Poznaj technologię WCF
i wykorzystaj potencjał
Microsoft Azure AppFabric Service Bus

WYDANIE III

Programowanie usług

WCF

HELION

O'REILLY®

Juval Löwy
Wprowadzenie Clemens Vasters

Tytuł oryginału: Programming WCF Services:
Mastering WCF and the Azure AppFabric Service Bus, 3rd edition

Tłumaczenie: Mikołaj Szczepaniak (wstęp, rozdz. 4 – 6, 11, dodatki),
Weronika Łabaj (rozdz. 1 – 3),
Krzysztof Rychlicki-Kicior (rozdz. 7 – 10)

ISBN: 978-83-246-3617-4

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/prowcf
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 4.

Zarządzanie instancjami

Zarządzanie instancjami (ang. *instance management*) to określenie, które stosuję w kontekście zbioru technik używanych przez technologię WCF do kojarzenia żądań klientów z instancjami usług — technik decydujących o tym, która instancja usługi obsługuje które żądanie klienta i kiedy to robi. Zarządzanie instancjami jest konieczne, ponieważ zasadnicze różnice dzielące poszczególne aplikacje w wymiarze potrzeb skalowalności, wydajności, przepustowości, trwałości, transakcji i wywołań kolejkowanych w praktyce uniemożliwiają opracowanie jednego, uniwersalnego rozwiązania. Istnieje jednak kilka klasycznych technik zarządzania instancjami, które można z powodzeniem stosować w wielu rożnych aplikacjach i które sprawdzają się w najróżniejszych scenariuszach i modelach programistycznych. Właśnie o tych technikach będzie mowa w niniejszym rozdziale. Ich dobre zrozumienie jest warunkiem tworzenia skalowalnych i spójnych aplikacji. Technologia WCF obsługuje trzy rodzaje aktywacji instancji: przydzielanie (i niszczenie) usług **przez wywołania**, gdzie każde żądanie klienta powoduje utworzenie nowej instancji; przydzielanie instancji dla każdego połączenia z klientem (w przypadku **usług sesyjnych**) oraz **usługi singletonowe**, w których jedna instancja usługi jest współdzielona przez wszystkie aplikacje klienckie (dla wszystkich połączeń i aktywacji). W tym rozdziale zostaną omówione zalety i wady wszystkich trzech trybów zarządzania instancjami. Rozdział zawiera też wskazówki, jak i kiedy najskuteczniej używać poszczególnych trybów. Omówię też kilka pokrewnych zagadnień, jak zachowania, konteksty, operacje demarkacyjne, dezaktywacja instancji, usługi trwałe czy tzw. dławienie.[1]

Zachowania

Tryb instancji usług jest w istocie jednym ze szczegółowych aspektów implementacji po stronie usługi, który w żaden sposób nie powinien wpływać na funkcjonowanie strony klienckiej. Na potrzeby obsługi tego i wielu innych lokalnych aspektów strony usługi technologia WCF definiuje pojęcie **zachowań** (ang. *behavior*). Zachowanie to pewien lokalny atrybut usługi lub klienta, który nie wpływa na wzorce komunikacji pomiędzy tymi stronami. Klienty nie powinny zależeć od zachowań usług, a same zachowania nie powinny być ujawniane w powiązaniach usług ani publikowanych metadanych. We wcześniejszych rozdziałach mieliśmy już do czynienia z dwoma zachowaniami usług. W rozdziale 1. użyto zachowań metadanych usługi do zasygnalizowania

[1] Ten rozdział zawiera fragmenty moich artykułów zatytułowanych *WCF Essentials: Discover Mighty Instance Management Techniques for Developing WCF Apps* („MSDN Magazine", czerwiec 2006) oraz *Managing State with Durable Services* („MSDN Magazine", październik 2008).

hostowi konieczności publikacji metadanych usługi za pośrednictwem żądania HTTP-GET oraz do implementacji punktu końcowego MEX, natomiast w rozdziale 3. wykorzystano zachowanie usługi do ignorowania rozszerzenia obiektu danych. Na podstawie przebiegu komunikacji czy wymienianych komunikatów żaden klient nie może stwierdzić, czy usługa ignoruje rozszerzenie obiektu danych ani jak zostały opublikowane jej metadane.

Technologia WCF definiuje dwa typy deklaratywnych zachowań strony usługi opisywane przez dwa odpowiednie atrybuty. Atrybut `ServiceBehaviorAttribute` służy do konfigurowania zachowań usługi, czyli zachowań wpływających na jej wszystkie punkty końcowe (kontrakty i operacje). Atrybut `ServiceBehavior` należy stosować bezpośrednio dla klasy implementacji usługi.

Atrybut `OperationBehaviorAttribute` służy do konfigurowania **zachowań operacji**, czyli zachowań wpływających tylko na implementację określonej operacji. Atrybutu `OperationBehavior` można użyć tylko dla metody implementującej operację kontraktu, nigdy dla definicji tej operacji w samym kontrakcie. Praktyczne zastosowania atrybutu `OperationBehavior` zostaną pokazane zarówno w dalszej części tego rozdziału, jak i w kolejnych rozdziałach.

W tym rozdziale atrybut `ServiceBehavior` będzie używany do konfigurowania trybu instancji usługi. Na listingu 4.1 pokazano przykład użycia tego atrybutu do zdefiniowania właściwości `InstanceContextMode` typu wyliczeniowego `InstanceContextMode`. Wartość wyliczenia `Instance ↪ContextMode` steruje trybem zarządzania instancjami stosowanym dla tej usługi.

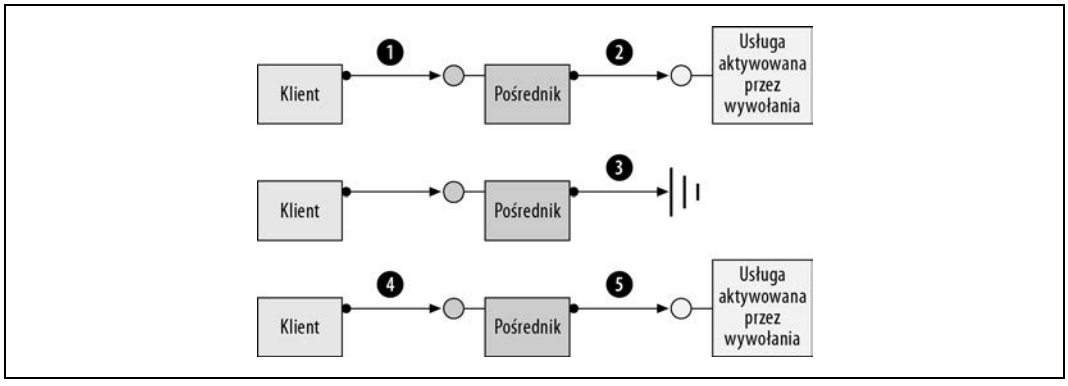
Listing 4.1. Przykład użycia atrybutu ServiceBehaviorAttribute do skonfigurowania trybu kontekstu instancji

```
public enum InstanceContextMode
{
   PerCall,
   PerSession,
   Single
}
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute,...
{
   public InstanceContextMode InstanceContextMode
   {get;set;}
   // Pozostałe składowe…
}
```

Typ wyliczeniowy nieprzypadkowo nazwano `InstanceContextMode` zamiast `InstanceMode`, ponieważ jego wartości sterują trybem tworzenia instancji kontekstu zarządzającego daną instancją, nie trybem działania samej instancji (w rozdziale 1. wspomniano, że kontekst instancji wyznacza najbardziej wewnętrzny zasięg usługi). Okazuje się jednak, że instancja i jej kontekst domyślnie są traktowane jako pojedyncza jednostka, zatem wspomniane wyliczenie steruje także cyklem życia instancji. W dalszej części tego rozdziału i w kolejnych rozdziałach omówię możliwe sposoby (i przyczyny) oddzielania obu elementów.

Usługi aktywowane przez wywołania

Jeśli rodzaj usługi skonfigurowano z myślą o **aktywacji przez wywołania** (ang. *per-call activation*), instancja tej usługi (obiekt środowiska CLR) istnieje tylko w trakcie obsługi wywołania ze strony klienta. Każde żądanie klienta (czyli wywołanie metody dla kontraktu WCF) otrzymuje nową, dedykowaną instancję usługi. Działanie usługi w trybie aktywacji przez wywołania wyjaśniono w poniższych punktach (wszystkie te kroki pokazano też na rysunku 4.1):



Rysunek 4.1. Tryb tworzenia instancji przez wywołania

1. Klient wywołuje pośrednika (ang. *proxy*), który kieruje to wywołanie do odpowiedniej usługi.
2. Środowisko WCF tworzy nowy kontekst z nową instancją usługi i wywołuje metodę tej instancji.
3. Jeśli dany obiekt implementuje interfejs `IDisposable`, po zwróceniu sterowania przez wywołaną metodę środowisko WCF wywołuje metodę `IDisposable.Dispose()` zdefiniowaną przez ten obiekt. Środowisko WCF niszczy następnie kontekst usługi.
4. Klient wywołuje pośrednika, który kieruje to wywołanie do odpowiedniej usługi.
5. Środowisko WCF tworzy obiekt i wywołuje odpowiednią metodę nowego obiektu.

Jednym z najciekawszych aspektów tego trybu jest zwalnianie (niszczenie) instancji usługi. Jak już wspomniano, jeśli usługa obsługuje interfejs `IDisposable`, środowisko WCF automatycznie wywoła metodę `Dispose()`, umożliwiając tej usłudze zwolnienie zajmowanych zasobów. Warto pamiętać, że metoda `Dispose()` jest wywoływana w tym samym wątku co oryginalna metoda usługi i że dysponuje kontekstem operacji (patrz dalsza część tego rozdziału). Po wywołaniu metody `Dispose()` środowisko WCF odłącza instancję usługi od pozostałych elementów swojej infrastruktury, co oznacza, że instancja może zostać zniszczona przez proces odzyskiwania pamięci.

Zalety usług aktywowanych przez wywołania

W klasycznym modelu programowania klient-serwer implementowanym w takich językach jak C++ czy C# każdy klient otrzymuje własny, dedykowany obiekt serwera. Zasadniczą wadą tego modelu jest niedostateczna skalowalność. Wyobraźmy sobie aplikację, która musi obsłużyć wiele aplikacji klienckich. Typowym rozwiązaniem jest tworzenie po stronie serwera obiektu w momencie uruchamiania każdej z tych aplikacji i zwalnianie go zaraz po zakończeniu działania aplikacji klienckiej. Skalowalność klasycznego modelu klient-serwer jest o tyle trudna, że aplikacje klienckie mogą utrzymywać swoje obiekty bardzo długo, mimo że w rzeczywistości używają ich przez niewielki ułamek tego czasu. Takie obiekty mogą zajmować kosztowne i deficytowe zasoby, jak połączenia z bazami danych, porty komunikacyjne czy pliki. Jeśli każdemu klientowi jest przydzielany osobny obiekt, te cenne i (lub) ograniczone zasoby są zajmowane przez długi czas, co prędzej czy później musi doprowadzić do ich wyczerpania.

Lepszym modelem aktywacji jest przydzielanie obiektu dla klienta tylko na czas wykonywania wywołania usługi. W takim przypadku należy utworzyć i utrzymywać w pamięci tylko tyle obiektów, ile współbieżnych wywołań jest obsługiwanych przez usługę (nie tyle obiektów, ile istnieje niezamkniętych aplikacji klienckich). Z doświadczenia wiem, że w typowym systemie korporacyjnym, szczególnie takim, gdzie o działaniu aplikacji klienckich decydują użytkownicy, tylko 1% klientów wykonuje współbieżne wywołania (w przypadku mocno obciążonych systemów ten odsetek wzrasta do 3%). Oznacza to, że jeśli system jest w stanie obsłużyć sto kosztownych instancji usługi, w typowych okolicznościach może współpracować z dziesięcioma tysiącami klientów. Tak duże możliwości systemu wynikają wprost z trybu aktywacji przez wywołania. Pomiędzy wywołaniami klient dysponuje tylko referencją do pośrednika, który nie zajmuje właściwego obiektu po stronie usługi. Oznacza to, że można zwolnić kosztowne zasoby zajmowane przez instancję usługi na długo przed zamknięciem pośrednika przez klienta. Podobnie uzyskiwanie dostępu do zasobów jest odkładane do momentu, w którym te zasoby rzeczywiście są potrzebne klientowi.

Należy pamiętać, że wielokrotne tworzenie i niszczenia instancji po stronie usługi bez zamykania połączenia z klientem (a konkretnie z pośrednikiem po stronie klienta) jest nieporównanie mniej kosztowne niż wielokrotne tworzenie zarówno instancji, jak i połączenia. Drugą ważną zaletą tego rozwiązania jest zgodność z zasobami transakcyjnymi i technikami programowania transakcyjnego (patrz rozdział 7.), ponieważ przydzielanie instancji usługi i niezbędnych zasobów osobno dla każdego wywołania ułatwia zapewnianie spójności stanu tych instancji. Trzecią zaletą usług aktywowanych przez wywołania jest możliwość stosowania tych usług w środowisku z kolejkowanymi, rozłączonymi wywołaniami (patrz rozdział 9.), ponieważ tak działające instancje można łatwo odwzorowywać na kolejkowane komunikaty.

Konfiguracja usług aktywowanych przez wywołania

Aby skonfigurować typ usługi aktywowanej przez wywołania, należy użyć atrybutu `Service ↪Behavior` z wartością `InstanceContextMode.PerCall` ustawioną we właściwości `InstanceContextMode`:

```
[ServiceContract]
interface IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{...}
```

Na listingu 4.2 pokazano przykład prostej usługi aktywowanej przez wywołania i jej klienta. Jak widać w danych wynikowych tego programu, dla każdego wywołania metody ze strony klienta jest konstruowana nowa instancja usługi.

Listing 4.2. Usługa aktywowana przez wywołania i jej klient

```
////////////////////////// Kod usługi ///////////////////////
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   void MyMethod();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract,IDisposable
{
```

```
   int m_Counter = 0;

   MyService()
   {
      Trace.WriteLine("MyService.MyService()");
   }
   public void MyMethod()
   {
      m_Counter++;
      Trace.WriteLine("Licznik = " + m_Counter);
   }
   public void Dispose()
   {
      Trace.WriteLine("MyService.Dispose()");
   }
}
////////////////////// Kod klienta ////////////////////
MyContractClient proxy = new MyContractClient();

proxy.MyMethod();
proxy.MyMethod();

proxy.Close();

// Możliwe dane wynikowe
MyService.MyService()
Licznik = 1
MyService.Dispose()
MyService.MyService()
Licznik = 1
MyService.Dispose()
```

Usługi aktywowane przez wywołania i sesje transportowe

Stosowanie usług aktywowanych przez wywołania nie zależy od obecności sesji transportowej (patrz rozdział 1.). Sesja transportowa porządkuje wszystkie komunikaty kierowane przez określonego klienta do konkretnego kanału. Nawet jeśli sesję skonfigurowano pod kątem aktywacji przez wywołania, stosowanie sesji transportowej wciąż jest możliwe, tyle że każde wywołanie usługi WCF będzie powodowało utworzenie nowego kontekstu tylko na potrzeby tego wywołania. Jeśli sesje poziomu transportowego nie są stosowane, usługa — o czym za chwilę się przekonamy — zawsze, niezależnie od konfiguracji zachowuje się tak, jakby była aktywowana przez wywołania.

Jeśli usługa aktywowana przez wywołania dysponuje sesją transportową, komunikacja z klientem jest przerywana po pewnym czasie braku aktywności tej sesji (odpowiedni limit czasowy domyślnie wynosi 10 minut). Po osiągnięciu tego limitu czasowego klient nie może dalej używać pośrednika do wywoływania operacji udostępnianych przez usługę aktywowaną przez wywołania, ponieważ sesja transportowa została zamknięta.

