci Windows, předložte informace o účtu uživatele, který spouští danou aplikaci.

Vlastní autentizace založená na lístcích

Autentizace Windows je dobrým řešením pro webové služby, pokud máte malý okruh uživatelů, kteří již mají účty Windows. Ovšem tento typ autentizace už tak dobře nefunguje u rozsáhlých veřejných webových služeb. Když pracujete s webovými stránkami ASP.NET, obvykle používáte formulářovou autentizaci, abyste zaplnili případné mezery. Formulářová autentizace však nebude fungovat v případě webové služby, protože webová služba nemůže uživatele nasměrovat na přihlašovací webovou stránku. Je potřeba si uvědomit, že pro přístup k webové službě nemusí být použit pouze prohlížeč, ale také nějaká aplikace Windows nebo dokonce automatizovaná služba Windows. Formulářová autentizace je navíc založena na cookies, což představuje pro webové služby zbytečné omezení. Webové služby totiž mohou používat protokoly, které cookies nepodporují, nebo mohou být použiti klienti, kteří je neočekávají.

Obvyklým řešením je vytvořit si vlastní systém autentizace. V tomto modelu musí uživatelé, kteří se chtějí přihlásit, zavolat specifickou webovou metodu webové služby a poté dodat své přihlašovací doklady (například kombinaci uživatelského jména a hesla). Metoda pro přihlášení zaregistruje uživatelskou relaci a vytvoří nový, jedinečný lístek. Od tohoto momentu se uživatel může opakovaně připojovat k webové službě tím, že jakékoli další metodě předloží tento lístek (ticket).

Správně navržený systém založený na lístku má řadu výhod. Podobně jako formulářová autentizace poskytuje absolutní flexibilitu. Optimalizuje také výkon a zaručuje rozšiřitelnost, protože lístek je možné cachovat. U dalších požadavků si můžete ověřit pouze lístek a nemusíte autentizovat uživatele s použitím informací v databázi. A nakonec vám umožňuje využít výhody SOAP záhlaví, které zajišťují, aby proces řízení lístků byl přehledný, stejně jako autorizace pro klienta.

V případě ASP.NET 2.0 je možné vlastní autentizaci zjednodušit v rámci webové služby. Ačkoliv pořád zůstává vaším úkolem přenášet přihlašovací doklady uživatele a sledovat, kdo se přihlásil prostřednictvím vydaných a ověřených lístků, pro řízení autentizace a autorizace už můžete použít schopnosti členství a rolí, které byly probrány v kapitolách 21 a 23. V následujících sekcích vám ukážeme, jak vytvořit vlastní systém autentizace, který bude založen na lístcích, a který bude využívat schopnosti členství a rolí.

Sledování identity uživatele

Pokud chcete použít vlastní bezpečnostní systém, musíte se prvně rozhodnout, jaké informace, které budou specifické pro konkrétního uživatele, chcete cachovat. Potřebujete vytvořit vlastní třídu, která bude tyto informace reprezentovat. Tato třída může například obsahovat informace o uživateli (jméno, e-mailová adresa atd.) a informace o oprávněních uživatele. Měla by také obsahovat lístek.

Zde je základní příklad, ve kterém se ukládá uživatelské jméno a lístek:

```
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;
        // Vytvoř lístek GUID.
        Ticket = Guid.NewGuid().ToString();
    }
}

```

POZNÁMKA Pravděpodobně jste si již všimli, že tato třída identity neimplementuje `IIdentity`. Je to proto, že tento postup vám neumožňuje se zapojit do bezpečnostního modelu .NET stejným způsobem, jako je tomu v případě vlastní autentizace webové stránky. Problém v podstatě spočívá v tom, že autentizaci potřebujete provést až poté, co byl vytvořen objekt `User`. Tento problém nelze obejít použitím třídy `global.asax`, protože ovladače událostí aplikace nebudou mít přístup k parametrům webové metody a k záhlaví SOAP, které potřebujete pro provedení autentizace a autorizace.

Jakmile máte pro uživatele vytvořenou třídu identity, potřebujete vytvořit SOAP záhlaví. Toto záhlaví sleduje jedinou informaci – lístek uživatele. Protože lístek je náhodně vygenerované GUID, je prakticky nemožné, aby zlomyslný uživatel "uhodl", jaký lístek byl vydán nějakému jinému uživateli.

```

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

```

Poté musíte do vaší webové služby přidat členskou proměnnou pro `TicketHeader`:

```

public class SoapSecurityService : WebService
{
    public TicketHeader Ticket;
}

```

```
...
}
```

Autentizace uživatele

Dalším krokem je vytvoření vyhrazené webové metody, která přihlásí uživatele. Uživatel musí této metodě předložit své přihlašovací doklady (například uživatelské jméno a heslo). Metoda poté získá informace o uživateli, vytvoří objekt `TicketIdentity` a vydá lístek.

V tomto příkladu zkontroluje webová metoda `Login()` přihlašovací doklady uživatele prostřednictvím statické metody `Membership.ValidateUser()`. Společně s informacemi o uživateli je vytvořen nový lístek, který je uložen ve specifické složce uživatele v kolekci `Application`. Současně s lístkem je vydáno nové SOAP záhlaví, takže uživatel má přístup i k dalším metodám.

Zde uvádíme kompletní kód metody `Login()`:

```
[WebMethod()]
[SoapHeader("Ticket", Direction = SoapHeaderDirection.Out)]
public void Login(string userName, string password)
{
    if (Membership.ValidateUser(username, password))
    {
        // Vytvoř nový lístek.
        TicketIdentity ticket = new TicketIdentity(username);
        // Přidej tento ticket ke stavu Application.
        Application[ticket.Ticket] = ticket;
        // Vytvoř SOAP záhlaví.
        Ticket = new TicketHeader(ticket.Ticket);
    }
    else
    {
        throw new SecurityException("Invalid credentials.");
    }
}
```

Povšimněte si, že v tomto případě je objekt `TicketIdentity` uložen v kolekci `Application`, která je společná pro všechny uživatele. Nemusíte si však dělat starosti s tím, že by lístek jednoho uživatele mohl být přepsán jiným lístkem. Je to proto, že lístky jsou indexovány použitím GUID. Každý uživatel má samostatný lístek GUID a tím i samostatný slot v kolekci `Application`.

Kolekce `Application` má ovšem určitá omezení – nepodporuje webové farmy a nehodí se pro velký počet uživatelů. Lístky se rovněž ztratí při restartu webové aplikace. Pro vylepšení tohoto řešení můžete informace ukládat na dvou místech – do objektu `Cache` a do koncové databáze. Takto může váš kód nejprve zkontrolovat objekt `Cache`, a pokud bude nalezen odpovídající `TicketIdentity`, nebude již vyžadováno volání databáze. Pokud však `TicketIdentity` nebude přítomen, příslušné informace mohou být získány z databáze. Je důležité pochopit, že toto vylepšení používá stejné SOAP záhlaví s lístkem a stejný objekt `TicketIdentity`. Jediný rozdíl je ve způsobu ukládání `TicketIdentity` a jeho získávání mezi jednotlivými požadavky.

Autorizace uživatele

Jakmile máte vytvořenou metodu Login(), má smysl vytvořit soukromou metodu, která může být zavolána za účelem ověření, zdali je uživatel přítomen. Tuto metodu pak můžete zavolat z jiných webových metod vaší webové služby.

Následující metoda AuthorizeUser() kontroluje odpovídající lístek a vrací TicketIdentity, pokud je nalezen. Pokud nalezen není, vznikne výjimka, která bude vrácena klientovi.

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

Následující přetížená verze AuthorizeUser() navíc ověřuje, zdali má uživatel lístek, a zdali je členem nějaké specifické role. Kontrola rolí je poskytována nástrojem ASP.NET pro řízení rolí.

```
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;
    }
}
```

Pomocí těchto dvou pomocných metod můžete vytvořit další metody webové služby, které testují uživatelská oprávnění před provedením určitých úloh, nebo před vrácením privilegovaných informací.

Testování systému autentizace SOAP

Nyní potřebujete vytvořit nějakou testovací webovou metodu, která bude používat metodu AuthorizeUser() za účelem ověření, zdali má uživatel potřebná oprávnění. Zde je uveden příklad, který zkontroluje, zdali je klient administrátorem. Kontrola se provede předtím, než bude uživateli umožněno získat sadu dat se seznamem zaměstnanců:

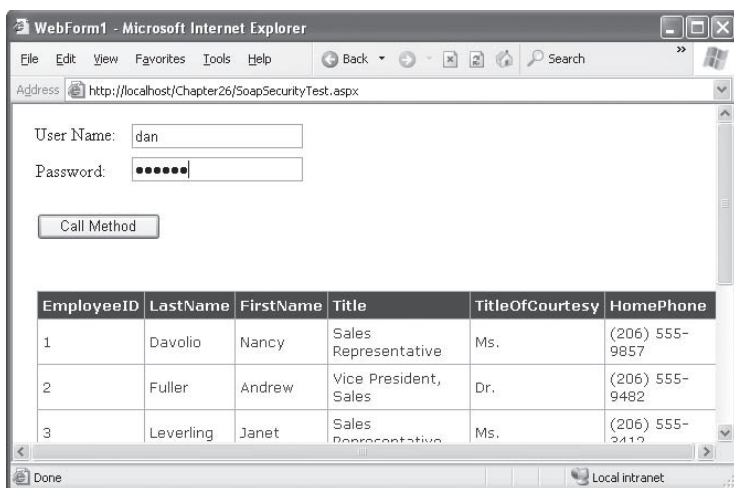
```
[WebMethod()]
```

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

Aby bylo možné snadněji nastavit tento test, vzorový kód pro tuto kapitolu obsahuje webovou metodu `CreateTestUser()`, která vygeneruje specifického uživatele a zařadí jej do role `Administrator`:

```
[WebMethod()]
public void CreateTestUser(string username, string password)
{
    if (Membership.GetUser(username) != null)
    {
        Membership.DeleteUser(username);
    }
    Membership.CreateUser(username, password);
    string role = "Administrator";
    if (!Roles.RoleExists(role))
    {
        Roles.CreateRole(role);
    }
    Roles.AddUserToRole(username, role);
}
```

Nyní můžete vytvořit klienta, který to všechno otestuje. V tomto případě poskytuje webová stránka dvě testová pole, do nichž uživatel doplní uživatelské jméno a heslo (viz obrázek 34-3). Tyto informace jsou pak předány metodě `Login()`, a poté je zavolána metoda `GetEmployees()`, která získá data. Tato metoda bude úspěšná u uživatele, který má roli administrátora. U jiných uživatelů však selže.



Obrázek 34-3. Testování webové metody, která používá autorizaci založenou na lístcích.

Zde uvádíme kód webové stránky:

```
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;
    }
}
```

Nejlepší na tom je, že klient si nemusí být vědom používání nějakého lístku. Je to proto, že metoda Login() vydá lístek a třída proxy jej automaticky udržuje. Dokud klient používá stejnou instanci třídy proxy, bude automaticky odesílána stejná hodnota lístku a uživatel bude autentizován.

Tento systém autentizace však můžete ještě podstatně vylepšit. Můžete například chtít pomocí TicketIdentity zaznamenat další údaje, jako například čas, kdy byl lístek vytvořen, a kdy byl naposledy použit, nebo zaznamenat síťovou adresu uživatele, který vlastní lístek. Tímto způsobem můžete do metody AuthorizeUser() zařadit další ověřovací způsoby. Můžete například zařadit nějaký časový limit (timeout) pro jejich používání, nebo můžete lístek odmítnout, pokud se změnila IP adresa klienta atd.

Rozšíření SOAP

Webové služby ASP.NET poskytují vysokoúrovňový přístup k SOAP. Jak jste už měli možnost vidět, nemusíte o SOAP vědět moc informací, abyste mohli vytvářet a volat webové služby. Pokud jste však vývojářem s dobrými znalostmi o SOAP a chcete se zabývat nízkoúrovňovým přístupem ke zprávám SOAP, .NET Framework vám to umožní. V této sekci si ukážeme, jak zachytávat SOAP zprávy a jak s nimi manipulovat.

POZNÁMKA Dokonce i tehdy, když nechcete manipulovat se SOAP zprávami přímo, stojí za to se dozvědět více informací o rozšířeních SOAP, protože jsou součástí infrastruktury, která podporuje WSE, o níž se zmíníme později v sekci "Vylepšení webových služeb".

Rozšíření SOAP představují mechanismus rozšiřitelnosti. Umožňují vývojářům třetí strany vytvářet komponenty, které se zapojují do modelu webové služby a poskytují další služby. Můžete například vytvořit rozšíření SOAP, které selektivně zašifruje nebo zkomprimuje části SOAP zprávy ještě předtím, než je odeslána z klienta. Na serveru samozřejmě potřebujete spustit odpovídající rozšíření SOAP pro dešifrování a dekomprimaci zprávy poté, co byla přijata, ale ještě před její deserializací.

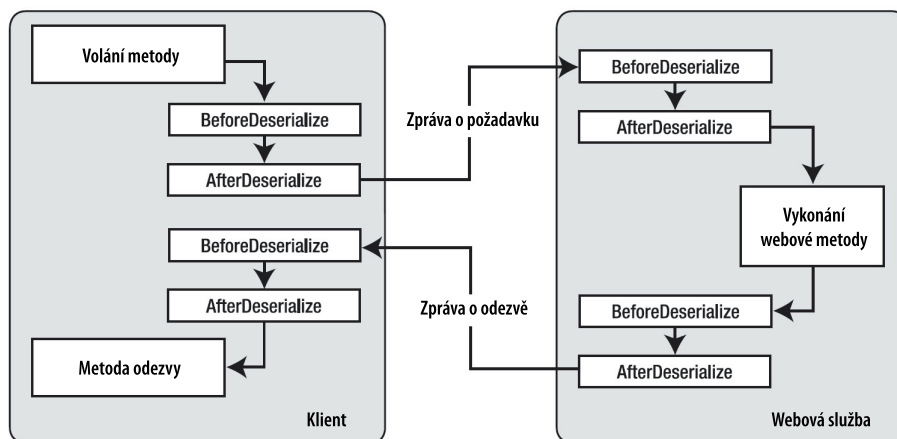
Chcete-li vytvořit rozšíření SOAP, potřebujete vytvořit třídu odvozenou ze třídy System.Web.Services.Protocols.SoapExtension. Třída SoapExtension obsahuje metodu ProcessMessage(), která je automaticky spuštěna,

jakmile SOAP zpráva prochází jednotlivými etapami. Spustíte-li například rozšíření SOAP na webovém serveru, proběhnou následující čtyři etapy:

- **SoapMessageStage.BeforeDeserialize.** Vznikne ihned poté, co webový server obdrží SOAP zprávu o požadavku.
- **SoapMessageStage.AfterDeserialize.** Vznikne poté, co je nezpracovaná (raw) SOAP zpráva převedena na datové typy .NET, ale ještě před spuštěním kódu webové metody.
- **SoapMessageStage.BeforeSerialize.** Vznikne poté, co je spuštěn kód webové metody, ale ještě předtím, než je návratová hodnota převedena do SOAP zprávy.
- **SoapMessageStage.AfterSerialize.** Vznikne poté, co jsou návratová data serializována do SOAP zprávy o odpovědi, ale ještě předtím, než jsou odeslána klientské aplikaci.