Sesje transportowe mają największy wpływ na działanie usług aktywowanych przez wywołania w sytuacji, gdy te usługi skonfigurowano z myślą o dostępie jednowątkowym (zgodnie z domyślnymi ustawieniami środowiska WCF — patrz rozdział 8.). Sesja transportowa wymusza wówczas wykonywanie operacji w trybie blokada-krok (ang. *lock-step*), gdzie wywołania od tego samego pośrednika są szeregowane. Oznacza to, że nawet jeśli jeden klient jednocześnie generuje wiele wywołań, wszystkie te wywołania są pojedynczo, kolejno wykonywane przez różne instancje. Takie działanie ma istotny wpływ na proces zwalniania instancji. Środowisko

WCF nie blokuje działania klienta w czasie zwalniania instancji usługi. Jeśli jednak w czasie wykonywania metody `Dispose()` klient wygeneruje kolejne wywołanie, dostęp do nowej instancji i obsługa tego wywołania będą możliwe dopiero po zwróceniu sterowania przez metodę `Dispose()`. Na przykład dane wynikowe z końca listingu 4.2 ilustrują przypadek, w którym istnieje sesja transportowa. W przedstawionym scenariuszu drugie wywołanie może być wykonane dopiero po zwróceniu sterowania przez metodę `Dispose()`. Gdyby usługa z listingu 4.2 nie stosowała sesji transportowej, dane wynikowe co prawda mogłyby być takie same, ale też mogłyby pokazywać zmienioną kolejność wywołań (w wyniku działania nieblokującej metody `Dispose()`):

```
MyService.MyService()
Licznik = 1
MyService.MyService()
Licznik = 1
MyService.Dispose()
MyService.Dispose()
```

Projektowanie usług aktywowanych przez wywołania

Mimo że w teorii tryb aktywacji instancji przez wywołania można stosować dla usług dowolnego typu, w rzeczywistości usługę i jej kontrakt należy od początku projektować z myślą o obsłudze tego trybu. Zasadniczy problem polega na tym, że klient nie wie, że w odpowiedzi na każde swoje wywołanie otrzymuje nową instancję. Usługi aktywowane przez wywołania muszą **utrzymywać swój stan**, tzn. muszą tak zarządzać tym stanem, aby klient miał wrażenie istnienia ciągłej sesji. Usługi stanowe z natury rzeczy różnią się od usług bezstanowych. Gdyby usługa aktywowana przez wywołania była w pełni bezstanowa, w praktyce aktywacja dla kolejnych wywołań w ogóle nie byłaby potrzebna. Właśnie istnienie stanu, a konkretnie kosztownego stanu obejmującego zasoby, decyduje o potrzebie stosowania trybu aktywacji przez wywołania. Instancja usługi aktywowanej przez wywołania jest tworzona bezpośrednio przed każdym wywołaniem metody i niszczona bezpośrednio po każdym wywołaniu. Oznacza to, że na początku każdego wywołania obiekt powinien inicjalizować swój stan na podstawie wartości zapisanych w jakiejś pamięci, a na końcu wywołania powinien zapisać swój stan w tej pamięci. W roli pamięci zwykle stosuje się albo bazę danych, albo system plików, jednak równie dobrze można wykorzystać jakąś pamięć ulotną, na przykład zmienne statyczne.

Okazuje się jednak, że nie wszystkie elementy stanu obiektu można zapisać. Jeśli na przykład stan obejmuje połączenie z bazą danych, obiekt musi ponownie uzyskiwać to połączenie podczas tworzenia instancji (lub na początku każdego wywołania) oraz zwalniać je po zakończeniu wywołania lub we własnej implementacji metody `IDisposable.Dispose()`.

Stosowanie trybu aktywacji przez wywołania ma zasadniczy wpływ na projekt operacji — każda operacja musi otrzymywać parametr identyfikujący instancję usługi, której stan należy odtworzyć. Instancja używa tego parametru do odczytania swojego stanu z pamięci (zamiast stanu innej instancji tego samego typu). Właśnie dlatego pamięć stanów zwykle jest porządkowana według kluczy (może mieć na przykład postać statycznego słownika w pamięci lub tabeli bazy danych). Parametry stanów mogą reprezentować na przykład numer konta w przypadku usługi bankowej, numer zamówienia w przypadku usługi przetwarzającej zamówienia itp.

Listing 4.3 zawiera szablon implementacji usługi aktywowanej przez wywołania.

Listing 4.3. Implementacja usługi aktywowanej przez wywołania

```
[DataContract]
class Param
{...}

[ServiceContract]
interface IMyContract
{
   [OperationContract]
   void MyMethod(Param stateIdentifier);
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyPerCallService : IMyContract,IDisposable
{
   public void MyMethod(Param stateIdentifier)
   {
      GetState(stateIdentifier);
      DoWork();
      SaveState(stateIdentifier);
   }
   void GetState(Param stateIdentifier)
   {...}
   void DoWork()
   {...}
   void SaveState(Param stateIdentifier)
   {...}
   public void Dispose()
   {...}
}
```

Klasa implementuje operację `MyMethod()`, która otrzymuje na wejściu parametr typu `Param` (czyli typu danych wymyślonego na potrzeby tego przykładu) identyfikujący odpowiednią instancję:

```
public void MyMethod(Param stateIdentifier)
```

Instancja usługi używa tego identyfikatora do uzyskania swojego stanu i jego ponownego zapisania na końcu wywołania wspomnianej metody. Elementy składowe stanu, które są wspólne dla wszystkich klientów, można przydzielać w kodzie konstruktora i zwalniać w kodzie metody `Dispose()`.

Tryb aktywacji przez wywołania sprawdza się najlepiej w sytuacji, gdy każde wywołanie metody w pełni realizuje określone zadanie (gdy po zwróceniu sterowania przez metodę nie są wykonywane w tle żadne dodatkowe czynności). Ponieważ obiekt jest usuwany zaraz po zakończeniu wykonywania metody, nie należy uruchamiać żadnych wątków działających w tle ani stosować wywołań asynchronicznych.

Ponieważ metoda usługi aktywowana przez wywołania każdorazowo odczytuje stan instancji z jakiejś pamięci, usługi tego typu wprost doskonale współpracują z mechanizmami równoważenia obciążeń (pod warunkiem że repozytorium stanów ma postać globalnego zasobu dostępnego dla wszystkich komputerów). Mechanizm równoważenia obciążeń może kierować wywołania na różne serwery, ponieważ każda usługa aktywowana przez wywołania może zrealizować wywołanie po uzyskaniu stanu odpowiedniej instancji.

Wydajność usług aktywowanych przez wywołania

Usługi aktywowane przez wywołania są kompromisem polegającym na nieznacznym spadku wydajności (z powodu dodatkowych kosztów uzyskiwania i zapisywania stanu instancji przy okazji każdego wywołania metody) na rzecz większej skalowalności (dzięki przechowywaniu stanu i związanych z nim zasobów). Nie istnieją jasne, uniwersalne reguły, według których można by oceniać, kiedy warto poświęcić część wydajności w celu znacznej poprawy skalowalności. W pewnych przypadkach najlepszym rozwiązaniem jest profilowanie systemu i zaprojektowanie części usług korzystających z tej formy aktywacji i części usług pracujących w innych trybach.

Operacje czyszczenia środowiska

To, czy typ usługi obsługuje interfejs `IDisposable`, należy traktować jako szczegół implementacji, który w żaden sposób nie wpływa na sposób funkcjonowania klienta. Sam klient w żaden sposób nie może wywołać metody `Dispose()`. Podczas projektowania kontraktu dla usługi aktywowanej przez wywołania należy unikać definiowania operacji, których zadaniem byłoby „czyszczenie" stanu czy zwalnianie zasobów, jak w poniższym przykładzie:

```
// Tego należy unikać
[ServiceContract]
interface IMyContract
{
   void DoSomething();
   void Cleanup();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyPerCallService : IMyContract,IDisposable
{
   public void DoSomething()
   {...}
   public void Cleanup()
   {...}
   public void Dispose()
   {
      Cleanup();
   }
}
```

Powyższy projekt ma dość oczywistą wadę — jeśli klient wywoła metodę `Cleanup()`, spowoduje utworzenie nowego obiektu tylko na potrzeby wykonania tej metody. Co więcej, po jej zakończeniu środowisko WCF wywoła metodę `IDisposable.Dispose()` (jeśli istnieje w tej usłudze), aby ponownie zwolnić zasoby i wyczyścić stan usługi.

Wybór usług aktywowanych przez wywołania

Mimo że model programowania usług aktywowanych przez wywołania może wydawać się dość dziwny programistom przyzwyczajonym do architektury klient-serwer, właśnie ten tryb zarządzania instancjami sprawdza się najlepiej w przypadku wielu usług WCF. Przewaga usług aktywowanych przez wywołania polega na większej skalowalności, a przynajmniej na stałej skalowalności. Podczas projektowania usług staram się trzymać pewnej reguły skalowalności, którą nazwałem 10X. Zgodnie z tą regułą każdą usługę należy tak zaprojektować, aby obsługiwała obciążenie większe o co najmniej rząd wielkości od początkowo sformułowanych wymagań. W żadnej dziedzinie inżynierii nie projektuje się rozwiązań czy systemów z myślą

o radzeniu sobie z minimalnym zakładanym obciążeniem. Nie chcielibyśmy przecież wchodzić do budynku, którego belki stropowe mogą podtrzymać tylko minimalne obciążenie, ani korzystać z windy, której liny mogą utrzymać tylko tylu pasażerów, dla ilu ta winda uzyskała atest. Dokładnie tak samo jest w świecie systemów informatycznych — po co projektować system z myślą o bieżącym obciążeniu, skoro każdy dodatkowy pracownik przyjmowany do firmy w celu poprawy jej wyników biznesowych będzie powodował dodatkowe obciążenie tego systemu? Systemy informatyczne należy projektować raczej na lata, tak aby radziły sobie zarówno z bieżącym obciążeniem, jak i dużo większym obciążeniem w przyszłości. Każdy, kto stosuje regułę 10X, błyskawicznie dochodzi do punktu, w którym skalowalność usług aktywowanych przez wywołania jest bezcenna.

Usługi sesyjne

Środowisko WCF może utrzymywać sesję logiczną łączącą klienta z określoną instancją usługi. Klient, który tworzy nowego pośrednika dla usługi skonfigurowanej jako tzw. **usługa sesyjna** (ang. *sessionful service*), otrzymuje nową, dedykowaną instancję tej usługi (niezależną od jej pozostałych instancji). Tak utworzona instancja zwykle pozostaje aktywna do momentu, w którym klient nie będzie jej potrzebował. Ten tryb aktywacji (nazywany także **trybem sesji prywatnej** — ang. *private-session mode*) pod wieloma względami przypomina klasyczny model klient-serwer: każda sesja prywatna jest unikatowym powiązaniem pośrednika i zbioru kanałów łączących strony klienta i serwera z określoną instancją usługi, a konkretnie z jej kontekstem. Tryb tworzenia instancji dla sesji prywatnych wymaga stosowania sesji transportowej (to zagadnienie zostanie omówione w dalszej części tego podrozdziału).

Ponieważ instancja usługi pozostaje w pamięci przez cały czas istnienia sesji, może przechowywać swój stan w pamięci, zatem opisywany model programistyczny bardzo przypomina klasyczną architekturę klient-serwer. Oznacza to, że usługi sesyjne są narażone na te same problemy związane ze skalowalnością i przetwarzaniem transakcyjnym co klasyczny model klient-serwer. Z powodu kosztów utrzymywania dedykowanych instancji usługa skonfigurowana z myślą o obsłudze sesji prywatnych zwykle nie może obsługiwać więcej niż kilkadziesiąt (w niektórych przypadkach maksymalnie sto lub dwieście) jednocześnie działających klientów.

Sesja klienta jest punktem końcowym konkretnej sesji utworzonym dla określonego pośrednika. Jeśli więc klient utworzy innego pośrednika dla tego samego lub innego punktu końcowego, drugi pośrednik zostanie powiązany z nową instancją usługi i nową sesją.

Konfiguracja sesji prywatnych

Istnieją trzy elementy wspomagające obsługę sesji: zachowanie, powiązanie i kontrakt. Zachowanie jest wymagane do utrzymywania przez środowisko WCF kontekstu instancji usługi przez cały czas trwania sesji oraz do kierowania do tego kontekstu komunikatów przysyłanych przez klienta. Konfiguracja zachowania lokalnego wymaga przypisania wartości `InstanceContextMode.PerSession` do właściwości `InstanceContextMode` atrybutu `ServiceBehavior`:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
class MyService : IMyContract
{...}
```

Ponieważ InstanceContextMode.PerSession jest domyślną wartością właściwości InstanceContext ↪Mode, poniższe definicje są równoważne:

```
class MyService : IMyContract
{...}

[ServiceBehavior]
class MyService : IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
class MyService : IMyContract
{...}
```

Sesja zwykle kończy się w momencie, w którym klient zamyka swojego pośrednika — pośrednik powiadamia wówczas usługę o zakończeniu bieżącej sesji. Jeśli usługa obsługuje interfejs IDisposable, metoda Dispose() zostanie wywołana asynchronicznie względem działania klienta. W takim przypadku metoda Disposed() zostanie wykonana przez wątek roboczy pozbawiony kontekstu operacji.

Aby prawidłowo skojarzyć wszystkie komunikaty od określonego klienta z konkretną instancją usługi, środowisko WCF musi mieć możliwość identyfikacji tego klienta. Jak wyjaśniono w rozdziale 1., właśnie do tego służy sesja transportowa. Jeśli usługa ma być projektowana z myślą o działaniu w roli usługi sesyjnej, musi istnieć sposób wyrażania tego założenia na poziomie kontraktu. Odpowiedni element kontraktu powinien wykraczać poza ograniczenia samej usługi, ponieważ także środowisko wykonawcze WCF po stronie klienta musi wiedzieć o konieczności stosowania sesji. W tym celu atrybut ServiceContract udostępnia właściwość SessionMode typu wyliczeniowego SessionMode:

```
public enum SessionMode
{
   Allowed,
   Required,
   NotAllowed
}
[AttributeUsage(AttributeTargets.Interface|AttributeTargets.Class,
                Inherited=false)]
public sealed class ServiceContractAttribute : Attribute
{
   public SessionMode SessionMode
   {get;set;}
   // Pozostałe składowe…
}
```

Domyślną wartością wyliczenia SessionMode jest SessionMode.Allowed. Skonfigurowana wartość typu SessionMode jest dołączana do metadanych usługi i prawidłowo uwzględniana w momencie importowania metadanych kontraktu przez klienta. Wartość typu wyliczeniowego SessionMode nie ma nic wspólnego z sesją usługi — bardziej adekwatną nazwą tego typu byłaby Transport ↪SessionMode, ponieważ dotyczy sesji transportowej, nie sesji logicznej utrzymywanej pomiędzy klientem a instancją usługi.

Wartość SessionMode.Allowed

SessionMode.Allowed jest domyślną wartością właściwości SessionMode, zatem obie poniższe definicje są sobie równoważne:

```
[ServiceContract]
interface IMyContract
{...}

[ServiceContract(SessionMode = SessionMode.Allowed)]
interface IMyContract
{...}
```

Możliwość skonfigurowania kontraktu w punkcie końcowym z wartością `SessionMode.Allowed` jest obsługiwana przez wszystkie powiązania. Przypisanie tej wartości do właściwości `SessionMode` oznacza, że sesje transportowe są dopuszczalne, ale nie są wymagane. Ostateczny kształt zachowania jest pochodną konfiguracji usługi i stosowanego powiązania. Jeśli usługę skonfigurowano z myślą o aktywacji przez wywołania, zachowuje się właśnie w ten sposób (patrz przykład z listingu 4.2). Jeśli usługę skonfigurowano z myślą o aktywacji na czas trwania sesji, będzie zachowywała się jak usługa sesyjna, pod warunkiem że użyte powiązanie utrzymuje sesję transportową. Na przykład powiązanie `BasicHttpBinding` nigdy nie może dysponować sesją na poziomie transportowym z racji bezpołączeniowego charakteru protokołu HTTP. Utrzymywanie sesji na poziomie transportowym nie jest możliwe także w przypadku powiązania `WSHttpBinding` bez mechanizmu zabezpieczania komunikatów. W obu przypadkach mimo skonfigurowania usługi z wartością `InstanceContextMode.PerSession` i kontraktu z wartością `SessionMode.Allowed` usługa będzie zachowywała się tak jak usługi aktywowane przez wywołania.

Jeśli jednak zostanie użyte powiązanie `WSHttpBinding` z mechanizmem zabezpieczenia komunikatów (czyli w domyślnej konfiguracji) lub z niezawodnym protokołem przesyłania komunikatów albo powiązanie `NetTcpBinding` lub `NetNamedPipeBinding`, usługa będzie zachowywała się jak usługa sesyjna. Jeśli na przykład użyjemy powiązania `NetTcpBinding`, tak skonfigurowana usługa będzie zachowywała się jak usługa sesyjna:

```
[ServiceContract]
interface IMyContract
{...}

class MyService : IMyContract
{...}
```

Łatwo zauważyć, że w powyższym fragmencie wykorzystano wartości domyślne zarówno właściwości `SessionMode`, jak i właściwości `InstanceContextMode`.

Wartość SessionMode.Required

Wartość `SessionMode.Required` wymusza stosowanie sesji transportowej, ale nie wymusza używania sesji na poziomie aplikacji. Oznacza to, że nie jest możliwe skonfigurowanie kontraktu z wartością `SessionMode.Required` dla punktu końcowego, którego powiązanie nie utrzymuje sesji transportowej — to ograniczenie jest sprawdzane na etapie ładowania usługi. Okazuje się jednak, że można tak skonfigurować tę usługę, aby była aktywowana przez wywołania, czyli aby jej instancje były tworzone i niszczone osobno dla każdego wywołania ze strony klienta. Tylko skonfigurowanie usługi jako usługi sesyjnej spowoduje, że instancja usługi będzie istniała przez cały czas trwania sesji klienta:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{...}

class MyService : IMyContract
{...}
```

Podczas projektowania kontraktu sesyjnego zaleca się jawnie użyć wartości `SessionMode.Required`, zamiast liczyć na zastosowanie domyślnej wartości `SessionMode.Allowed`. W pozostałych przykładach usług sesyjnych prezentowanych w tej książce wartość `SessionMode.Required` będzie jawnie stosowana w zapisach konfiguracyjnych.