V každé etapě můžete získat jiné informace o zprávě SOAP. V krocích BeforeDeserialize nebo AfterSerialize můžete získat úplný text zprávy SOAP.

Rozšíření SOAP rovněž můžete implementovat u klienta. V tom případě proběhnou stejné čtyři etapy. Zpráva je získána, deserializována a zpracována třídou proxy (nikoliv webovou službou). Obrázek 34-4 znázorňuje celý proces.



Obrázek 34-4. Zpracování SOAP u klienta a na serveru.

Vytvoření rozšíření SOAP, které je určeno pro prodej někomu jinému, není vůbec jednoduché. Narazíte zde na množství závažných problémů:

- Rozšíření SOAP musí být v mnoha případech vykonávána jak u klienta, tak i na serveru. To znamená, že si musíte dělat starosti s distribucí a řízením další komponenty, která může být v rámci velkého distribuovaného systému neobyčejně komplexní.
- Rozšíření SOAP způsobí, že vaše webové služby budou méně kompatibilní. Klienti třetí strany si možná neuvědomí, že si potřebují nainstalovat a spustit rozšíření SOAP, aby mohli používat vaši webovou službu, protože tato informace se nenachází ve WSDL dokumentu. Pokud klienti používají jiné platformy než .NET, nebudou moci používat třídu SoapExtension, kterou jste vytvořili a vy pravděpodobně budete muset najít jiný způsob, jak vytvořit toho rozšíření SOAP pro jiné platformy, což nemusí být zrovna jednoduché.

- Internet se hemží mnoha příklady rozšíření SOAP, které používají chabé a nedostatečně zabezpečené šifrování. Jejich vady spočívají např. v nejisté práci s klíči, v neschopnosti provádět výměny klíčů nebo v pomalém výkonu, protože se spoléhají na asymetrické, a nikoliv na symetrické šifrování. Mnohem lepší je použít IIS s protokolem SSL, který je neprůstřelný.

Ve většině případů je vytvoření rozšíření SOAP úlohou, kterou je nejlépe přenechat architektům v Microsoftu, kteří jsou zaměřeni na problémy vývoje a implementace na úrovni podniku. Microsoft ve skutečnosti implementuje několik novějších (a tudíž nezralejších) standardů webových služeb s použitím rozšíření SOAP, které je založeno na volně stažitelné WSE komponenty, o které se dozvíte více v sekci "Vylepšení webových služeb".

POZNÁMKA Rozšíření SOAP fungují pouze přes SOAP. Když testujete webovou metodu na testovací stránce prohlížeče, nebudou vyvolány. SOAP zprávy rovněž nebudou fungovat, pokud nastavíte atribut `BufferResponse` z atributu `WebMethod` na `false`. V takovém případě nemohou rozšíření SOAP pracovat se SOAP daty, protože ASP.NET je začne hned posílat – tedy ještě předtím, než jsou kompletně vygenerována.

Vytvoření rozšíření SOAP

V následujícím příkladě si ukážeme vzorové rozšíření SOAP, které bude zaznamenávat zprávy SOAP do protokolu událostí Windows. Vady SOAP budou do tohoto protokolu událostí zapsány jako chyby.

TIP Někteří vývojáři používají rozšíření SOAP pro účely trasování. Jak jste se však už dozvěděli v kapitole 33, není to nutné, protože trasovací utilita, která je součástí Microsoft SOAP Toolkit, je mnohem pohodlnější. Avšak hlavní výhodou rozšíření SOAP určeného pro trasování je ta, že jej můžete použít pro permanentní zachytávání a ukládání zpráv SOAP i v případě, kdy trasovací utilita není spuštěna. K tomu, abyste mohli zprávy směřovat přes trasovací utilitu, nemusíte v kódu klienta měnit port.

Všechna rozšíření SOAP se skládají ze dvou součástí – z vlastní třídy, odvozené ze `System.Web.Services.Protocols.SoapExtension`, a z vlastního atributu, který použijete u webové metody, a který indikuje, že vaše rozšíření SOAP by mělo být použito.

Atribut SoapExtension

Atribut `SoapExtension` vám umožňuje spojit specifická rozšíření SOAP s metodami webové třídy. Když vytváříte atribut `SoapExtension`, odvozujete ze `System.Web.Services.Protocols.SoapExtensionAttribute`, jak vidíte zde:

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

Všimněte si, že třída atributu obsahuje další atribut – `AttributeUsage`. Ten označuje, kde můžete váš vlastní atribut použít. Atributy rozšíření SOAP se vždy používají pro jednotlivé deklarace metod (podobně jako atributy `SoapHeader` a `WebMethod`). Měli byste tudíž používat `AttributeTargets.Method` pro prevenci toho, aby

atributy rozšíření SOAP nebyly použity na jinou konstrukci kódu (jako například na deklaraci třídy). Atribut `AttributeUsage` byste měli použít pokaždé, když potřebujete vytvořit vlastní atribut – není totiž omezen pouze na scénáře webových služeb.

Každý atribut `SoapExtension` musí překrýt (override) dvě abstraktní vlastnosti – `Priority` a `ExtensionType`. Vlastnost `Priority` nastavuje pořadí, v jakém pracují rozšíření SOAP, pokud máte nakonfigurováno více rozšíření. Tato vlastnost však není nutná pro jednodušší rozšíření, jako je například rozšíření uvedené v tomto příkladě. Vlastnost `ExtensionType` vrací objekt `Type`, který reprezentuje vaši vlastní třídu `SoapExtension` a umožňuje, aby .NET mohlo připojit vaše rozšíření SOAP k metodě. V tomto příkladu je použit `SoapLog` jako název třídy rozšíření SOAP.

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

K vašemu rozšíření SOAP můžete přidat vlastnosti, které představují několik bitů inicializačních informací navíc. V následujícím příkladě je přidána vlastnost `Name`, která uchovává zdrojový řetězec, který bude použit při zapisování položek do protokolu událostí (tzv. event log), a vlastnost `Level`, která specifikuje, které typy zpráv budou zaznamenány. Pokud má `Level` (úroveň) hodnotu 1, rozšíření `SoapLog` zaznamená pouze chybové zprávy. Má-li `Level` hodnotu 2 a větší, rozšíření `SoapLog` запиše všechny typy zpráv. Má-li `Level` hodnotu 3 a více, rozšíření `SoapLog` přidá ke každé zprávě další informaci, která zaznamená, v jakém stupni byla položka protokolu zapsána.

```
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; }
}
```

Tento vlastní atribut můžete nyní použít u webové metody a nastavit vlastnosti `Name` a `Level`. Zde uvádíme příklad, který používá zdrojové jméno protokolu `EmployeesService.GetEmployeesCount` s hodnotou `Level 3`:

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

```
public int GetEmployeesCount()  
{ ... }
```

Když je nyní zavolána metoda `GetEmployeesCount()` se zprávou SOAP, je vytvořena, inicializována a vykonána třída `SoapLog`. Tato třída možnost zpracovat zprávu SOAP o požadavku ještě předtím, než ji obdrží metoda `GetEmployeesCount()` a má také možnost zpracovat zprávu SOAP o odpovědi ihned poté, co metoda `GetEmployeesCount()` vrátí výsledek.