Na listingu 4.4 pokazano kod tej samej usługi i tego samego klienta co na wcześniejszym listingu 4.2 — z tą różnicą, że tym razem kontrakt i usługę skonfigurowano w sposób wymuszający stosowanie sesji prywatnej. Jak widać w danych wynikowych, klient dysponuje własną, dedykowaną instancją.

Listing 4.4. Usługa sesyjna i jej klient

```
////////////////////////// Kod usługi ///////////////////////
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
   [OperationContract]
   void MyMethod();
}
class MyService : IMyContract,IDisposable
{
   int m_Counter = 0;

   MyService()
   {
      Trace.WriteLine("MyService.MyService()");
   }
   public void MyMethod()
   {
      m_Counter++;
      Trace.WriteLine("Licznik = " + m_Counter);
   }
   public void Dispose()
   {
      Trace.WriteLine("MyService.Dispose()");
   }
}
////////////////////////// Kod klienta ///////////////////////
MyContractClient proxy = new MyContractClient();

proxy.MyMethod();
proxy.MyMethod();

proxy.Close();

// Dane wynikowe
MyService.MyService()
Licznik = 1
Licznik = 2
MyService.Dispose()
```

Wartość SessionMode.NotAllowed

Wartość `SessionMode.NotAllowed` uniemożliwia stosowanie sesji transportowej, wykluczając — tym samym — możliwość korzystania z sesji na poziomie aplikacji. Użycie tej wartości powoduje, że niezależnie od swojej konfiguracji usługa zachowuje się jak usługa aktywowana przez wywołania.

Ponieważ zarówno protokół TCP, jak i protokół IPC utrzymuje sesję na poziomie transportowym, nie można skonfigurować punktu końcowego usługi używającego powiązania `NetTcp↪Binding` lub `NetNamedPipeBinding`, aby udostępniał kontrakt oznaczony wartością `SessionMode.Not↪Allowed` (odpowiednie ograniczenie jest sprawdzane na etapie ładowania usługi). Okazuje się jednak, że stosowanie powiązania `WSHttpBinding` łącznie z emulowaną sesją transportową jest dopuszczalne. Aby poprawić czytelność pisanego kodu, zachęcam do dodatkowego konfigurowania usługi jako aktywowanej przez wywołania zawsze wtedy, gdy jest stosowana wartość `SessionMode.NotAllowed`:

```
[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{...}
```

Ponieważ powiązanie `BasicHttpBinding` nie może dysponować sesją na poziomie transportowym, punkty końcowe używające tego powiązania zawsze zachowują się tak, jakby ich kontrakty skonfigurowano z wartością `SessionMode.NotAllowed`. Sam traktuję wartość `SessionMode.NotAllowed` jako ustawienie przede wszystkim poprawiające kompletność dostępnych rozwiązań i z reguły nie używam jej w swoim kodzie.

Powiązania, kontrakty i zachowanie usługi

W tabeli 4.1 podsumowano tryby aktywowania instancji usługi jako pochodne stosowanego powiązania, obsługi sesji opisanej w kontrakcie oraz trybu obsługi kontekstu instancji skonfigurowanego w zachowaniu usługi. Tabela nie zawiera nieprawidłowych konfiguracji, na przykład połączenia wartości `SessionMode.Required` z powiązaniem `BasicHttpBinding`.

Tabela 4.1. Tryb aktywowania instancji jako pochodna powiązania, konfiguracji kontraktu i zachowania usługi

Powiązanie	Tryb sesji	Tryb kontekstu	Tryb instancji
Podstawowe	`Allowed/NotAllowed`	`PerCall/PerSession`	`PerCall`
TCP, IPC	`Allowed/Required`	`PerCall`	`PerCall`
TCP, IPC	`Allowed/Required`	`PerSession`	`PerSession`
WS (bez zabezpieczeń komunikatów i niezawodności)	`NotAllowed/Allowed`	`PerCall/PerSession`	`PerCall`
WS (z zabezpieczeniami komunikatów lub niezawodnością)	`Allowed/Required`	`PerSession`	`PerSession`
WS (z zabezpieczeniami komunikatów lub niezawodnością)	`NotAllowed`	`PerCall/PerSession`	`PerCall`

Spójna konfiguracja

Jeśli jeden kontrakt implementowany przez usługę jest sesyjny, wszystkie pozostałe kontrakty także powinny być sesyjne. Należy unikać mieszania kontraktów aktywowanych wywołaniami z kontraktami sesyjnymi dla tego samego typu usługi sesyjnej (mimo że środowisko WCF dopuszcza taką możliwość):

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{...}

[ServiceContract(SessionMode = SessionMode.NotAllowed)]
```

```
interface IMyOtherContract
{...}

// Tego należy unikać
class MyService : IMyContract,IMyOtherContract
{...}
```

Powód takiego rozwiązania jest dość oczywisty — usługi aktywowane przez wywołania muszą bezpośrednio zarządzać swoim stanem, zaś usługi sesyjne są zwolnione z tego obowiązku. Mimo że przytoczona para kontraktów będzie dostępna w dwóch różnych punktach końcowych i może być niezależnie wykorzystywana przez dwie różne aplikacje klienckie, łączne stosowanie dwóch trybów wymaga stosowania nieporęcznych zabiegów implementacyjnych w klasie usługi.

Sesje i niezawodność

Sesja łącząca klienta z instancją usługi może być niezawodna tylko na tyle, na ile jest niezawodna stosowana sesja transportowa. Oznacza to, że wszystkie punkty końcowe usługi implementującej kontrakt sesyjny powinny używać powiązań umożliwiających stosowanie niezawodnych sesji transportowych. Programista powinien się upewnić, że używane powiązania obsługują niezawodność i że ta niezawodność jest wprost zadeklarowana zarówno po stronie klienta, jak i po stronie użytkownika (programowo lub administracyjnie — patrz listing 4.5).

Listing 4.5. Włączanie niezawodności dla usług sesyjnych

```
<!—Konfiguracja hosta:—>
<system.serviceModel>
   <services>
      <service name = "MyPerSessionService">
         <endpoint
            address  = "net.tcp://localhost:8000/MyPerSessionService"
            binding  = "netTcpBinding"
            bindingConfiguration = "TCPSession"
            contract = "IMyContract"
         />
      </service>
   </services>
   <bindings>
      <netTcpBinding>
         <binding name = "TCPSession">
            <reliableSession enabled = "true"/>
         </binding>
      </netTcpBinding>
   </bindings>
</system.serviceModel>

<!—Konfiguracja klienta:—>
<system.serviceModel>
   <client>
      <endpoint
         address  = "net.tcp://localhost:8000/MyPerSessionService/"
         binding  = "netTcpBinding"
         bindingConfiguration = "TCPSession"
         contract = "IMyContract"
      />
   </client>
   <bindings>
      <netTcpBinding>
```

```
            <binding name = "TCPSession">
               <reliableSession enabled = "true"/>
            </binding>
         </netTcpBinding>
      </bindings>
   </system.serviceModel>
```

Jedynym wyjątkiem od tej reguły jest powiązanie IPC. Powiązanie IPC nie potrzebuje protokołu niezawodnego przesyłania komunikatów (wszystkie wywołania i tak są wykonywane na tym samym komputerze) i samo w sobie jest traktowane jako niezawodny mechanizm transportu danych.

Podobnie jak niezawodna sesja transportowa, także dostarczanie komunikatów w oryginalnej kolejności jest opcjonalne, a środowisko WCF prawidłowo utrzymuje sesję nawet w przypadku wyłączenia porządkowania komunikatów. Obsługa sesji aplikacji powoduje jednak, że klient usługi sesyjnej z reguły oczekuje zgodności kolejności dostarczania komunikatów z kolejnością ich wysyłania. Dostarczanie komunikatów w oryginalnej kolejności na szczęście jest domyślnie włączone, jeśli tylko włączono niezawodne sesje transportowe, zatem nie są potrzebne żadne dodatkowe ustawienia.

Identyfikator sesji

Każda sesja ma unikatowy identyfikator dostępny zarówno dla klienta, jak i dla usługi. Identyfikator sesji to w dużej części identyfikator GUID, którego można z powodzeniem używać podczas rejestrowania zdarzeń i diagnozowania systemu. Usługa może uzyskać dostęp do identyfikatora sesji za pośrednictwem tzw. **kontekstu wywołania operacji** (ang. *operation call context*), czyli zbioru właściwości (w tym identyfikatora sesji) używanych na potrzeby wywołań zwrotnych, tworzenia nagłówków komunikatów, zarządzania transakcjami, bezpieczeństwa, dostępu do hosta oraz dostępu do obiektu reprezentującego sam kontekst wykonywania. Każda operacja wykonywana przez usługę ma swój kontekst wywołania dostępny za pośrednictwem klasy `OperationContext`. Usługa może uzyskać referencję do kontekstu operacji właściwego bieżącej metodzie za pośrednictwem metody statycznej `Current` klasy `OperationContext`:

```
public sealed class OperationContext : ...
{
   public static OperationContext Current
   {get;set;}
   public string SessionId
   {get;}
}
```

Aby uzyskać dostęp do identyfikatora sesji, usługa musi odczytać wartość właściwości `SessionId`, która zwraca (w formie łańcucha) identyfikator niemal identyczny jak identyfikator GUID. W przypadku zawodnego powiązania TCP po identyfikatorze GUID następuje zwykły numer sesji z danym hostem:

```
string sessionID = OperationContext.Current.SessionId;
Trace.WriteLine(sessionID);
// Zapisuje identyfikator:
// uuid:8a0480da-7ac0-423e-9f3e-b2131bcbad8d;id=1
```

Próba uzyskania dostępu do właściwości `SessionId` przez usługę aktywowaną przez wywołania bez sesji transportowej spowoduje zwrócenie wartość `null`, ponieważ w takim przypadku nie istnieje ani sesja, ani — co oczywiste — jej identyfikator.

Klient może uzyskać dostęp do identyfikatora sesji za pomocą pośrednika. Jak wspomniano w rozdziale 1., klasą bazową dla pośrednika jest `ClientBase<T>`. Klasa `ClientBase<T>` definiuje właściwość dostępną tylko do odczytu `InnerChannel` typu `IClientChannel`. Interfejs `IClientChannel` dziedziczy po interfejsie `IContextChannel`, który z kolei zawiera właściwość `SessionId` zwracającą łańcuch z identyfikatorem sesji:

```
public interface IContextChannel : ...
{
   string SessionId
   {get;}
   // Pozostałe składowe…
}
public interface IClientChannel : IContextChannel,...
{...}
public abstract class ClientBase<T> : ...
{
   public IClientChannel InnerChannel
   {get;}
   // Pozostałe składowe…
}
```

Jeśli stosujemy definicje z listingu 4.4, klient może uzyskać identyfikator sesji w następujący sposób:

```
MyContractClient proxy = new MyContractClient();
proxy.MyMethod();

string sessionID = proxy.InnerChannel.SessionId;
Trace.WriteLine(sessionID);
```

Okazuje się jednak, że stopień zgodności identyfikatora sesji po stronie klienta z identyfikatorem zwracanym po stronie usługi (nawet wtedy, gdy klient ma dostęp do właściwości `SessionId`) jest pochodną stosowanego powiązania i jego konfiguracji. Za dopasowywanie identyfikatorów sesji po stronie klienta i po stronie usługi odpowiada sesja na poziomie transportowym. Jeśli jest stosowane powiązanie TCP i jeśli jest włączona niezawodna sesja (która w takim przypadku powinna być włączona), klient może uzyskać prawidłowy identyfikator sesji dopiero po wysłaniu do usługi pierwszego wywołania metody i — tym samym — ustanowieniu nowej sesji lub po bezpośrednim otwarciu pośrednika. W takim przypadku identyfikator sesji uzyskany przez klienta odpowiada identyfikatorowi stosowanemu po stronie usługi. (Jeśli klient uzyskuje dostęp do identyfikatora sesji przed pierwszym wywołaniem, właściwość `SessionId` ma wartość `null`). Jeśli jest stosowane powiązanie TCP, ale niezawodne sesje są wyłączone, klient może uzyskać dostęp do identyfikatora sesji przed pierwszym wywołaniem, jednak otrzymany identyfikator będzie inny od tego dostępnego po stronie usługi. W przypadku powiązania WS (przy włączonym mechanizmie niezawodnego przesyłania komunikatów) identyfikator sesji będzie miał wartość `null` do czasu zakończenia pierwszego wywołania (lub otwarcia pośrednika przez klienta). Po tym wywołaniu klient i usługa zawsze będą miały dostęp do tego samego identyfikatora sesji. W razie braku niezawodnego przesyłania komunikatów przed uzyskaniem dostępu do identyfikatora sesji klient musi użyć pośrednika (lub tylko go otworzyć); w przeciwnym razie istnieje ryzyko wystąpienia wyjątku `InvalidOperationException`. Po otwarciu pośrednika klient i usługa mają dostęp do tego samego, skorelowanego identyfikatora sesji. W przypadku powiązania IPC klient może uzyskać dostęp do właściwości `SessionId` przed pierwszym wywołaniem, jednak otrzymany w ten sposób identyfikator sesji będzie inny od tego dostępnego po stronie usługi. Podczas stosowania tego powiązania najlepszym rozwiązaniem jest całkowite ignorowanie identyfikatora sesji.

Kończenie sesji

Sesja zwykle kończy się w momencie, w którym klient zamyka swojego pośrednika. Jeśli jednak klient sam nie zamyka pośrednika, jeśli działanie klienta kończy się w jakiś nieoczekiwany sposób lub jeśli wystąpił jakiś problem komunikacyjny, sesja zostanie zakończona po określonym czasie nieaktywności sesji transportowej.

Usługa singletonowa

Usługa singletonowa (ang. *singleton service*) to w istocie usługa współdzielona. Skonfigurowanie usługi jako singletonu powoduje, że wszystkie aplikacje klienckie niezależnie od siebie łączą się z jednym, dobrze znanym kontekstem instancji i pośrednio z jedną instancją usługi (niezależnie od używanego punktu końcowego usługi). Singleton jest tworzony tylko raz (podczas tworzenia hosta) i nigdy nie jest niszczony. Zwolnienie singletonu następuje dopiero w momencie wyłączenia hosta.

Singleton udostępniany na serwerze IIS lub WAS jest tworzony z chwilą uruchamiania procesu hosta, czyli w odpowiedzi na pierwsze żądanie dotyczące dowolnej usługi w tym procesie. Okazuje się jednak, że w przypadku stosowania mechanizmu automatycznego uruchamiania pakietu Windows Server AppFabric singleton jest tworzony zaraz po uruchomieniu komputera. Zachowanie semantyki singletonu wymaga użycia albo mechanizmu samohostingu (ang. *self-hosting*), albo pakietu Windows Server AppFabric z funkcją automatycznego uruchamiania.

Stosowanie usługi singletonowej nie wymaga od klientów utrzymywania sesji logicznej z instancją tej usługi ani używania powiązań obsługujących sesje transportowe. Jeśli kontrakt pobrany przez klienta obejmuje sesję, do wywołania usługi singletonowej zostanie przypisany ten sam identyfikator sesji, którym dysponuje klient (jeśli stosowane powiązanie dopuszcza taką możliwość), ale zamknięcie pośrednika tego klienta spowoduje zakończenie tylko sesji transportowej — kontekst singletonu ani sama instancja usługi nie zostaną zamknięte. Jeśli usługa singletonowa obsługuje kontrakty bez sesji, także te kontrakty nie będą aktywowane przez wywołania, tylko trwale powiązane z tą samą instancją. Usługa singletonowa z natury rzeczy ma charakter współdzielony, a każdy klient powinien utworzyć własnego pośrednika lub własnych pośredników w dostępie do jedynej instancji tej usługi.

Konfiguracja usługi singletonowej wymaga ustawienia wartości `InstanceContextMode.Single` we właściwości `InstanceContextMode`:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class MySingleton : ...
{...}
```

Listing 4.6 demonstruje usługę singletonową z dwoma kontraktami, z których jeden wymaga sesji, drugi — nie. Przykład wywołania ze strony klienta pokazuje, że dwa wywołania dotyczące dwóch różnych punktów końcowych są kierowane do tej samej instancji, a zamknięcie pośredników nie powoduje zakończenia działania instancji usługi singletonowej.