SoapExtension

Třída `SoapExtension`, kterou budeme používat pro protokolování zpráv, je poměrně dlouhá, ačkoliv většina kódu je standardní a je používán každým SOAP rozšířením. Prozkoumáme ji kousek po kousku.

První věcí, kterou byste si měli všimnout je ta, že třída je povinně odvozena z abstraktní základní třídy `SoapExtension`. Třída `SoapExtension` poskytuje mnoho metod, které potřebujete překrýt, včetně těchto tří následujících:

- **GetInitializer() a Initialize().** Tyto metody předávají počáteční informace pro rozšíření SOAP poté, když bylo rozšíření poprvé vytvořeno.
- **ProcessMessage().** Zde se odehrává vlastní zpracování, které umožňuje vašemu rozšíření se podívat a modifikovat nezpracované SOAP.
- **ChainStream().** Tato metoda je základním článkem infrastruktury, který by měla poskytovat každá webová služba. Umožňuje vám získat přístup k proudu SOAP, bez přerušení jiných rozšíření.

Definice této třídy je uvedena zde:

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

ASP.NET volá metodu `GetInitializer()` poté, co je vaše rozšíření poprvé použito v nějaké webové metodě. To vám dává možnost inicializovat a uložit některá data, která budou použita při zpracování SOAP zpráv. Tyto informace uložíte tak, že je předáte zpět jako návratovou hodnotu metody `GetInitializer()`.

Když je zavolána metoda `GetInitializer()`, získáte důležitou informaci – vlastní atribut, který byl použit u odpovídající webové metody. V případě `SoapLog` je to instance třídy `SoapLogAttribute`, která poskytuje vlastnost `Name` a `Level`. Pokud chcete tyto informace uložit pro opakované použití v budoucnosti, můžete vrátit atribut metody `GetInitializer()`, jak vidíte zde:

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

Metoda `GetInitializer()` má v podstatě dvě verze. Je vyvolána pouze jedna, v závislosti na tom, zdali je rozšíření SOAP nakonfigurováno prostřednictvím atributu (jako v tomto příkladě), nebo prostřednictvím konfiguračního souboru. Pokud byl použit konfigurační soubor, rozšíření SOAP automaticky běží pro každou metodu každé webové služby.

A dokonce i tedy, když neplánujete pro inicializaci rozšíření SOAP použít konfigurační soubor, musíte implementovat druhou verzi `GetInitializer()`. V tomto případě má smysl vrátit novou instanci `SoapLogAttribute`, takže standardní nastavení `Name` a `Level` budou k dispozici později:

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

Metoda `GetInitializer()` je zavolána pouze při prvním vykonání rozšíření SOAP pro danou metody. Avšak pokaždé, když je tato metoda vyvolána, je spuštěna metoda `Initialize()`. Pokud jste vrátili objekt metody `GetInitializer()`, ASP.NET poskytne tento objekt metodě `Initialize()` vždy, když je zavolána. Rozšíření `SoapLog` je vhodným místem pro extrahování informací `Name` a `Level` a jejich uložení do členských proměnných, takže budou k dispozici až do konce zpracování SOAP. (Tyto informace není možné ukládat do metody `GetInitialize()`, protože tato metoda nebude zavolána při každém vykonání rozšíření SOAP.)

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

Hlavním tahounem tohoto rozšíření je metoda `ProcessMessage()`, kterou ASP.NET volá v různých etapách serializačního procesu. Objekt `SoapMessage` je předán metodě `ProcessMessage()` a vy můžete tuto metodu prozkoumat a získat informace o zprávě, jako třeba text zprávy a etapu, ve které se nachází. Rozšíření `SoapLog` přečte celou zprávu pouze během `AfterSerialize` a `BeforeDeserialize`, protože pouze v těchto etapách můžete získat kompletní XML zprávu SOAP. Pokud je však úroveň (tj. `Level`) 3 a více, základní položka protokolu bude vytvořena během `BeforeSerialize` a `AfterDeserialize`, přičemž pouze zaznamenaná název etapy.

Zde uvádíme úplný kód `ProcessMessage()`:

```
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;
    }
}

```

Metoda `ProcessMessage()` neobsahuje kód protokolování. Místo toho volá jiné soukromé metody jako například `WriteLogLog()`, `LogOutputMessage()` a `LogInputMessage()`. `WriteToLog()` představuje konečný bod, ve kterém je vytvořena položka protokolu s použitím třídy `System.Diagnostics.EventLog`. V případě potřeby tento kód vytvoří nový protokol události a nový zdroj protokolu pomocí názvu, který byl nastaven v rámci vlastnosti `Name` vašeho vlastního atributu rozšíření.

Zde je uveden kompletní kód metody `WriteToLog()`:

```

private void WriteToLog(string message, EventLogEntryType type)
{
    // Vytvoř nový protokol s názvem Web Service Log, se zdrojem událostí
    // který je specifikován pomocí atributu.
    EventLog log;
    if (!EventLog.SourceExists(name))
        EventLog.CreateEventSource(name, "Web Service Log");
    log = new EventLog();
    log.Source = name;
    log.WriteEntry(message, type);
}

```

Jestliže se SOAP zpráva nachází v `BeforeSerialize` nebo `AfterDeserialize`, je přímo zavolána metoda `WriteToLog()` a je zapsán název etapy. Jestliže se SOAP zpráva nachází v `AfterSerialize` nebo `BeforeDeserialize`, budete muset k jejímu získání provést o něco více úkonů.

Než budete moci tyto metody vybudovat, potřebujete ještě něco – metodu `CopyStream()`. To proto, že XML zprávy SOAP je obsažen v proudu. Proud obsahuje ukazatel, který indikuje aktuální pozici v proudu. Problémem je, že při čtení zprávy z proudu (například kvůli jejich protokolování) posouváte ukazatelem. To znamená, že pokud rozšíření protokolu čte proud, který se má deserializovat, posune se ukazatel na konec proudu. Aby mohlo ASP.NET správně deserializovat SOAP zprávu, musí být ukazatel nastaven zpět na začátek proudu. Pokud neprovedete tento krok, vznikne chyba deserializace.

Abyste tento proces zjednodušili, můžete použít soukromou metodu `CopyStream()`. Tato metoda zkopíruje obsah jednoho proudu do dalšího proudu. Poté, co je tato metoda vykonána, bude ukazatel nastaven na konec obou datových proudů.

```

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

```

Další součástí, kterou potřebujete, je metoda `ChainStream()`, kterou volá ASP.NET před provedením serializace či deserializace. Vaše rozšíření SOAP může překrýt metodu `ChainStream()` tak, aby byla vložena do zpracovávacího procesu. V tomto momentu může rozšíření cachovat referenci pro původní proud a vytvořit v paměti nový proud, který je pak vrácen do dalšího rozšíření v řetězci.

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

Toto je pochopitelně pouze jedna část příběhu. Je na dalších metodách, aby buď data přečetly ze starého proudu, nebo je zapsaly do nového proudu, v závislosti na etapě, ve které se zpráva nachází. Provedete to voláním metody `CopyStream()`. Výsledkem implementace tohoto poněkud komplikovaného návrhu bude to, že všechna rozšíření SOAP budou mít možnost modifikovat proud SOAP, aniž by si navzájem přepisovali změny. Obvykle jsou metody `ChainStream()` a `CopyStream()` základními součástmi architektury rozšíření SOAP, přičemž jsou identické ve všech SOAP rozšířeních, se kterými se setkáte.

Úkolem metod `LogInputMessage()` a `LogOutputMessage()` je extrahovat informace ze zprávy a zaznamenat je. Obě tyto metody používají metodu `CopyStream()`. Při deserializaci obsahuje vstupní proud kód XML, který má být deserializován, přičemž ukazatel je nastaven na začátek proudu. Metoda `LogInputMessage()` kopíruje vstupní proud do paměťového bufferu proudu a zaprotokoluje obsah proudu. Metoda také nastaví ukazatel na začátek bufferu proudu tak, aby další rozšíření mohlo přistupovat k proudu.

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

Když probíhá serializace, serializér zapisuje do paměťového proudu vytvořeného v `ChainStream()`. Když je po serializaci zavolána funkce `LogOutputMessage()`, ukazatel se nachází na konci proudu. Funkce `LogOutputMessage()` nastaví ukazatel na začátek proudu tak, aby rozšíření mohlo zaprotokolovat obsah proudu. Před vrácením je obsah paměťového proudu zkopírován do výstupního proudu, a ukazatel je vrácen na konec obou datových proudů.

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


Jakmile je proud umístěn ve správné pozici, metody `LogInputMessage()` i `LogOutputMessage()` extrahují data zprávy z proudu SOAP a zapíší položku zprávy protokolu se získanými informacemi. Tato funkce rovněž kontroluje, zdali zpráva SOAP neobsahuje nějakou chybu (fault). V tom případě by byla do protokolu událostí zapsána zpráva jako chyba.

```
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)
    {
        // Tělo SOAP obsahuje vadu.
        if (level > 0)
            WriteToLog(eventMessage, EventLogEntryType.Error);
    }
    else
    {
        // Tělo SOAP obsahuje zprávu.
        if (level > 1)
            WriteToLog(eventMessage, EventLogEntryType.Information);
    }
}
```

Tímto je kód rozšíření `SoapLog` kompletní.

Použití rozšíření `SoapLog`

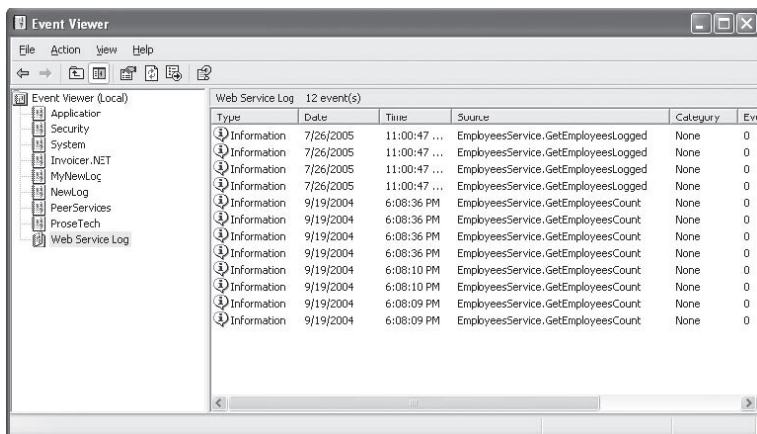
Chcete-li otestovat rozšíření `SoapLog`, musíte na webovou metodu aplikovat `SoapLogAttribute`, jak vidíte zde:

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

Poté musíte vytvořit klientskou aplikaci, která zavolá tuto metodu. Když spustíte klienta a zavoláte metodu, spustí se rozšíření `SoapLog` a vytvoří jednotlivé položky protokolu událostí.

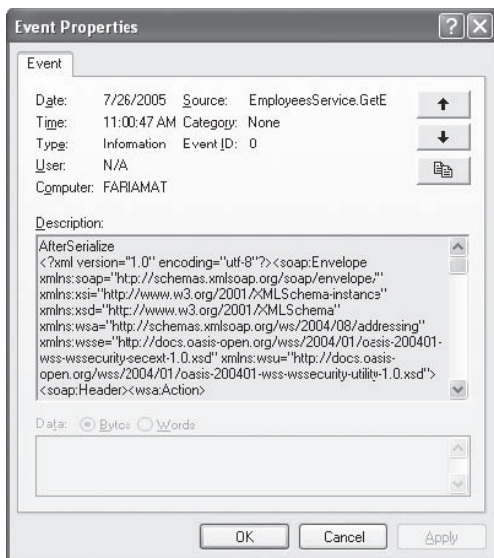
POZNÁMKA Aby rozšíření `SoapLog` úspěšně zapisovalo do protokolu událostí, musí mít pracovní proces ASP.NET (v typickém případě účet ASP.NET) oprávnění pro přístup do protokolu událostí Windows. V opačném případě nebudou zapsány žádné položky. Všimněte si, že pokud rozšíření SOAP selže a v jakémkoliv kroku vygeneruje výjimku, jednoduše bude ignorováno. Váš klientský kód a metody webové služby o tom nebudou informovány.

Abyste si ověřili, zdali se položky objeví, spusťte Event Viewer (z menu Start vyberte Programs/Administrative Tools/Event Viewer). Vyhledejte protokol s názvem Web Service Log. Obrázek 34-5 zobrazuje položky protokolu událostí, které uvidíte, pokud dvakrát zavoláte `GetEmployeesCount()` na úrovni (level) protokolu 3.



Obrázek 34-5. Položky protokolu událostí pro rozšíření SOAP.

Na jednotlivé položky se můžete podívat tak, že na ně dvakrát kliknete. Pole Description zobrazuje úplnou zprávu protokolu událostí s daty XML ze zprávy SOAP, jak vidíte na obrázku 34-6.



Obrázek 34-6. SOAP zpráva v položce protokolu událostí.

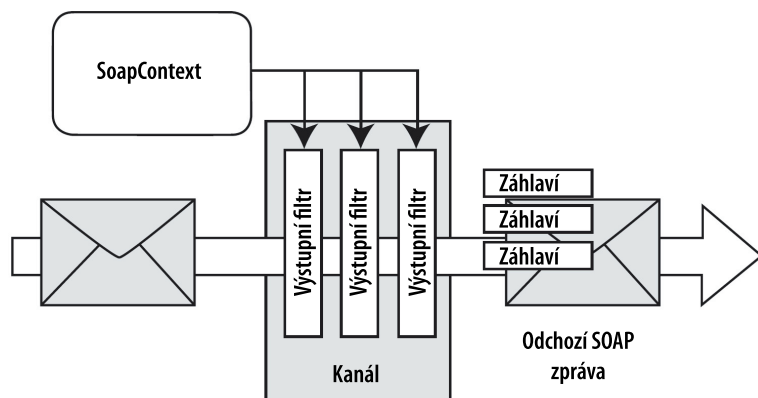
Rozšíření SoapLog jsou užitečným nástrojem při vyvíjení nebo monitorování webových služeb. Měli byste je ale používat s rozvahou. Přestože trasujete například jenom jedinou metodu, počet položek protokolu událostí může rychle narůstat až k tisícům. Jakmile se protokol událostí naplní, staré zprávy jsou automaticky odstraněny. Tyto vlastnosti můžete nakonfigurovat, když kliknete pravým tlačítkem myši na protokol událostí v Event Viewer a vyberete položku Properties.