Listing 4.6. Usługa singletonowa i jej klient

```
////////////////////////// Kod usługi //////////////////////
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
```

```
{
   [OperationContract]
   void MyMethod();
}
[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyOtherContract
{
   [OperationContract]
   void MyOtherMethod();
}
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
class MySingleton : IMyContract,IMyOtherContract,IDisposable
{
   int m_Counter = 0;

   public MySingleton()
   {
      Trace.WriteLine("MySingleton.MySingleton()");
   }
   public void MyMethod()
   {
      m_Counter++;
      Trace.WriteLine("Licznik = " + m_Counter);
   }
   public void MyOtherMethod()
   {
      m_Counter++;
      Trace.WriteLine("Licznik = " + m_Counter);
   }
   public void Dispose()
   {
      Trace.WriteLine("Singleton.Dispose()");
   }
}
////////////////////// Kod klienta ////////////////////
MyContractClient proxy1 = new MyContractClient();
proxy1.MyMethod();
proxy1.Close();

MyOtherContractClient proxy2 = new MyOtherContractClient();
proxy2.MyOtherMethod();
proxy2.Close();

// Dane wynikowe
MySingleton.MySingleton()
Licznik = 1
Licznik = 2
```

Inicjalizacja usługi singletonowej

W pewnych przypadkach tworzenie i inicjalizacja singletonu za pomocą konstruktora domyślnego nie jest najlepszym rozwiązaniem. Na przykład inicjalizacja stanu może wymagać wykonania jakichś niestandardowych kroków lub specjalistycznej wiedzy, która albo nie powinna obciążać klientów, albo w ogóle nie jest dla nich dostępna. Niezależnie od powodów programista może zdecydować o utworzeniu singletonu przy użyciu mechanizmu spoza hosta usług środowiska WCF. Z myślą o podobnych scenariuszach środowisko WCF umożliwia bezpośrednie utworzenie instancji usługi singletonowej za pomocą standardowego mechanizmu

tworzenia instancji klas w środowisku CLR. Tak utworzoną instancję można następnie zainicjalizować, po czym otworzyć host z tą instancją w roli usługi singletonowej. Klasa `ServiceHost` oferuje odpowiedni konstruktor, który otrzymuje na wejściu parametr typu `object`:

```
public class ServiceHost : ServiceHostBase,...
{
   public ServiceHost(object singletonInstance,params Uri[] baseAddresses);
   public object SingletonInstance
   {get;}
   // Pozostałe składowe…
}
```

Warto pamiętać, że przekazany parametr typu `object` musi być skonfigurowany jako singleton. Przeanalizujmy na przykład kod z listingu 4.7. Klasa `MySingleton` jest najpierw inicjalizowana, po czym umieszczana na hoście w formie usługi singletonowej.

Listing 4.7. Inicjalizacja usługi singletonowej i jej umieszczanie na hoście

```
// Kod usługi
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   void MyMethod();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class MySingleton : IMyContract
{
   public int Counter
   {get;set;}

   public void MyMethod()
   {
      Counter++;
      Trace.WriteLine("Licznik = " + Counter);
   }
}
// Kod hosta
MySingleton singleton = new MySingleton();
singleton.Counter = 287;

ServiceHost host = new ServiceHost(singleton);
host.Open();

// Kod klienta
MyContractClient proxy = new MyContractClient();
proxy.MyMethod();
proxy.Close();

// Dane wynikowe:
Licznik = 288
```

Jeśli programista decyduje się na inicjalizację singletonu i umieszczenie go na hoście w ten sposób, może być zainteresowany także bezpośrednim dostępem do tego singletonu z poziomu hosta. Środowisko WCF umożliwia obiektom uczestniczącym w łańcuchu wywołań bezpośredni dostęp do singletonu za pomocą właściwości `SingletonInstance` klasy `ServiceHost`. Każdy element łańcucha wywołań łączącego kolejne elementy, począwszy od oryginalnego wywołania operacji singletonu, ma dostęp do hosta za pośrednictwem właściwości `Host` (dostępnej tylko do odczytu) kontekstu operacji:

```
public sealed class OperationContext : ...
{
   public ServiceHostBase Host
   {get;}
   // Pozostałe składowe…
}
```

Po uzyskaniu referencji do singletonu dalsze działania można podejmować bezpośrednio na tym obiekcie:

```
ServiceHost host = OperationContext.Current.Host as ServiceHost;
Debug.Assert(host != null);
MySingleton singleton = host.SingletonInstance as MySingleton;
Debug.Assert(singleton != null);
singleton.Counter = 388;
```

Jeśli host nie dysponuje żadną instancją usługi singletonowej, właściwość `SingletonInstance` zwraca wartość `null`.

Usprawnienie przy użyciu klasy ServiceHost<T>

Klasę `ServiceHost<T>`, którą przedstawiono w rozdziale 1., można uzupełnić o kod zapewniający inicjalizację singletonu i dostęp do jego instancji (z zachowaniem bezpieczeństwa typów):

```
public class ServiceHost<T> : ServiceHost
{
   public ServiceHost(T singleton,params Uri[] baseAddresses) :
                                                   base(singleton,baseAddresses)
   {}
   public virtual T Singleton
   {
      get
      {
         if(SingletonInstance == null)
         {
            return default(T);
         }
         return (T)SingletonInstance;
      }
   }
   // Pozostałe składowe…
}
```

Parametr typu zapewnia powiązanie gwarantujące bezpieczeństwo typów dla obiektu przekazanego na wejściu konstruktora:

```
MySingleton singleton = new MySingleton();
singleton.Counter = 287;

ServiceHost<MySingleton> host = new ServiceHost<MySingleton>(singleton);
host.Open();
```

i obiektu zwróconego przez właściwość `Singleton`:

```
ServiceHost<MySingleton> host = OperationContext.Current.Host
                                                    as ServiceHost<MySingleton>;

Debug.Assert(host != null);
host.Singleton.Counter = 388;
```

Także klasę `InProcFactory<T>` (wprowadzoną w rozdziale 1.) można w podobny sposób uzupełnić o kod inicjalizujący instancję singletonu.

Wybór singletonu

Usługa singletonowa jest zadeklarowanym wrogiem skalowalności. Powodem tej „wrogości" jest synchronizacja stanu usługi singletonowej, nie koszt utrzymania jej jedynej instancji. Stosowanie singletonu wynika z potrzeby użycia pojedynczej instancji reprezentującej pewien cenny stan, który ma być współdzielony przez wszystkie aplikacje klienckie. Problem w tym, że jeśli stan singletonu jest zmienny i jeśli wiele klientów łączy się z jedyną instancją tej usługi, wszystkie aplikacje klienckie mogą to robić jednocześnie, a generowane przez nie wywołania są obsługiwane przez wiele wątków roboczych. Oznacza to, że singleton musi synchronizować dostęp do swojego stanu, aby uniknąć jego uszkodzenia. To z kolei powoduje, że w danym momencie dostęp do singletonu może mieć tylko jeden klient. Wspomniane ograniczenie może znacznie obniżyć przepustowość, wydłużyć czas odpowiedzi i zmniejszyć dostępność singletonu. W tej sytuacji już w systemach średniej wielkości singleton może stać się po prostu bezużyteczny. Jeśli na przykład pojedyncza operacja wykonywana przez usługę singletonową zajmuje jedną dziesiątą sekundy, singleton może obsłużyć zaledwie dziesięć klientów w ciągu sekundy. W przypadku większej liczby klientów (na przykład dwudziestu czy stu) wydajność całego systemu drastycznie spadnie.

Ogólnie usługi singletonowe należy stosować tylko wtedy, gdy odwzorowują naturalne singletony występujące w dziedzinie aplikacji. **Naturalny singleton** to zasób, który z natury rzeczy występuje pojedynczo i jest niepowtarzalny. Przykładami naturalnych singletonów są globalne dzienniki zdarzeń, w których wszystkie usługi rejestrują swoje działania, pojedyncze porty komunikacyjne czy pojedyncze silniki mechaniczne. Stosowania singletonów należy unikać, jeśli istnieje choćby cień szansy na obsługę większej liczby instancji danej usługi w przyszłości (na przykład poprzez dodanie kolejnego silnika lub drugiego portu komunikacyjnego). Powód jest jasny — skoro wszystkie aplikacje klienckie będą początkowo zależały od połączenia z jedną instancją i skoro z czasem będzie dostępna druga instancja danej usługi, aplikacje będą potrzebowały możliwości nawiązania połączenia z właściwą instancją. Konieczność obsługi większej liczby instancji ma zasadniczy wpływ na stosowany model programowania aplikacji. Z powodu tych ograniczeń radzę unikać singletonów w ogólnych przypadkach i poszukiwać sposobów współdzielenia raczej stanu singletonu, a nie jego instancji. Jak już wspomniano, w pewnych przypadkach stosowanie jest w pełni uzasadnione.

Operacje demarkacyjne

W pewnych przypadkach kontrakty stanowe niejawnie zakładają, że wywołania operacji będą następowały w określonej kolejności. Niektóre operacje nie mogą być wywoływane jako pierwsze, inne muszą być wywoływane jako ostatnie. Przeanalizujmy na przykład kontrakt używany do zarządzania zamówieniami klientów:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IOrderManager
{
   [OperationContract]
```

```
    void SetCustomerId(int customerId);

    [OperationContract]
    void AddItem(int itemId);

    [OperationContract]
    decimal GetTotal();

    [OperationContract]
    bool ProcessOrders();
}
```

Kontrakt przewiduje następujące ograniczenia: w pierwszej operacji w ramach sesji aplikacja kliencka musi przekazać identyfikator nabywcy (w przeciwnym razie nie można wykonać pozostałych operacji); w kolejnych operacjach można dodać do zamówienia produkty; usługa oblicza łączną wartość zamówienia na każde żądanie klienta; ostateczne przetworzenie zamówienia kończy sesję, zatem musi być wykonane jako ostatnie. W klasycznym frameworku .NET wymagania tego typu często wymuszają na programistach stosowanie maszyny stanów lub flag stanów oraz weryfikacji bieżącego stanu przy okazji każdej operacji.

Okazuje się jednak, że w środowisku WCF projektanci kontraktów mają możliwość wyznaczania operacji, które mogą rozpoczynać bądź kończyć sesje (lub nie mogą tego robić). Odpowiednie ustawienia można definiować za pomocą właściwości `IsInitiating` i `IsTerminating` atrybutu `OperationContract`:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute,...
{
   public bool IsInitiating
   {get;set;}
   public bool IsTerminating
   {get;set;}
   // Pozostałe składowe…
}
```

Wymienione właściwości mogą być używane do izolowania granic sesji; sam określam tę technikę mianem **operacji demarkacyjnych** (ang. *demarcating operations*). Jeśli w czasie ładowania usługi (lub w czasie stosowania pośrednika po stronie klienta) te dwie właściwości mają przypisane wartości inne niż domyślne, środowisko WCF sprawdza, czy operacje demarkacyjne wchodzą w skład kontraktu wymuszającego stosowanie sesji (tj. czy właściwość `SessionMode` ma wartość `SessionMode.Required`); jeśli nie, środowisko zgłasza wyjątek `InvalidOperationException`. Zarówno usługi sesyjne, jak i usługi singletonowe mogą implementować kontrakty używające operacji demarkacyjnych do zarządzania sesjami klientów.

Wartościami domyślnymi właściwości `IsInitiating` i `IsTerminating` są odpowiednio `true` i `false`. Oznacza to, że następujące dwie definicje są sobie równoważne:

```
[OperationContract]
void MyMethod();

[OperationContract(IsInitiating = true,IsTerminating = false)]
void MyMethod();
```

Jak widać, obie te właściwości można ustawić dla tej samej metody. Operacje domyślnie nie izolują granic sesji — mogą być wywoływane jako pierwsze, ostatnie lub pomiędzy dowolnymi innymi operacjami w ramach swojej sesji. Użycie wartości innych niż domyślne pozwala określić, że dana metoda nie może być wywoływana jako pierwsza, musi być wywoływana jako ostatnia lub powinna spełniać oba te warunki jednocześnie:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
   [OperationContract]
   void StartSession();

   [OperationContract(IsInitiating = false)]
   void CannotStart();

   [OperationContract(IsTerminating = true)]
   void EndSession();

   [OperationContract(IsInitiating = false,IsTerminating = true)]
   void CannotStartCanEndSession();
}
```

Wróćmy teraz do przykładu kontraktu na potrzeby zarządzania zamówieniami — operacji demarkacyjnych można użyć do wymuszania pewnych ograniczeń interakcji:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IOrderManager
{
   [OperationContract]
   void SetCustomerId(int customerId);

   [OperationContract(IsInitiating = false)]
   void AddItem(int itemId);

   [OperationContract(IsInitiating = false)]
   decimal GetTotal();

   [OperationContract(IsInitiating = false,IsTerminating = true)]
   bool ProcessOrders();
}
// Kod klienta
OrderManagerClient proxy = new OrderManagerClient();

proxy.SetCustomerId(123);
proxy.AddItem(4);
proxy.AddItem(5);
proxy.AddItem(6);
proxy.ProcessOrders();

proxy.Close();
```

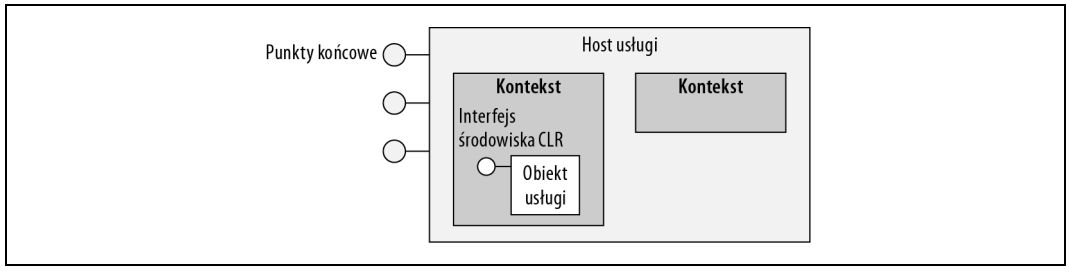
Jeśli właściwość `IsInitiating` ma wartość `true` (czyli swoją wartość domyślną), odpowiednia operacja albo rozpocznie nową sesję, jeśli będzie pierwszą metodą wywołaną przez klienta, albo będzie częścią trwającej sesji, jeśli wcześniej wywołano inną operację. Jeśli właściwość `IsInitiating` ma wartość `false`, klient nigdy nie może wywołać danej metody jako pierwszej operacji nowej sesji, zatem metoda ta może być wywołana tylko w ramach trwającej sesji.

Jeśli właściwość `IsTerminating` ma wartość `false` (czyli swoją wartość domyślną), sesja nie kończy się po zwróceniu sterowania przez daną operację. Jeśli właściwość `IsTerminating` ma wartość `true`, sesja kończy się wraz ze zwróceniem sterowania przez odpowiednią metodę, a środowisko WCF asynchronicznie zwalnia instancję danej usługi. Klient nie będzie mógł przekazać żadnych dodatkowych wywołań do odpowiedniego pośrednika. Warto przy tym pamiętać, że klient wciąż ma obowiązek zamknięcia tego pośrednika.

Po wygenerowaniu pośrednika dla usługi korzystającej z operacji demarkacyjnych zaimportowana definicja kontraktu obejmuje ustawienia właściwości. Środowisko WCF wymusza izolowanie granic sesji osobno po stronie klienta i osobno po stronie usługi, zatem w praktyce oba zadania mogą być realizowane niezależnie od siebie.

Dezaktywacja instancji

Na poziomie koncepcyjnym technika zarządzania instancjami usługi sesyjnej (przynajmniej w dotychczas opisanej formie) łączy klienta (lub wiele aplikacji klienckich) z instancją usługi. Okazuje się jednak, że działanie tego mechanizmu jest bardziej złożone. W rozdziale 1. wspomniano, że każda instancja usługi należy do pewnego kontekstu (patrz rysunek 4.2).



Rysunek 4.2. Konteksty i instancje

W praktyce zadaniem sesji jest kojarzenie komunikatów wysyłanych przez aplikacje klienckie nie tyle z instancjami, co z hostami zawierającymi tę instancję. W momencie rozpoczęcia sesji host tworzy nowy kontekst. Z chwilą zakończenia sesji kończy się także ten kontekst. Czas życia kontekstu jest więc taki sam jak czas życia należącej do niego instancji. Okazuje się jednak, że aby poprawić optymalizację i rozszerzalność, technologia WCF umożliwia projektantom usług rozdzielanie tych dwóch cykli życia i dezaktywację instancji niezależnie od jej kontekstu. W rzeczywistości technologia WCF dopuszcza nawet istnienie kontekstu bez jakiejkolwiek powiązanej instancji usługi (patrz rysunek 4.2). Opisaną technikę zarządzania instancjami określam mianem **dezaktywacji kontekstu** (ang. *context deactivation*). Typowym sposobem sterowania procesem dezaktywacji kontekstu jest korzystanie z pośrednictwa właściwości `Release` ↪`InstanceMode` atrybutu `OperationBehavior`:

```
public enum ReleaseInstanceMode
{
   None,
   BeforeCall,
   AfterCall,
   BeforeAndAfterCall
}
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute,...
{
   public ReleaseInstanceMode ReleaseInstanceMode
   {get;set;}
   // Pozostałe składowe…
}
```

Właściwość `ReleaseInstanceMode` jest egzemplarzem typu wyliczeniowego `ReleaseInstanceMode`. Poszczególne wartości typu `ReleaseInstanceMode` sterują czasem zwalniania instancji względem

wywołania metody: przed, po, przed i po lub wcale. Jeśli dana usługa obsługuje interfejs IDispo
↪sable, podczas zwalniania instancji tej usługi jest wywoływana metoda Dispose(), która dysponuje kontekstem operacji.

Dezaktywację instancji zwykle stosuje się tylko dla wybranych metod usługi lub dla wielu różnych metod z odmiennymi ustawieniami właściwości ReleaseInstanceMode:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
   [OperationContract]
   void MyMethod();

   [OperationContract]
   void MyOtherMethod();
}
class MyService : IMyContract,IDisposable
{
   [OperationBehavior(ReleaseInstanceMode = ReleaseInstanceMode.AfterCall)]
   public void MyMethod()
   {...}
   public void MyOtherMethod()
   {...}
   public void Dispose()
   {...}
}
```

Opisane rozwiązanie jest stosowane sporadycznie, ponieważ jego spójne wykorzystywanie wymagałoby tworzenia usług zbliżonych do tych aktywowanych przez wywołania — w takim przypadku równie dobrze można po prostu posłużyć się tym trybem aktywacji.

Jeśli mechanizm dezaktywacji instancji wymaga określonego porządku wywołań, warto spróbować wymusić stosowanie tej kolejności przy użyciu operacji demarkacyjnych.

Konfiguracja z wartością ReleaseInstanceMode.None

Wartością domyślną właściwości ReleaseInstanceMode jest ReleaseInstanceMode.None, zatem poniższe definicje są sobie równoważne:

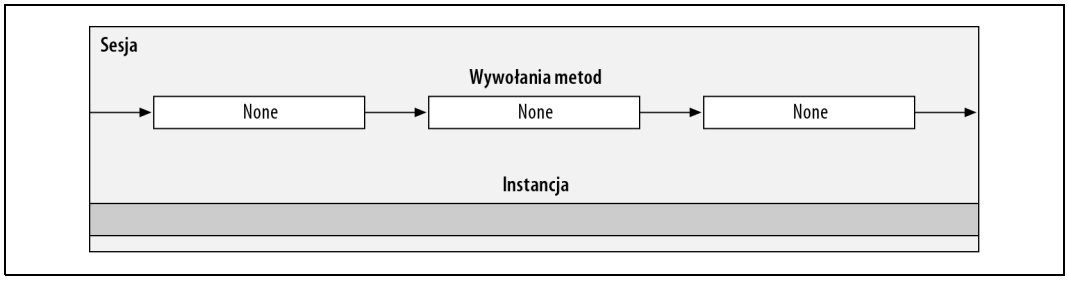
```
[OperationBehavior(ReleaseInstanceMode = ReleaseInstanceMode.None)]
public void MyMethod()
{...}

public void MyMethod()
{...}
```

Wartość ReleaseInstanceMode.None oznacza, że wywołanie nie wpływa na czas życia instancji (patrz rysunek 4.3).

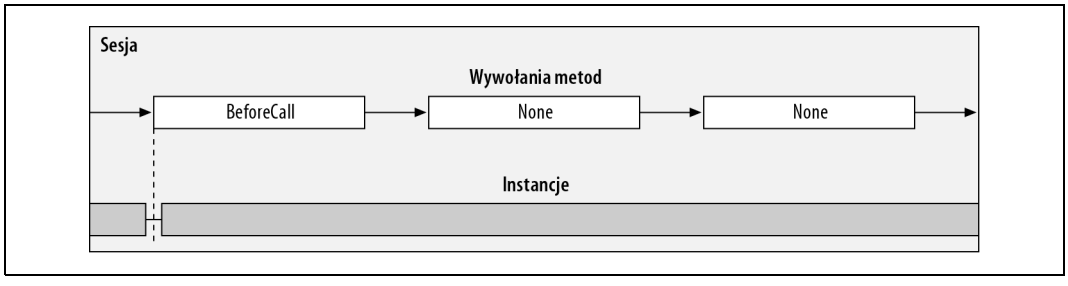
Konfiguracja z wartością ReleaseInstanceMode.BeforeCall

Jeśli skonfigurowano metodę z wartością ReleaseInstanceMode.BeforeCall i jeśli istnieje jakaś instancja w ramach sesji, przed przekazaniem wywołania środowisko dezaktywuje tę instancję i tworzy w jej miejsce nową, która obsługuje to wywołanie (patrz rysunek 4.4).

Środowisko WCF dezaktywuje instancje i wywołuje metodę Dispose() przed zakończeniem wywołania w wątku obsługującym wywołanie przychodzące (w tym czasie wykonywanie kodu



Rysunek 4.3. Czas życia instancji w przypadku metod skonfigurowanych z wartością ReleaseInstanceMode.None

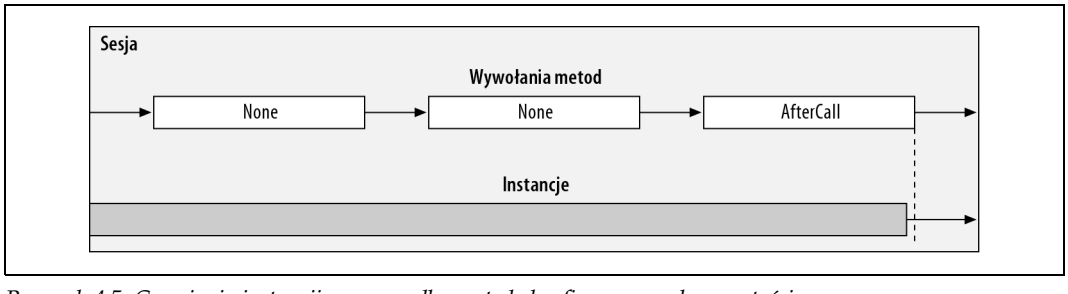


Rysunek 4.4. Czas życia instancji w przypadku metod skonfigurowanych z wartością ReleaseInstanceMode.BeforeCall

klienta jest zablokowane). Takie rozwiązanie gwarantuje, że dezaktywacja instancji rzeczywiście zakończy się przed rozpoczęciem wywołania, nie równolegle do jego realizacji. Wartość `ReleaseInstanceMode.BeforeCall` zaprojektowano z myślą o optymalizacji takich metod jak `Create()`, które uzyskują cenne zasoby, ale też powinny każdorazowo zwalniać wcześniej zajęte zasoby. Zamiast uzyskiwać zasoby w momencie rozpoczęcia sesji, usługa czeka na wywołanie metody `Create()`, która nie tylko uzyskuje nowe zasoby, ale też zwalnia te przydzielone wcześniej. Po wywołaniu metody `Create()` usługa jest gotowa do obsługi pozostałych metod danej instancji, które zwykle konfiguruje się przy użyciu wartości `ReleaseInstanceMode.None`.

Konfiguracja z wartością ReleaseInstanceMode.AfterCall

Jeśli skonfigurowano metodę z wartością `ReleaseInstanceMode.AfterCall`, środowisko WCF dezaktywuje instancję po wywołaniu tej metody (patrz rysunek 4.5).

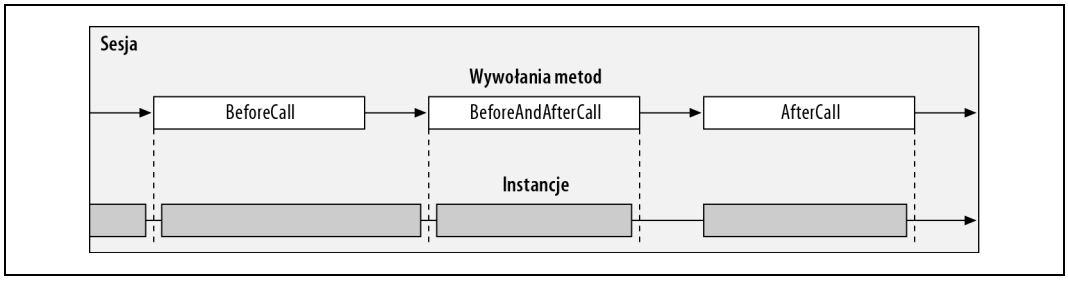


Rysunek 4.5. Czas życia instancji w przypadku metod skonfigurowanych z wartością ReleaseInstanceMode.AfterCall

Tę wartość zaprojektowano z myślą o optymalizacji takich metod jak `Cleanup()`, które zwalniają cenne zasoby zajmowane przez instancję, nie czekając na zakończenie odpowiedniej sesji. Wartość `ReleaseInstanceMode.AfterCall` zwykle stosuje się dla metod wywoływanych po metodach skonfigurowanych z wartością `ReleaseInstanceMode.None`.

Konfiguracja z wartością ReleaseInstanceMode.BeforeAndAfterCall

Jak nietrudno się domyślić, skonfigurowanie metody z wartością `ReleaseInstanceMode.BeforeAnd↪AfterCall` umożliwia połączenie skutków użycia wartości `ReleaseInstanceMode.BeforeCall` i `ReleaseInstanceMode.AfterCall`. Jeśli przed wywołaniem tak skonfigurowanej metody kontekst zawiera instancję usługi, bezpośrednio przed przekazaniem tego wywołania środowisko WCF dezaktywuje tę instancję i tworzy nową na potrzeby tego wywołania. Nowa instancja jest dezaktywowana zaraz po zakończeniu wywołania (patrz rysunek 4.6).



Rysunek 4.6. Czas życia instancji w przypadku metod skonfigurowanych z wartością ReleaseInstanceMode.BeforeAndAfterCall

Na pierwszy rzut oka metoda `ReleaseInstanceMode.BeforeAndAfterCall` może sprawiać wrażenie nadmiarowej, ale w rzeczywistości stanowi cenne uzupełnienie pozostałych wartości. Wartość `ReleaseInstanceMode.BeforeAndAfterCall` stosuje się dla metod wywoływanych po metodach oznaczonych wartością `ReleaseInstanceMode.BeforeCall` lub `None` bądź przed metodami oznaczonymi wartością `ReleaseInstanceMode.AfterCall` lub `None`. Wyobraźmy sobie sytuację, w której usługa sesyjna ma korzystać z zachowania stanowego (podobnie jak usługa aktywowana przez wywołania) i jednocześnie zajmować zasoby tylko w czasie, gdy są rzeczywiście potrzebne (aby zoptymalizować zarządzanie zasobami i zapewnić bezpieczeństwo). Gdyby jedyną dostępną opcją była wartość `ReleaseInstanceMode.BeforeCall`, zasoby byłyby niepotrzebnie zajmowane przez ten obiekt przez pewien czas po zakończeniu wywołania. Podobna sytuacja miałaby miejsce, gdyby jedyną dostępną opcją była wartość `ReleaseInstanceMode.AfterCall` — wówczas zasoby byłyby niepotrzebnie zajmowane przez pewien czas przed wywołaniem tak skonfigurowanej metody.

Bezpośrednia dezaktywacja

Decyzji o tym, które metody mają dezaktywować instancję usługi, nie trzeba podejmować na etapie projektowania systemu — równie dobrze można ją podjąć w czasie wykonywania, po zwróceniu sterowania przez odpowiednią metodę. Wystarczy wywołać metodę `ReleaseService↪Instance()` dla obiektu kontekstu instancji. Obiekt kontekstu instancji można uzyskać za pośrednictwem właściwości `InstanceContext` kontekstu operacji:

```
public sealed class InstanceContext : ...
{
   public void ReleaseServiceInstance();
   // Pozostałe składowe…
}
public sealed class OperationContext : ...
{
   public InstanceContext InstanceContext
   {get;}
   // Pozostałe składowe…
}
```

Listing 4.8 zawiera przykład użycia techniki bezpośredniej (jawnej) dezaktywacji w implementacji niestandardowej techniki zarządzania instancjami zależnie od wartości licznika.

Listing 4.8. Przykład użycia metody ReleaseServiceInstance()

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
   [OperationContract]
   void MyMethod();
}
class MyService : IMyContract,IDisposable
{
   int m_Counter = 0;

   public void MyMethod()
   {
      m_Counter++;
      if(m_Counter > 4)
      {
         OperationContext.Current.InstanceContext.ReleaseServiceInstance();
      }
   }
   public void Dispose()
   {...}
}
```

Wywołanie metody `ReleaseServiceInstance()` daje podobny efekt do użycia wartości `Release` ↪`InstanceMode.AfterCall`. Wywołanie tej metody wewnątrz metody oznaczonej wartością `Release` ↪`InstanceMode.BeforeCall` spowoduje działanie podobne do użycia samej wartości `ReleaseInstance` ↪`Mode.BeforeAndAfterCall`.

Dezaktywacja usługi wpływa także na sposób działania singletonu, jednak łączenie obu rozwiązań nie ma większego sensu — instancja usługi singletonowej z natury rzeczy nie musi i nie powinna być dezaktywowana.

Stosowanie dezaktywacji instancji

Dezaktywacja instancji to jedna z technik optymalizacji, której — jak wszystkich technik optymalizacji — nie należy stosować we wszystkich przypadkach. Dezaktywacja instancji wprowadza dodatkową złożoność do aplikacji i utrudnia konserwację kodu programistom, którzy nie są ekspertami w dziedzinie technologii WCF. Stosowanie tej techniki należy rozważać tylko wtedy, gdy inne sposoby nie wystarczą do osiągnięcia zakładanej wydajności i skalowalności oraz gdy uważna analiza i profilowanie kodu ostatecznie dowiodły, że dezaktywacja instancji

rzeczywiście poprawi sytuację. W razie problemów z niedostateczną skalowalnością i przepustowością warto skorzystać raczej z prostoty trybu aktywowania usługi przez wywołania, unikając dezaktywacji instancji. Opisuję tę technikę przede wszystkim dlatego, że samo środowisko WCF często stosuje mechanizm dezaktywacji instancji; znajomość tej techniki jest więc niezbędna do zrozumienia wielu innych aspektów tej technologii, w tym usług trwałych i transakcji.

Usługi trwałe

Warto przeanalizować przypadek, w którym jakiś proces biznesowy lub system przepływu pracy (składający się z wielu sekwencji wykonywania) trwa wiele dni lub nawet tygodni.