Vylepšení webových služeb

Od té doby, co se poprvé objevil .NET, není svět standardů webových služeb klidný. Komunita vývojářů neustále vyvíjí a používá množství standardů v různých fázích testování, revidování a standardizace. V příštích verzích .NET bude mnoho z těchto doplňků sloučeno do jediné knihovny třídy. Protože tyto standardy jsou pořád ještě poměrně nové a nestále se mění, nejsou ještě dokončeny. Můžete si však stáhnout jiný nástroj Microsoftu – bezplatnou sadu nástrojů WSE, která vám už dnes umožní získat podporu velkého množství nových standardů webových služeb.

POZNÁMKA Základním pravidlem je, že SOAP, WSDL a UDDI jsou začleněny do .NET a poslední vylepšení (jako například SOAP 1.2) jsou začleněny do .NET 2.0. Avšak standardy, které jsou vytvořeny na jejich základě (jako například standardy kódování založené na SOAP, bezpečnostní standardy a další) součástí .NET ještě nejsou a představují výlučně teritorium WSE.

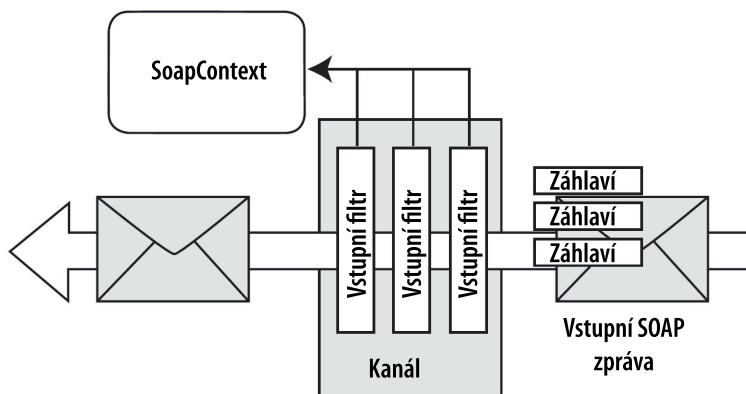
Chcete-li si nainstalovat WSE (nebo si o něm pouze něco přečíst), podívejte se na <http://msdn.microsoft.com/webservices/building/wse>. WSE poskytuje assembly knihovny třídy se sadou užitečných tříd .NET. Kromě toho WSE používá rozšíření SOAP pro úpravu zpráv webové služby. To znamená, že váš kód reaguje na sadu pomocných objektů, nehledě na to, že WSE implementuje odpovídající standardy SOAP. Používání WSE má dvě nevýhody. První je, že sada nástrojů se mění. Verze, která pracuje s Visual Studiem 2005 (verze 3.0) není kompatibilní s dřívějšími verzemi. Druhou nevýhodou je to, že WSE musíte použít jak na straně klienta, tak i serveru. Řečeno jinými slovy – chcete-li použít webovou službu, která používá WSE pro podporu nové funkce, vaše aplikace (pokud je klientem .NET) musí rovněž používat WSE nebo jinou sadu nástrojů, která podporuje stejné standardy (není-li klientem .NET). Mnoho z funkcí WSE je implementováno přes speciální třídu SoapContext. Váš kód reaguje na objekt SoapContext. Poté, co zašlete zprávu, různé filtry (rozšíření SOAP) prozkoumají vlastnosti SoapContext a přidají záhlaví SOAP k odchozí zprávě. Obrázek 34-7 ilustruje tento proces.



Obrázek 34-7. Zpracování odchozí SOAP zprávy.

Hlavní myšlenkou filtrů je to, že mohou být zapojeny pouze tehdy, pouze je to potřeba. Pokud například nepotřebujete bezpečnostní záhlaví, můžete filtr bezpečnostního záhlaví vynechat z kanálu (pipeline) zpracování pouhou změnou konfiguračních nastavení. Protože každý filtr funguje tak, že přidává odlišná SOAP záhlaví, máte možnost zkombinovat současně tolik rozšíření, kolik potřebujete.

Po obdržení zprávy SOAP se celý proces zopakuje v obráceném pořadí. V tomto případě filtry hledají specifická záhlaví SOAP a používají informace v nich obsažené pro konfiguraci objektu SoapContext. Obrázek 34-8 zobrazuje model tohoto zpracování.



Obrázek 34-8. Zpracování příchozí SOAP zprávy.

Které jsou tedy standardy podporované WSE? Úplný seznam je k dispozici v dokumentaci WSE, a obsahuje i podporu součinnosti, autentizace, šifrování zpráv, ustanovování důvěry či směřování (routing) zpráv. WSE vám rovněž umožňuje používat zprávy SOAP pro přímou komunikaci přes TCP spojení (protokol HTTP není vyžadován). V následujících sekcích se podíváme na použití WSE v jednoduchém bezpečnostním scénáři. Chcete-li se s WSE seznámit opravdu důkladně, prostudujte si knihu nazvanou jako *Expert Service-Oriented Architecture in C#: Using the Web Services Enhancements 2.0* (Apress, 2004) od Jeffreyho Hasana nebo dokumentaci, která je součástí sady nástrojů WSE.

Instalace WSE

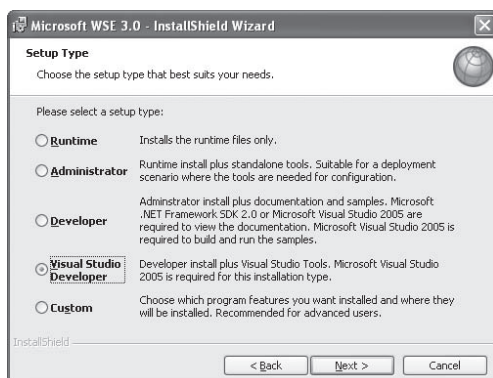
Než budete moci pokračovat, musíte si stáhnout a nainstalovat WSE. Když spustíte průvodce instalací, vyberte si Visual Studio Developer (pokud máte nainstalované Visual Studio 2003), protože ten umožňuje podporu projektu (viz obrázek 34-9).

Chcete-li použít WSE v projektu, musíte provést ještě jeden krok. Ve Visual Studiu klikněte v okně Solution Explorer na název projektu, a z dolní části menu vyberte WSE Settings. Uvidíte dvě zaškrťovací políčka. Pokud vytváříte webovou službu, vyberte obě nastavení, jak vidíte na obrázku 34-10.

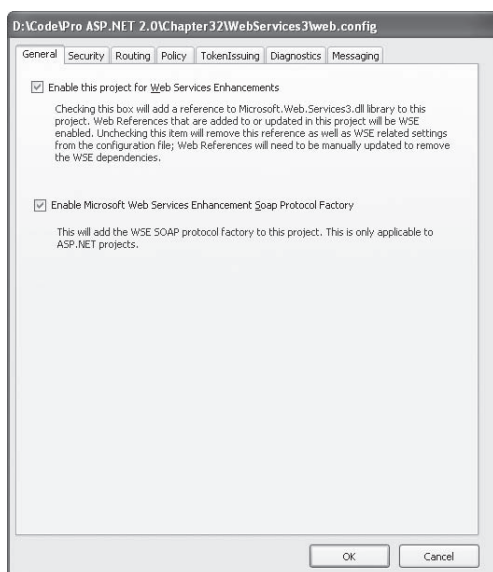
Pokud vytváříte klientskou aplikaci, vyberte pouze první nastavení – Enable This Project for Web Services Enhancements (Aktivovat tento projekt pro vylepšení webových služeb).