Używam terminu **przepływ pracy** (ang. *workflow*) w kontekście ogólnego, biznesowego przepływu pracy, niekoniecznie przepływu obsługiwanego przez produkt Windows Workflow czy powiązanego z tym produktem.

Takie długotrwałe procesy mogą współpracować z klientami (lub wręcz użytkownikami końcowymi) łączącymi się z aplikacją, wykonującymi określone, skończone zadania, zmieniającymi stan przepływu pracy i rozłączającymi się na nieznany okres przed nawiązaniem kolejnego połączenia (i dalszym wpływaniem na stan przepływu pracy). Aplikacje klienckie mogą w pewnym momencie zdecydować o zakończeniu bieżącego przepływu pracy i rozpoczęciu nowego. Przepływ pracy może zostać zakończony także przez usługę, która go obsługuje. Utrzymywanie pośredników i usług w pamięci w oczekiwaniu na wywołania ze strony klientów nie miałoby oczywiście większego sensu. Taki model z pewnością nie przetrwałby próby czasu; połączenia byłyby zrywane choćby z powodu upływających limitów czasowych, a ponowne uruchamianie lub wylogowywanie komputerów po obu stronach połączenia z pewnością nie byłoby łatwe. Konieczność stosowania niezależnych cyklów życia klientów i usług jest szczególnie ważna w przypadku długotrwałych procesów biznesowych, ponieważ bez tej niezależności nie sposób umożliwić klientom nawiązywanie połączenia, wykonywanie pewnych zadań związanych z przepływem pracy i rozłączanie się. Po stronie hosta z czasem może zaistnieć potrzeba kierowania wywołań do różnych komputerów.

W przypadku długotrwałych usług właściwym rozwiązaniem jest unikanie utrzymywania stanu usługi w pamięci. Każde wywołanie powinno być obsługiwane przez nową instancję dysponującą własnym stanem (przechowywanym w pamięci). Na potrzeby każdej operacji usługa powinna uzyskiwać swój stan z jakiejś pamięci trwałej (na przykład pliku lub bazy danych), wykonywać żądaną jednostkę pracy, po czym (na zakończenie obsługi wywołania) zapisywać stan w odpowiedniej pamięci trwałej. Usługi działające według tego modelu określa się mianem **usług trwałych**. Ponieważ używana pamięć trwała może być współdzielona przez wiele komputerów, stosowanie usług trwałych dodatkowo stwarza możliwość rozdzielania wywołań pomiędzy różne komputery z myślą o skalowalności, nadmiarowości lub na potrzeby konserwacji.

Usługi trwałe i tryby zarządzania instancjami

Model zarządzania stanem obowiązujący w usługach trwałych pod wieloma względami przypomina zaproponowane wcześniej rozwiązania dla usług aktywowanych przez wywołania, które także aktywnie zarządzają swoim stanem. Stosowanie usług aktywowanych przez wywołania ma

jeszcze tę zaletę, że eliminuje konieczność utrzymywania instancji pomiędzy wywołaniami (nie ma takiej potrzeby, skoro stan jest odczytywany z pamięci trwałej). Jedynym aspektem, który odróżnia usługi trwałe od klasycznych usług aktywowanych przez wywołania, jest konieczność stosowania trwałego repozytorium stanów.

O ile w teorii nic nie stoi na przeszkodzie, aby usługi trwałe zaimplementować na bazie usług sesyjnych czy wręcz usług singletonowych, które przechowywałyby swój stan w jakiejś pamięci, praktyka pokazuje, że takie rozwiązanie jest zupełnie nieefektywne. W przypadku usługi sesyjnej należałoby utrzymywać przez długi czas otwarty obiekt pośrednika po stronie klienta, zatem nie byłaby możliwa ciągła współpraca z klientami kończącymi i ponownie nawiązującymi połączenia. W przypadku usługi singletonowej, której czas życia w założeniu jest nieskończony i która obsługuje aplikacje klienckie wielokrotnie nawiązujące kolejne połączenia, stosowanie pamięci trwałej nie ma większego sensu. W tej sytuacji tryb aktywowania przez wywołania wydaje się zdecydowanie najlepszym rozwiązaniem. Warto pamiętać, że w przypadku usług trwałych aktywowanych przez wywołania, gdzie zasadniczym celem jest obsługa długotrwałych przepływów pracy (nie zapewnianie skalowalności czy efektywne zarządzanie zasobami), obsługa interfejsu `IDisposable` jest opcjonalna. W przypadku usług trwałych także obecność sesji transportowej jest opcjonalna, ponieważ nie ma potrzeby utrzymywania sesji logicznych pomiędzy klientem a usługą. Sesja transportowa będzie pełniła funkcję elementu używanego kanału transportowego i jako taka nie będzie decydowała o czasie życia instancji usługi.

Tworzenie i niszczenie instancji usługi

W momencie rozpoczęcia długoterminowego procesu przepływu pracy odpowiednia usługa musi najpierw zapisać swój stan w pamięci trwałej, tak aby kolejne operacje miały dostęp do stanu w tej pamięci. Po zakończeniu pracy usługa musi usunąć swój stan z pamięci trwałej; w przeciwnym razie pamięć byłaby stopniowo zaśmiecana starymi, niepotrzebnymi stanami instancji.

Identyfikatory instancji i pamięć trwała

Ponieważ nowa instancja usługi jest tworzona dla każdej operacji, instancja musi mieć możliwość odnalezienia i załadowania stanu przechowywanego w pamięci trwałej. Oznacza to, że klient musi dostarczyć instancji usługi jakiś identyfikator stanu. Taki identyfikator określa się mianem **identyfikatora instancji**. Obsługa klientów sporadycznie nawiązujących połączenie z usługą oraz aplikacji klienckich czy nawet komputerów, które mogą ulegać zasadniczym zmianom pomiędzy wywołaniami (wszystko w czasie trwania jednego przepływu pracy), wymaga zapisywania identyfikatora instancji w jakiejś pamięci trwałej po stronie klienta (na przykład w pliku) i przekazywania go w każdym wywołaniu. Klient może usunąć identyfikator instancji dopiero po zakończeniu odpowiedniego przepływu pracy. Typ danych używany do reprezentowania identyfikatora instancji powinien zapewniać możliwość szeregowania (serializacji) i porównywania. Warunek szeregowalności jest o tyle ważny, że usługa będzie musiała zapisać identyfikator w pamięci trwałej (wraz ze swoim stanem). Możliwość porównywania identyfikatora jest niezbędna, jeśli usługa ma odczytywać odpowiedni stan z pamięci trwałej. Wszystkie typy proste frameworku .NET (jak `int`, `string` czy `Guid`) spełniają wymagania stawiane identyfikatorom instancji.

Pamięć trwała zwykle ma postać słownika, który obejmuje pary złożone z identyfikatora i stanu instancji. Usługa z reguły używa pojedynczego identyfikatora w roli reprezentacji całego swo-

jego stanu, jednak istnieje możliwość stosowania bardziej złożonych relacji, w tym wielu kluczy czy wręcz hierarchii kluczy. Dla uproszczenia w tym materiale będą omawiane tylko pojedyncze identyfikatory. Wiele usług dodatkowo używa klas lub struktur pomocniczych łączących wszystkie zmienne składowe i zapisujących dane zagregowane w tym typie w pamięci trwałej (oraz odczytujących te dane z pamięci trwałej). I wreszcie sam dostęp do pamięci trwałej musi być synchronizowany i gwarantować bezpieczeństwo przetwarzania wielowątkowego. Spełnienie tych warunków jest konieczne, ponieważ zawartość pamięci trwałej może być jednocześnie odczytywana i modyfikowana przez wiele instancji.

Aby ułatwić Ci implementację i obsługę prostych usług trwałych, napisałem następującą klasę `FileInstanceStore<ID,T>`:

```
public interface IInstanceStore<ID,T> where ID : IEquatable<ID>
{
   void RemoveInstance(ID instanceId);
   bool ContainsInstance(ID instanceId);
   T this[ID instanceId]
   {get;set;}
}

public class FileInstanceStore<ID,T> : IInstanceStore<ID,T> where ID :
                                                                IEquatable<ID>
{
   protected readonly string Filename;
   public FileInstanceStore(string fileName);
   // Dalsza część implementacji…
}
```

Klasa `FileInstanceStore<ID,T>` reprezentuje uniwersalną, plikową pamięć instancji. Klasa `File` ↪`InstanceStore<ID,T>` otrzymuje dwa parametry typów: parametr typu `ID` musi reprezentować typ porównywalny, zaś parametr typu `T` reprezentuje stan instancji. W czasie wykonywania statyczny konstruktor klasy `FileInstanceStore<ID,T>` sprawdza, czy oba te typy są szeregowalne (mogą podlegać serializacji).

Klasa `FileInstanceStore<ID,T>` udostępnia prosty mechanizm indeksujący, który umożliwia zapisywanie stanu instancji w pliku i odczytywanie go z tego pliku. Stan instancji można też usunąć z pliku. Istnieje także możliwość sprawdzenia, czy dany plik zawiera stan instancji. Wszystkie te operacje zdefiniowano w interfejsie `IInstanceStore<ID,T>`. Implementacja tego interfejsu w formie klasy `FileInstanceStore<ID,T>` reprezentuje słownik, a każda próba dostępu powoduje serializację i deserializację tego słownika w i z pliku. Klasa `FileInstanceStore<ID,T>` użyta po raz pierwszy tworzy i inicjalizuje plik, umieszczając w nim pusty słownik (po sprawdzeniu, czy rzeczywiście ten plik nie zawiera żadnych danych).

Bezpośrednie przekazywanie identyfikatorów instancji

Najprostszym sposobem udostępniania identyfikatora instancji przez klienta jest bezpośrednie (jawne) przekazywanie tego identyfikatora w formie parametru każdej operacji, która wymaga dostępu do stanu instancji. Odpowiedni przykład klienta i usługi wraz z definicjami typów pomocniczych pokazano na listingu 4.9.

Listing 4.9. Bezpośrednie przekazywanie identyfikatorów instancji

```
[DataContract]
class SomeKey : IEquatable<SomeKey>
{...}
```

```
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   void MyMethod(SomeKey instanceId);
}

// Typ pomocniczy używany przez usługę do uzyskiwania swojego stanu
[Serializable]
struct MyState
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
   public void MyMethod(SomeKey instanceId)
   {
      GetState(instanceId);
      DoWork();
      SaveState(instanceId);
   }
   void DoWork()
   {...}

   // Uzyskuje i ustawia MyState na podstawie pamięci trwałej
   void GetState(SomeKey instanceId)
   {...}

   void SaveState(SomeKey instanceId)
   {...}
}
```

Aby lepiej zrozumieć kod z listingu 4.9, warto przyjrzeć się listingowi 4.10 obsługującemu kieszonkowy kalkulator z pamięcią trwałą w formie pliku dyskowego.

Listing 4.10. Kalkulator z jawnie przekazywanym identyfikatorem instancji

```
[ServiceContract]
interface ICalculator
{
   [OperationContract]
   double Add(double number1,double number2);

   /* Pozostałe działania arytmetyczne */

   // Operacje związane z zarządzaniem pamięcią

   [OperationContract]
   void MemoryStore(string instanceId,double number);

   [OperationContract]
   void MemoryClear(string instanceId);
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculator : ICalculator
{
   static IInstanceStore<string,double> Memory =
            new FileInstanceStore<string,double>(Settings.Default.MemoryFileName);

   public double Add(double number1,double number2)
   {
```

```
        return number1 + number2;
    }
    public void MemoryStore(string instanceId,double number)
    {
        lock(typeof(MyCalculator))
        {
            Memory[instanceId] = number;
        }
    }
    public void MemoryClear(string instanceId)
    {
        lock(typeof(MyCalculator))
        {
            Memory.RemoveInstance(instanceId);
        }
    }
    // Dalsza część implementacji…
}
```

W kodzie z listingu 4.10 nazwa pliku jest dostępna we wszystkich właściwościach projektu w klasie `Settings`. Wszystkie instancje usługi kalkulatora używają tej samej pamięci statycznej reprezentowanej przez klasę `FileInstanceStore<string,double>`. Kalkulator synchronizuje dostęp do tej pamięci w każdej operacji i we wszystkich instancjach, blokując typ usługi. Usunięcie zawartości pamięci jest traktowane przez kalkulator jako sygnał końca przepływu pracy, po którym usługa czyści swój stan w pamięci trwałej.

Identyfikatory instancji w nagłówkach

Zamiast bezpośrednio przekazywać identyfikator instancji, klient może umieścić ten identyfikator w nagłówkach komunikatów. Stosowanie nagłówków komunikatów jako techniki przekazywania dodatkowych parametrów na potrzeby niestandardowych kontekstów zostanie szczegółowo omówione w dodatku B. W tym przypadku klient używa klasy pośrednika `HeaderClientBase<T,H>`, a usługa może odczytać identyfikator instancji za pomocą odpowiednich operacji opracowanej przeze mnie klasy pomocniczej `GenericContext<H>`. Usługa może albo używać klasy `GenericContext<H>` w jej oryginalnej formie, albo opakować ją w ramach dedykowanego kontekstu.

Ogólny wzorzec stosowania tej techniki pokazano na listingu 4.11.

Listing 4.11. Przekazywanie identyfikatorów instancji w nagłówkach komunikatów

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
// Strona klienta
class MyContractClient : HeaderClientBase<IMyContract,SomeKey>,IMyContract
{
    public MyContractClient(SomeKey instanceId)
    {}
    public MyContractClient(SomeKey instanceId,string endpointName) :
                                                    base(instanceId,endpointName)
    {}
```

```
    // Pozostałe konstruktory…

    public void MyMethod()
    {
       Channel.MyMethod();
    }
}
// Strona usługi
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    public void MyMethod()
    {
       SomeKey instanceId = GenericContext<SomeKey>.Current.Value;
       ...
    }
    // Dalsza część taka sama jak na listingu 4.9
}
```

Także tym razem, aby schemat z listingu 4.11 nie był zbyt abstrakcyjny, warto przeanalizować listing 4.12 z kodem kalkulatora stosującego technikę przekazywania identyfikatorów w formie nagłówków komunikatów.

Listing 4.12. Kalkulator z identyfikatorem instancji w nagłówkach komunikatów

```
[ServiceContract]
interface ICalculator
{
    [OperationContract]
    double Add(double number1,double number2);

    /* Pozostałe działania arytmetyczne */

    // Operacje związane z zarządzaniem pamięcią

    [OperationContract]
    void MemoryStore(double number);

    [OperationContract]
    void MemoryClear();
}
// Strona klienta
class MyCalculatorClient : HeaderClientBase<ICalculator,string>,ICalculator
{
    public MyCalculatorClient(string instanceId)
    {}

    public MyCalculatorClient(string instanceId,string endpointName) :
                                                    base(instanceId,endpointName)
    {}

    // Pozostałe konstruktory…

    public double Add(double number1,double number2)
    {
       return Channel.Add(number1,number2);
    }

    public void MemoryStore(double number)
    {
       Channel.MemoryStore(number);
    }
```

```
   // Dalsza część implementacji…
}
// Strona usługi
// Jeśli stosowanie klasy GenericContext<T> jest niewygodne, można ją opakować:
static class CalculatorContext
{
   public static string Id
   {
      get
      {
         return GenericContext<string>.Current.Value ?? String.Empty;
      }
   }
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculator : ICalculator
{
   static IInstanceStore<string,double> Memory =
            new FileInstanceStore<string,double>(Settings.Default.MemoryFileName);

   public double Add(double number1,double number2)
   {
      return number1 + number2;
   }
   public void MemoryStore(double number)
   {
      lock(typeof(MyCalculator))
      {
         Memory[CalculatorContext.Id] = number;
      }
   }
   public void MemoryClear()
   {
      lock(typeof(MyCalculator))
      {
         Memory.RemoveInstance(CalculatorContext.Id);
      }
   }
   // Dalsza część implementacji…
}
```

Powiązania kontekstu dla identyfikatorów instancji

Technologia WCF udostępnia powiązania dedykowane, które umożliwiają przekazywanie niestandardowych parametrów kontekstu. Powiązania tego typu, określane mianem **powiązań kontekstu** (ang. *context bindings*), zostaną szczegółowo omówione w dodatku B. Aplikacje klienckie mogą używać klasy `ContextClientBase<T>` do przekazywania identyfikatorów instancji za pośrednictwem protokołu powiązań kontekstu. Ponieważ powiązania kontekstu wymagają klucza i wartości dla każdego parametru kontekstu, aplikacje klienckie będą musiały przekazywać oba te elementy do swoich pośredników. W przypadku zastosowania tego samego interfejsu `IMyContract` co na listingu 4.11 odpowiedni pośrednik mógłby mieć następującą postać:

```
class MyContractClient : ContextClientBase<IMyContract>,IMyContract
{
   public MyContractClient(string key,string instanceId) : base(key,instanceId)
   {}
   public MyContractClient(string key,string instanceId,string endpointName) :
                                             base(key,instanceId,endpointName)
```

```
    {}

    // Pozostałe konstruktory…

    public void MyMethod()
    {
       Channel.MyMethod();
    }
}
```

Warto pamiętać, że protokół kontekstu obsługuje tylko łańcuchy w roli kluczy i wartości. Skoro wartość klucza musi być znana usłudze z wyprzedzeniem, klient może równie dobrze trwale zakodować ten sam klucz w klasie samego pośrednika. Usługa może następnie uzyskać identyfikator instancji przy użyciu mojej klasy pomocniczej `ContextManager` (opisanej w dodatku B). Podobnie jak w przypadku nagłówków komunikatów usługa może też opakować interakcję z klasą `ContextManager` w ramach dedykowanej klasy kontekstu.

Na listingu 4.13 pokazano ogólny wzorzec przekazywania identyfikatora instancji za pośrednictwem powiązań kontekstu. Warto zwrócić szczególną uwagę na klucz identyfikatora instancji trwale zapisany w kodzie pośrednika i na to, że jest to identyfikator znany usłudze.

Listing 4.13. Przekazywanie identyfikatora instancji za pośrednictwem powiązania kontekstu

```
// Strona klienta
class MyContractClient : ContextClientBase<IMyContract>,IMyContract
{
   public MyContractClient(string instanceId) : base("MyKey",instanceId)
   {}

   public MyContractClient(string instanceId,string endpointName) :
                                          base("MyKey",instanceId,endpointName)
   {}

   // Pozostałe konstruktory…

   public void MyMethod()
   {
      Channel.MyMethod();
   }
}
// Strona usługi
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
   public void MyMethod()
   {
      string instanceId = ContextManager.GetContext("MyKey");

      GetState(instanceId);
      DoWork();
      SaveState(instanceId);
   }
   void DoWork()
   {...}

   // Uzyskuje i ustawia stan na podstawie pamięci trwałej
   void GetState(string instanceId)
   {...}
```

```
   void SaveState(string instanceId)
   {...}
}
```

Na listingu 4.14 pokazano przykład konkretnego użycia tego schematu na potrzeby usługi kalkulatora.

Listing 4.14. Kalkulator z identyfikatorem instancji przekazywanym za pośrednictwem powiązania kontekstu

```
// Strona klienta
class MyCalculatorClient : ContextClientBase<ICalculator>,ICalculator
{
   public MyCalculatorClient(string instanceId) : base("CalculatorId",instanceId)
   {}
   public MyCalculatorClient(string instanceId,string endpointName) :
                                       base("CalculatorId",instanceId,endpointName)
   {}

   // Pozostałe konstruktory…

   public double Add(double number1,double number2)
   {
      return Channel.Add(number1,number2);
   }
   public void MemoryStore(double number)
   {
      Channel.MemoryStore(number);
   }

   // Dalsza część implementacji…
}
// Strona usługi
static class CalculatorContext
{
   public static string Id
   {
      get
      {
         return ContextManager.GetContext("CalculatorId") ?? String.Empty;
      }
   }
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculator : ICalculator
{
   // To samo co na listingu 4.12
}
```

Stosowanie standardowego identyfikatora dla powiązania kontekstu

Konieczność trwałego kodowania i znajomości z wyprzedzeniem klucza używanego dla identyfikatora instancji jest pewnym utrudnieniem. Powiązania kontekstu zaprojektowano z myślą o usługach trwałych, zatem każde takie powiązanie zawiera automatycznie wygenerowany identyfikator instancji w postaci obiektu klasy `Guid` (w formacie łańcuchowym), który jest dostępny za pośrednictwem zastrzeżonego klucza `instanceId`. Klient i usługa mają dostęp do tej samej wartości identyfikatora klucza. Wartość tego klucza jest inicjalizowana zaraz po zwróceniu sterowania przez pierwsze wywołanie pośrednika — po tym, jak powiązanie zyska możliwość skojarzenia klienta i usługi. Jak każdy parametr przekazywany za pośrednictwem powiązania kontekstu, tak i wartość identyfikatora instancji jest niezmienna przez cały czas życia pośrednika.

Aby uprościć interakcję ze standardowym identyfikatorem instancji, uzupełniłem klasę `Context ➥Manager` o metody, właściwości i metody pośrednika zarządzające tym identyfikatorem (patrz listing 4.15).

Listing 4.15. Zarządzanie standardowym identyfikatorem instancji w klasie ContextManager

```
public static class ContextManager
{
   public const string InstanceIdKey = "instanceId";

   public static Guid InstanceId
   {
      get
      {
         string id = GetContext(InstanceIdKey) ?? Guid.Empty.ToString();
         return new Guid(id);
      }
   }
   public static Guid GetInstanceId(IClientChannel innerChannel)
   {
      try
      {
         string instanceId =
            innerChannel.GetProperty<IContextManager>().GetContext()[InstanceIdKey];
         return new Guid(instanceId);
      }
      catch(KeyNotFoundException)
      {
         return Guid.Empty;
      }
   }
   public static void SetInstanceId(IClientChannel innerChannel,Guid instanceId)
   {
      SetContext(innerChannel,InstanceIdKey,instanceId.ToString());
   }
   public static void SaveInstanceId(Guid instanceId,string fileName)
   {
      using(Stream stream =
                    new FileStream(fileName,FileMode.OpenOrCreate,FileAccess.Write))
      {
         IFormatter formatter = new BinaryFormatter();
         formatter.Serialize(stream,instanceId);
      }
   }

   public static Guid LoadInstanceId(string fileName)
   {
      try
      {
         using(Stream stream = new FileStream(fileName,FileMode.Open,
                                              FileAccess.Read))
         {
            IFormatter formatter = new BinaryFormatter();
            return (Guid)formatter.Deserialize(stream);
         }
      }
      catch
      {
         return Guid.Empty;
      }
```

```
    }
    // Pozostałe składowe…
}
```

Klasa `ContextManager` definiuje metody `GetInstanceId()` i `SetInstanceId()`, które umożliwiają klientowi odpowiednio odczytanie identyfikatora instancji z kontekstu i zapisanie tego identyfikatora w kontekście. Do uzyskiwania identyfikatora usługa używa właściwości `InstanceId` (dostępnej tylko do odczytu). Klasa `ContextManager` w tej formie dodatkowo zapewnia bezpieczeństwo typów, ponieważ traktuje identyfikator instancji jako wartość typu `Guid` (nie łańcuch). Klasa wprowadza też mechanizmy obsługi błędów.

I wreszcie klasa `ContextManager` udostępnia metody `LoadInstanceId()` i `SaveInstanceId()`, które odpowiednio odczytują identyfikator instancji z pliku i zapisują ten identyfikator w pliku. Wymienione metody są szczególnie wygodne po stronie klienta, który może zapisywać identyfikator pomiędzy kolejnymi sesjami współpracy aplikacji klienckiej z usługą.

Klient może co prawda użyć klasy `ContextClientBase<T>` do przekazania standardowego identyfikatora instancji (jak na listingu 4.13), jednak lepszym rozwiązaniem jest wykorzystanie wbudowanej obsługi tego identyfikatora (patrz listing 4.16).

Listing 4.16. Klasa ContextClientBase<T> uzupełniona o obsługę standardowych identyfikatorów instancji

```
public abstract class ContextClientBase<T> : ClientBase<T> where T : class
{
   public Guid InstanceId
   {
      get
      {
         return ContextManager.GetInstanceId(InnerChannel);
      }
   }
   public ContextClientBase(Guid instanceId) :
                           this(ContextManager.InstanceIdKey,instanceId.ToString())
   {}

   public ContextClientBase(Guid instanceId,string endpointName) :
            this(ContextManager.InstanceIdKey,instanceId.ToString(),endpointName)
   {}

   // Pozostałe konstruktory…
}
```

Na listingu 4.17 pokazano przykład użycia standardowego identyfikatora instancji przez klienta i usługę kalkulatora.

Listing 4.17. Usługa kalkulatora korzystająca ze standardowego identyfikatora

```
// Strona klienta
class MyCalculatorClient : ContextClientBase<ICalculator>,ICalculator
{
   public MyCalculatorClient()
   {}
   public MyCalculatorClient(Guid instanceId) : base(instanceId)
   {}
   public MyCalculatorClient(Guid instanceId,string endpointName) :
                                                   base(instanceId,endpointName)
   {}
```

```
   // Dalsza część taka sama jak na listingu 4.14
}
// Strona usługi
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculator : ICalculator
{
   static IInstanceStore<Guid,double> Memory =
               new FileInstanceStore<Guid,double>(Settings.Default.MemoryFileName);

   public double Add(double number1,double number2)
   {
      return number1 + number2;
   }
   public void MemoryStore(double number)
   {
      lock(typeof(MyCalculator))
      {
      Memory[ContextManager.InstanceId] = number;
      }
   }
   public void MemoryClear()
   {
      lock(typeof(MyCalculator))
      {
         Memory.RemoveInstance(ContextManager.InstanceId);
      }
   }
   // Dalsza część implementacji…
}
```

Automatyczne zachowanie trwałe

Wszystkie opisane do tej pory techniki obsługi usług trwałych wymagały wykonywania dość złożonych czynności po stronie samych usług — w szczególności operowania na trwałej pamięci stanów i bezpośredniego zarządzania stanem instancji przy okazji każdej operacji. Powtarzalność tych działań oznacza, że środowisko WCF może je zautomatyzować, serializując i deserializując stan usługi dla każdej operacji na podstawie wskazanej pamięci stanów (przy użyciu standardowego identyfikatora instancji).

Jeśli programista zdecyduje się przekazać środowisku WCF odpowiedzialność za zarządzanie stanem instancji, zarządzanie będzie przebiegało według następujących reguł:

- Jeśli klient nie przekazał identyfikatora, środowisko WCF utworzy nową instancję usługi, korzystając z jej konstruktora. Po zakończeniu wywołania środowisko WCF serializuje instancję w pamięci stanów.
- Jeśli klient przekazuje identyfikator do pośrednika i jeśli pamięć stanów zawiera już stan pasujący do tego identyfikatora, środowisko WCF nie wywołuje konstruktora instancji. W takim przypadku wywołanie jest obsługiwane przez instancję odczytaną (w procesie deserializacji) z pamięci stanów.
- Jeśli klient przekazuje prawidłowy identyfikator, środowisko WCF każdorazowo deserializuje instancję z pamięci stanów, wywołuje odpowiednią operację i ponownie serializuje w pamięci nowy stan (zmodyfikowany przez tę operację).
- Jeśli klient przekazuje identyfikator, który nie występuje w pamięci stanów, środowisko WCF zgłasza stosowny wyjątek.

Atrybut zachowania usługi trwałej

Do włączania automatycznego zachowania trwałego służy atrybut zachowania `DurableService`, który zdefiniowano w następujący sposób:

```
public sealed class DurableServiceAttribute : Attribute,IServiceBehavior,...
{...}
```

Atrybut `DurableService` należy zastosować bezpośrednio dla klasy usługi. Co ważne, klasa usługi musi być dodatkowo oznaczona albo jako szeregowalna (przystosowana do serializacji), albo jako kontrakt danych (z atrybutem `DataMember` użytym dla wszystkich składowych wymagających zarządzania trwałym stanem):

```
[Serializable]
[DurableService]
class MyService : IMyContract
{
   /* Tylko szeregowalne zmienne składowe */

   public void MyMethod()
   {
      // Właściwe działania
   }
}
```

Instancja usługi trwałej może teraz zarządzać swoim stanem w zmiennych składowych, tak jakby była zwykłą instancją — tym razem to środowisko WCF odpowiada za trwałość tych składowych. Gdyby usługa nie została oznaczona jako szeregowalna (lub jako kontrakt danych), pierwsze wywołanie zakończyłoby się niepowodzeniem z powodu podjętej przez środowisko WCF próby serializacji instancji w pamięci stanów. Każda usługa korzystająca z mechanizmu automatycznego zarządzania trwałym stanem musi zostać skonfigurowana jako usługa sesyjna, a mimo to zachowuje się jak usługa aktywowana przez wywołania (środowisko WCF dezaktywuje kontekst po każdym wywołaniu). Co więcej, usługa musi stosować jeden ze związków kontekstu dla każdego punktu końcowego, aby było możliwe stosowanie standardowego identyfikatora instancji. Sam kontrakt musi dopuszczać lub wymuszać stosowanie sesji transportowej (nie może jej wyłączać). Oba te ograniczenia są sprawdzane na etapie ładowania usługi.

Atrybut zachowania operacji trwałej

Usługa może opcjonalnie użyć atrybutu zachowania `DurableOperation` do zasygnalizowania środowisku WCF konieczności wyczyszczenia stanu w pamięci trwałej na końcu przepływu pracy:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class DurableOperationAttribute : Attribute,...
{
   public bool CanCreateInstance
   {get;set;}
   public bool CompletesInstance
   {get;set;}
}
```

Przypisanie właściwości `CompletesInstance` wartości `true` powoduje, że środowisko WCF usunie identyfikator instancji z pamięci trwałej zaraz po zwróceniu sterowania przez wywołanie operacji. Wartością domyślną właściwości `CompletesInstance` jest `false`. Jeśli klient nie przekazał identyfikatora instancji, można zapobiec utworzeniu nowej instancji przez operację — wystarczy przypisać wartość `false` właściwości `CanCreateInstance`. Kod z listingu 4.18 demonstruje użycie właściwości `CompletesInstance` dla operacji `MemoryClear()` usługi kalkulatora.

Listing 4.18. Przykład użycia właściwości CompletesInstance do usunięcia stanu

```
[Serializable]
[DurableService]
class MyCalculator : ICalculator
{
   double Memory
   {get;set;}
   public double Add(double number1,double number2)
   {
      return number1 + number2;
   }
   public void MemoryStore(double number)
   {
      Memory = number;
   }
   [DurableOperation(CompletesInstance = true)]
   public void MemoryClear()
   {
      Memory = 0;
   }
   // Dalsza część implementacji…
}
```

Stosowanie właściwości `CompletesInstance` jest o tyle problematyczne, że identyfikator kontekstu jest niezmienny. Oznacza to, że jeśli klient próbuje wykonać kolejne wywołania poprzez obiekt pośrednika (już po wykonaniu operacji z wartością `true` we właściwości `CompletesInstance`), wszystkie te wywołania zakończą się niepowodzeniem, ponieważ pamięć trwała nie zawiera już identyfikatora instancji. Oznacza to, że klient musi wiedzieć o braku możliwości dalszego korzystania z tego samego pośrednika — jeśli ma kierować dalsze wywołania do danej usługi, musi użyć nowego pośrednika, który nie ma jeszcze identyfikatora instancji (w ten sposób klient rozpocznie nowy przepływ pracy). Jednym ze sposobów wymuszania odpowiedniego zastosowania jest zamykanie programu klienta po zakończeniu przepływu pracy (lub tworzenie nowej referencji do obiektu pośrednika). Na listingu 4.19 pokazano kod demonstrujący, jak zarządzać tym samym pośrednikiem usługi kalkulatora po wyczyszczeniu pamięci (w kodzie wykorzystano definicję pośrednika z listingu 4.17).