Pokud vyberete první volbu, Enable This Project for Web Services Enhancements, Visual Studio automaticky přidá referenci k assembly `Microsoft.Web.Services2.dll` a modifikuje soubor `web.config` tak, aby aplikace přidala podporu pro konfigurační ovladač WSE. Jakékoliv webové reference, které budou od tohoto okamžiku vytvořeny, budou obsahovat podporu WSE v třídě proxy. Již vytvořené webové reference nebudou mít podporu WSE do té doby, dokud je neaktualizujete.

Když vyberete druhou volbu, Enable Microsoft Web Services Enhancements SOAP Extensions, Visual Studio modifikuje soubor `web.config`, aby byly zaregistrovány rozšíření SOAP, které přidávají podporu pro vaše webové služby. Tato volba je nutná pouze pro webové služby ASP.NET, které podporují WSE.



Obrázek 34-9. Instalace WSE.



Obrázek 34-10. Aktivace WSE v projektu.

Provádění autentizace s WSE

Mnoho tříd WSE podporuje bezpečnostní standardy. Jednou z nejjednodušších tříd je třída UsernameToken, která reprezentuje přihlašovací doklady uživatele.

Informace UsernameToken jsou přidány ke zprávě jako záhlaví SOAP. Jsou však přidány způsobem, který odpovídá standardu WS-Security, a jehož implementace by pro vás byla poněkud náročná. Výhodou je, že implementace tohoto obecného bezpečnostního modelu vám umožní přidat autentizaci bez nutnosti vyvíjet proprietární přístup, který může zkomplikovat meziplatformové použití vaší webové služby a její použití třetí stranou. Je také pravděpodobné, že přístup přes ASP.NET a WS-Security bude mnohem bezpečnější a robustnější než přístup, který je vyvinut individuálním vývojářem nebo organizací (bez investování značného množství času a práce).

V současné době má ovšem bezpečnostní infrastruktura WSE několik děr a zcela nesplňuje očekávání na ni kladená. Nicméně se snadno používá a rozšiřuje. Abyste viděli, jak bezpečnostní infrastruktura WSE funguje, podívejte se na následující příklad.

Nastavení ověřovacích informací klienta

V následujícím příkladu si ukážeme, jak použít WSE pro bezpečné zavolání EmployeesService. Prvním krokem je přidání webové reference. Jestliže přidáte referenci na EmployeesService do projektu s aktivovanými WSE, Visual Studio v podstatě vytvoří dvě třídy proxy. První třída proxy (která má stejný název jako třída webové služby) je stejná jako webová služba, která je vygenerována v projektech bez WSE. Druhá třída proxy má k názvu třídy připojenou příponu WSE. Tato třída pochází z třídy Microsoft.Web.Services3.WebServicesClientProtocol a obsahuje podporu pro funkcionality WSE (v tomto případě přidává tzv. WS-Security tokens).

Pokud tedy použít WS-Security ve webové službě EmployeesService, potřebujete vytvořit projekt s aktivovanými WSE, přidat nebo obnovit webovou referenci a pak modifikovat svůj kód tak, aby používal proxy EmployeeServiceWse. Pokud jste absolvovali všechny tyto kroky, můžete přidat nový UsernameToken s vašimi přihlašovacími informacemi:

```
// Vytvoř proxy.
EmployeesServiceWse proxy = new EmployeesServiceWse();
// Přidej WS-Security token.
proxy.RequestSoapContext.Security.Tokens.Add(
    new UsernameToken(userName, password, PasswordOption.SendPlainText));
// Navaž výsledky.
GridView1.DataSource = proxy.GetEmployees().Tables[0];
```

Jak je vidět z tohoto kódu, bezpečnostní příznak (security token) je přidán jako objekt UsernameToken. Je vložen do kolekce Security.Tokens z RequestSoapContext, což je objekt SoapContext, který reprezentuje zprávu o požadavku, kterou chcete zaslat.

Pokud chcete použít tento kód tak, jak byl napsán, musíte importovat následující jmenné prostory:

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

POZNÁMKA Všimněte si, že jmenné prostory WSE obsahují číslo 3, které označuje třetí verzi sady nástrojů WSE. Je to proto, že třetí verze není zpětně kompatibilní s prvními dvěma. Pro předcházení konfliktů s částečně aktualizovanými aplikacemi jsou třídy WSE podle své verze rozděleny do odlišných jmenných prostorů. Je to část chaotické reality, která se objevuje při práci s novými standardy webových služeb.

Konstruktor třídy UsernameToken akceptuje tři parametry: řetězec se jménem uživatele, řetězec s heslem a volbu pro hašování. Bohužel – pokud chcete ve WSE použít standardního poskytovatele autentizace (který používá autentizaci Windows), musíte použít PasswordOption.SendPlainText. Výsledkem je, že tento kód je nedostatečně chráněný a často se stává terčem síťové špionáže, pokud ovšem požadavek nezašlete přes bezpečné SSL spojení.

Ačkoliv jsou v tomto případě k požadavku přidány pouze dva další detaily, SOAP zpráva je ve skutečnosti mnohem komplexnější – kvůli struktuře standardu WS-Security. Ten definuje dodatečné detaily, jako je třeba

datum vypršení platnosti (kvůli prevenci opakovaných útoků) a příležitostná hodnota (což je náhodná hodnota, která může být připojena do hashe kvůli zvýšení bezpečnosti). Zde uvádíme poněkud zkrácený příklad SOAP zprávy o požadavku:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
xmlns:wss="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+XWIqf083NiQ==</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>
```

Čtení pověřovacích dokladů ve webové službě

Služba s aktivovanými WSE prozkoumá dodaný příznak (token) a neprodleně jej ověří. Standardní poskytovatel autentizace, který je začleněn do WSE, používá autentizaci Windows, což znamená, že uživatelské jméno a heslo je extrahováno ze SOAP záhlaví, přičemž oba tyto údaje jsou použity pro přihlášení uživatele pod nějakým účtem Windows. Jestliže příznak (token) není namapován k platnému účtu Windows, je klientovi vrácena chybová zpráva. Pokud však není dodán žádný příznak, žádná chyba nevznikne. Je na pouze vás zkontrolovat tento stav na webovém serveru, abyste omezili přístup ke specifickým webovým metodám.

Bohužel – WSE není integrován takovým způsobem, aby mohl používat objekt User. Příznaky (tokens) proto potřebujete získat z aktuálního kontextu. WSE poskytuje RequestSoapContext. Pomocí vlastnosti RequestSoapContext.Current můžete získat instanci třídy SoapContext, která reprezentuje poslední obdrženou zprávu. Poté můžete prozkoumat kolekci SoapContext.Security.Tokens.

Abyste tuto úlohu zjednodušili, je dobré si vytvořit nějakou soukromou metodu, jako je například metoda, kterou uvádíme zde. Ta ověřuje, zdali existuje příznak (token), a vyvolává výjimku, pokud tomu tak není. Jinak vrátí jméno uživatele.

```
private string GetUsernameToken()
{
    // Ačkoli může existovat více příznaků, pouze jeden
    // z těchto příznaků bude UsernameToken.
    foreach (UsernameToken token in RequestSoapContext.Current.Security.Tokens)
    {
        return token.Username;
    }
    throw new SecurityException("Missing security token");
}
```

Na začátku webové metody můžete zavolat metodu GetUsernameToken(), čímž zajistíte, že zabezpečení bude účinné. To je obecně dobrý přístup pro posílení bezpečnosti. Je však také důležité mít na paměti jeho omezení. Nepodporuje například hašování nebo šifrování pověřovacích dokladů uživatele. Rovněž nepodporuje pokročilejší protokoly autentizace Windows, jako je například digestní autentizace nebo integrovaná autentizace Windows. Je to proto, že autentizace je zajišťována WSE rozšířeními, nikoliv IIS. Klient pro posílání informací vždy potřebuje nějaké uživatelské jméno heslo. Klient nemá k dispozici žádný způsob, jak automaticky posílat přihlašovací doklady aktuálního uživatele, jak jsme vám to ukázali u objektu CredentialCache v této kapitole. Vlastnost Credentials z proxy je totiž zcela ignorována.

Naštěstí nemusíte být omezováni mírnější formou Windows Autentizace, která je poskytována výchozí službou WSE autentizace. Snadno si totiž můžete vytvořit svoji vlastní logiku autentizace, jak si ukážeme v následující sekci.

Vlastní autentizace

Vytvořením vlastní třídy autentizace můžete provádět autentizaci libovolného zdroje dat, včetně souboru XML nebo databáze. Pokud si chcete vytvořit svůj vlastní autentizátor, musíte jednoduše vytvořit třídu, která bude odvozena z UsernameTokenManager a překrýt metodu AuthenticateToken(). V této metodě váš kód potřebuje vyhledat uživatele, který se snaží být autentizován a musí vrátit heslo pro tohoto uživatele. ASP.NET pak porovná toto heslo s přihlašovacími doklady uživatele a rozhodne, zdali bude autentizace úspěšná, či nikoliv.

Vytvoření této třídy je poměrně jednoduché. Zde uvádíme příklad, který jednoduše vrátí natvrdo vložená hesla dvou uživatelů. Je to rychlý a snadný test, ačkoliv v situaci z reálného světa by pro získání stejných informací byl použit kód ADO.NET.

```
public class CustomAuthenticator : UsernameTokenManager
{
    protected override string AuthenticateToken(UsernameToken token)
    {

```



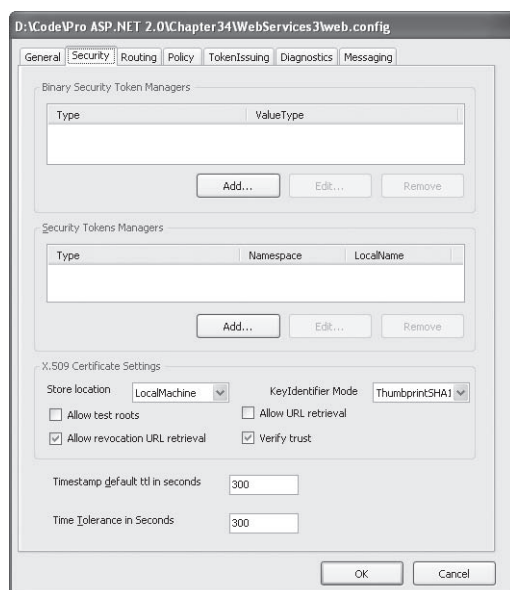
```

string username = token.Username;
if (username == "dan")
return "secret";
else if (username == "jenny")
return "opensesame";
else
return "";
}
}

```

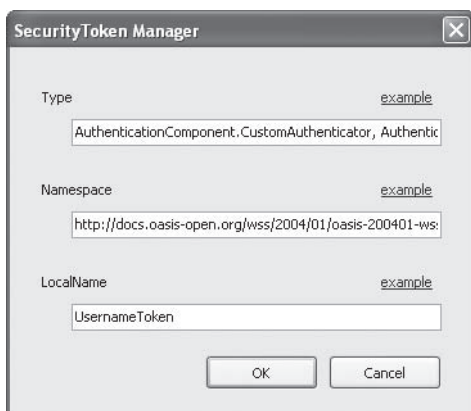
Důvodem, proč sami neprovádíte test hesla, je to, že druh porovnání je závislý na tom, jak jsou přihlašovací doklady zašifrovány. Jsou-li například předávány jako čistý text, potřebujete provést jednoduché porovnání řetězců. Pokud jsou hašované, potřebujete vytvořit nový haš hesla s použitím stejného standardizovaného algoritmu, který musí vycházet ze stejných dat (včetně stejné příležitostné hodnoty ze zprávy klienta). WSE může toto srovnání provést zcela automaticky, což dramaticky zjednodušuje váš kód. Jediný potenciální problém je v tom, že heslo uživatele potřebujete mít uloženo na webovém serveru v takové formě, aby bylo možné jej získat. Pokud v databázi ukládáte pouze haš hesla, nebudete schopni předat ASP.NET původní heslo, což má za následek tu nepříjemnou věc, že nebudete schopni vytvořit haš pověřovacích dokladů, který je potřebný pro autentizaci uživatele.

Jakmile jste vytvořili autentizační třídu, potřebujete sdělit WSE, aby tuto třídu použil pro autentizaci příznaků (tokens) uživatele jejím zaregistrováním do souboru web.config. To provedete tak, že v okně Solution Explorer kliknete pravým tlačítkem myši na název projektu a vyberte WSE Settings. Dále pak vyberte záložku Security (zobrazenou na obrázku 34-11).



Obrázek 34-11. Nastavení zabezpečení.

V sekci Security Tokens Managers pak klikněte na tlačítko Add, čímž zobrazíte dialogové okno SecurityToken Manager (viz obrázek 34-12).



Obrázek 34-12. Konfigurace nového UsernameTokenManager.

V dialogovém okně SecurityToken Manager potřebujete specifikovat tři informace:

- **Type.** Vložte plně kvalifikovaný název třídy, za kterým bude následovat čárka a za ní název assembly (bez přípony .dll). Pokud například máte webový projekt s názvem MyProject s autentizační třídou CustomAuthenticator, vložte MyProject.CustomAuthenticator,MyProject.
- **Namespace.** Vložte následující řetězec: `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd`.
- **QName.** Vložte následující řetězec: UsernameToken.

V zásadě stejným způsobem vybudujete i klienta, bez ohledu na způsob, jakým je vaše autentizace prováděna na serveru. Když si však vytvoříte vlastní autentizační třídu, můžete si specifikovat používání hašovaných hesel, jak vidíte zde:

```
// Vytvoř proxy.
EmployeesServiceWse proxy = new EmployeesServiceWse();
// Přidej WS-Security token.
proxy.RequestSoapContext.Security.Tokens.Add(
    new UsernameToken("dan", "secret", PasswordOption.SendHashed));
// Navaž výsledky.
GridView1.DataSource = proxy.GetEmployees().Tables[0];
GridView1.DataBind();
```

Nejlepší na tom je, že uživatelské heslo nemusíte hašovat nebo šifrovat ručně. WSE jej totiž provádí hašování podle vašich instrukcí a automaticky porovnává haš na straně serveru. Dokonce využívá i příležitostnou hodnotu pro prevenci opakovaných útoků.

Shrnutí

V této kapitole jste se naučili používat nejrůznější pokročilé techniky SOAP včetně způsobů asynchronního volání webových služeb, podpory bezpečnosti a použití rozšíření SOAP. Svět webových služeb se určitě bude dále rozvíjet, a proto sledujte nové standardy a nové schopnosti, které se objeví v příštích verzích .NET Frameworku.