Listing 4.19. Resetowanie pośrednika po zakończeniu przepływu pracy

```
class CalculatorProgram
{
   MyCalculatorClient m_Proxy;

   public CalculatorProgram()
   {
      Guid calculatorId =
               ContextManager.LoadInstanceId(Settings.Default.CalculatorIdFileName);

      m_Proxy = new MyCalculatorClient(calculatorId);
   }
   public void Add()
   {
      m_Proxy.Add(2,3);
   }
   public void MemoryClear()
   {
      m_Proxy.MemoryClear();

      ResetDurableSession(ref m_Proxy);
```

```
    }
    public void Close()
    {
       ContextManager.SaveInstanceId(m_Proxy.InstanceId,
                                     Settings.Default.CalculatorIdFileName);
       m_Proxy.Close();
    }
    void ResetDurableSession(ref MyCalculatorClient proxy)
    {
       ContextManager.SaveInstanceId(Guid.Empty,
                                     Settings.Default.CalculatorIdFileName);
       Binding binding = proxy.Endpoint.Binding;
       EndpointAddress address = proxy.Endpoint.Address;

       proxy.Close();

       proxy = new MyCalculatorClient(binding,address);
    }
}
```

W kodzie z listingu 4.19 wykorzystano klasę pomocniczą `ContextManager` do załadowania identyfikatora instancji z pliku i zapisania jej w tym pliku. Konstruktor programu klienta tworzy nowy obiekt pośrednika na podstawie identyfikatora odczytanego z tego pliku. Jak widać na wcześniejszym listingu 4.15, jeśli plik nie zawiera identyfikatora instancji, metoda `LoadInstanceId()` zwraca wartość `Guid.Empty`. Klasę `ContextClientBase<T>` zaprojektowałem w taki sposób, aby obsługiwała pusty identyfikator GUID w roli identyfikatora kontekstu — w razie przekazania pustego identyfikatora GUID klasa `ContextClient Base<T>` konstruuje swój obiekt bez identyfikatora instancji, rozpoczynając tym samym nowy przepływ pracy. Po wyczyszczeniu pamięci usługi kalkulatora klient wywołuje metodę pomocniczą `ResetDurableSession()`. Metoda `ResetDurableSession()` zapisuje najpierw pusty identyfikator GUID w pliku, po czym tworzy kopię istniejącego pośrednika. Metoda kopiuje adres i powiązanie dotychczasowego pośrednika, zamyka go i tak ustawia referencję do pośrednika, aby wskazywała nowo skonstruowany obiekt z tym samym adresem i powiązaniem oraz pustym identyfikatorem GUID w roli identyfikatora instancji.

Programowe zarządzanie instancją

Technologia WCF oferuje prostą klasę pomocniczą dla usług trwałych, nazwaną `DurableOpera ↪tionContext`:

```
public static class DurableOperationContext
{
   public static void AbortInstance();
   public static void CompleteInstance();
   public static Guid InstanceId
   {get;}
}
```

Metoda `CompleteInstance()` umożliwia usłudze programowe (nie deklaratywnie, jak w przypadku atrybutu `DurableOperation`) zakończenie działania instancji i usunięcie jej stanu z pamięci zaraz po zwróceniu sterowania przez wywołanie. Metoda `AbortInstance()` anuluje wszystkie zmiany wprowadzone w pamięci trwałej w czasie danego wywołania, przywracając stan sprzed tego wywołania. Właściwość `InstanceId` przypomina wspomnianą wcześniej właściwość `Context ↪Manager.InstanceId`.

Dostawcy trwałości

Atrybut `DurableService` instruuje środowisko WCF, kiedy serializować i deserializować daną instancję usługi, ale w żaden sposób nie wskazuje miejsca tej serializacji i deserializacji (nie zawiera żadnych informacji na temat pamięci stanów). Środowisko WCF stosuje wzorzec projektowy mostu (ang. *bridge*) w postaci modelu dostawcy — dzięki temu programista może przekazywać informacje o pamięci stanów niezależnie od wspomnianego atrybutu. Oznacza to, że atrybut `DurableService` jest oddzielony od pamięci stanów, dzięki czemu istnieje możliwość stosowania mechanizmu automatycznego zachowania trwałego w przypadku pamięci zapewniających zgodność z tym mechanizmem.

Jeśli usługę skonfigurowano z atrybutem `DurableService`, należy jeszcze skonfigurować hosta z odpowiednią fabryką dostawców trwałości. Klasa tej fabryki dziedziczy po klasie abstrakcyjnej `PersistenceProviderFactory` i tworzy podklasę klasy abstrakcyjnej `PersistenceProvider`:

```
public abstract class PersistenceProviderFactory : CommunicationObject
{
   protected PersistenceProviderFactory();
   public abstract PersistenceProvider CreateProvider(Guid id);
}

public abstract class PersistenceProvider : CommunicationObject
{
   protected PersistenceProvider(Guid id);

   public Guid Id
   {get;}

   public abstract object Create(object instance,TimeSpan timeout);
   public abstract void Delete(object instance,TimeSpan timeout);
   public abstract object Load(TimeSpan timeout);
   public abstract object Update(object instance,TimeSpan timeout);

   // Pozostałe składowe…
}
```

Najbardziej popularnym sposobem wskazywania fabryki dostawców trwałości jest umieszczanie odpowiednich zapisów w formie zachowania usługi w pliku konfiguracyjnym hosta i odwoływanie się do tego zachowania w definicji usługi:

```
<behaviors>
   <serviceBehaviors>
      <behavior name = "DurableService">
         <persistenceProvider
            type = "...type...,...assembly ..."
            <!-- Parametry dostawcy -->
         />
      </behavior>
   </serviceBehaviors>
</behaviors>
```

Po skonfigurowaniu hosta z uwzględnieniem fabryki dostawców trwałości środowisko używa klasy `PersistenceProvider` dla każdego wywołania do serializacji i deserializacji instancji. Gdyby nie określono żadnej fabryki dostawców trwałości, środowisko WCF przerwałoby tworzenie hosta usługi.

Niestandardowi dostawcy trwałości

Dobrą ilustracją sposobu pisania prostego, niestandardowego dostawcy trwałości jest moja klasa `FilePersistenceProviderFactory`, której definicja jest następująca:

```
public class FilePersistenceProviderFactory : PersistenceProviderFactory
{
   public FilePersistenceProviderFactory();
   public FilePersistenceProviderFactory(string fileName);
   public FilePersistenceProviderFactory(NameValueCollection parameters);
}
public class FilePersistenceProvider : PersistenceProvider
{
   public FilePersistenceProvider(Guid id,string fileName);
}
```

Klasa `FilePersistenceProvider` jest opakowaniem mojej klasy `FileInstanceStore<ID,T>`. Konstruktor klasy `FilePersistenceProviderFactory` wymaga wskazania nazwy odpowiedniego pliku. Jeśli na wejściu konstruktora nie zostanie przekazana żadna nazwa, klasa `FilePersistenceProvider ↪Factory` użyje domyślnego pliku *Instances.bin*.

Warunkiem stosowania fabryki niestandardowych dostawców trwałości w pliku konfiguracyjnym jest zdefiniowanie konstruktora, który otrzyma na wejściu kolekcję typu `NameValue ↪Collection` (reprezentującą parametry). Parametry w tej kolekcji mają postać zwykłych par kluczy i wartości łańcuchowych wskazanych w sekcji zachowania fabryki dostawców w pliku konfiguracyjnym. W tej roli można stosować niemal dowolne klucze i wartości. Poniższy przykład pokazuje, jak można w ten sposób określić nazwę pliku:

```
<behaviors>
   <serviceBehaviors>
      <behavior name = "Durable">
         <persistenceProvider
            type = "FilePersistenceProviderFactory,ServiceModelEx"
            fileName = "MyService.bin"
         />
      </behavior>
   </serviceBehaviors>
</behaviors>
```

Konstruktor może teraz użyć kolekcji `parameters` do uzyskania dostępu do tych parametrów:

```
string fileName = parameters["fileName"];
```

Dostawca trwałości na bazie systemu SQL Server

Środowisko WCF jest dostarczane wraz z dostawcą trwałości, który zapisuje stan instancji w dedykowanej tabeli bazy danych systemu SQL Server. Po przeprowadzeniu instalacji z ustawieniami domyślnymi odpowiednie skrypty instalacyjne tej bazy danych można znaleźć w katalogu *C:\Windows\Microsoft.NET\Framework\v4.0.30316\SQL\EN*. Warto pamiętać, że dołączony do środowiska WCF dostawca trwałości może współpracować tylko z bazami danych SQL Server 2005, SQL Server 2008 lub nowszymi. Wspomniany dostawca trwałości ma postać klas `SqlPersistenceProviderFactory` i `SqlPersistenceProvider` należących do podzespołu `System.Workflow ↪Services` w przestrzeni nazw `System.ServiceModel.Persistence`.

Użycie tego dostawcy wymaga tylko wskazania odpowiedniej fabryki dostawców i zdefiniowania łańcucha połączenia:

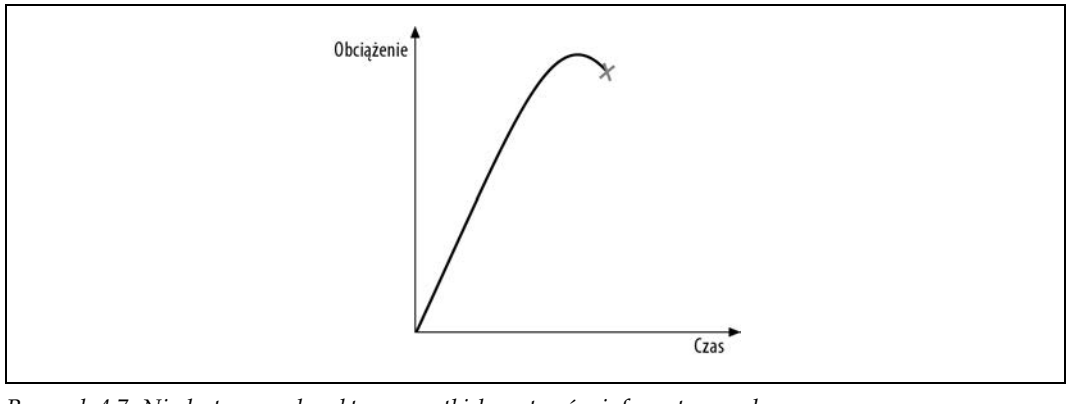
```
<connectionStrings>
   <add name = "DurableServices"
      connectionString = "..."
      providerName = "System.Data.SqlClient"
   />
</connectionStrings>

<behaviors>
   <serviceBehaviors>
      <behavior name = "Durable">
         <persistenceProvider
            type = "System.ServiceModel.Persistence.SqlPersistenceProviderFactory,
                    System.WorkflowServices,Version=4.0.0.0,Culture=neutral,
                    PublicKeyToken=31bf3856ad364e35"
            connectionStringName = "DurableServices"
         />
      </behavior>
   </serviceBehaviors>
</behaviors>
```

Istnieje też możliwość wymuszenia na środowisku WCF serializacji instancji w formie tekstowej (zamiast domyślnej serializacji binarnej) na przykład na potrzeby diagnostyczne czy analityczne:

```
<persistenceProvider
   type = "System.ServiceModel.Persistence.SqlPersistenceProviderFactory,
           System.WorkflowServices,Version=4.0.0.0,Culture=neutral,
           PublicKeyToken=31bf3856ad364e35"
   connectionStringName = "DurableServices"
   serializeAsText = "true"
/>
```

Dławienie

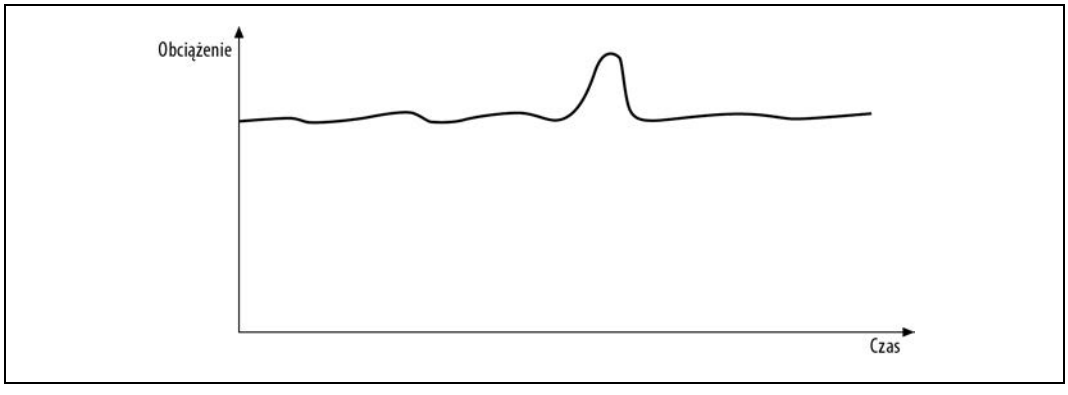
Dławienie (ang. *throttling*) nie jest co prawda techniką bezpośredniego zarządzania instancjami, ale pozwala opanować połączenia nawiązywane przez aplikacje klienckie i obciążenie usługi powodowane przez te połączenia. Dławienie jest niezbędne, ponieważ systemy informatyczne nie są elastyczne (patrz rysunek 4.7).



Rysunek 4.7. Nieelastyczny charakter wszystkich systemów informatycznych

Oznacza to, że nie jest możliwe zwiększanie w nieskończoność obciążenia systemu w nadziei na stopniowy, proporcjonalny spadek wydajności — system informatyczny to nie guma do żucia,

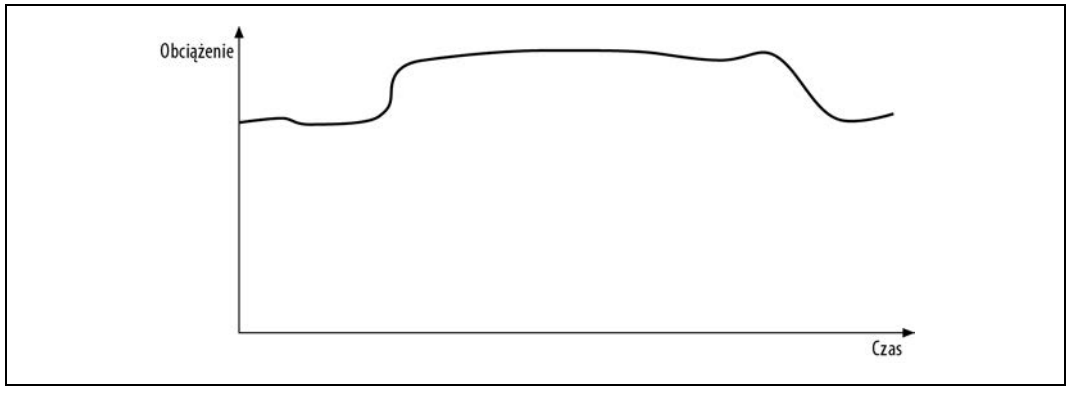
którą można rozciągać niemal bez końca. Większość systemów początkowo dobrze radzi sobie ze wzrostem obciążenia, jednak po pewnym czasie niespodziewanie odmawiają współpracy. W ten sposób zachowują się wszystkie systemy informatyczne — szczegółowa analiza przyczyn tego zachowania wykraczałaby poza zakres tematyczny tej książki (wymagałaby analizy teorii kolejkowania i kosztów związanych z zarządzaniem zasobami). To niepożądane, nieelastyczne zachowanie jest szczególnie kłopotliwe w przypadku nagłych wzrostów obciążenia (patrz rysunek 4.8).



Rysunek 4.8. Chwilowy wzrost obciążenia może spowodować, że stan systemu wyjdzie poza ograniczenia przyjęte przez projektantów

Nawet jeśli system doskonale radzi sobie z nominalnym obciążeniem (pozioma linia na rysunku 4.8), nagły wzrost obciążenia może spowodować przekroczenie założeń projektowych i doprowadzić do istotnego spadku poziomu obsługi (w stopniu odczuwalnym przez klientów). Takie czasowe skoki mogą też rodzić problemy w kontekście oceny ogólnego wzrostu obciążenia, nawet jeśli średni poziom nie powinien mieć negatywnego wpływu na funkcjonowanie systemu.

Dławienie umożliwia zapobieganie przekraczaniu limitów obowiązujących dla danej usługi i używanych przez nią zasobów. Jeśli dławienie jest włączone, przekroczenie skonfigurowanych ustawień spowoduje, że środowisko WCF automatycznie umieści oczekujące aplikacje klienckie w kolejce i obsłuży ich żądania w kolejności przesłania do usługi. Jeśli w czasie, w którym wywołanie oczekuje w kolejce, upłynie limit czasowy, klient otrzyma wyjątek `TimeoutException`. Dławienie jest przykładem niesprawiedliwej techniki, ponieważ aplikacje klienckie, których żądania znalazły się w kolejce, doświadczają istotnego spadku poziomu obsługi. W tym przypadku ta niesprawiedliwość jest jednak uzasadniona — gdyby wszystkie aplikacje wywołujące, które powodują skokowy wzrost obciążenia, zostały obsłużone, system co prawda działałby sprawiedliwie, ale spadek poziomu obsługi byłby dostrzegalny dla wszystkich klientów. Dławienie jest więc uzasadnione w sytuacji, gdy czas skoków obciążenia jest stosunkowo krótki w porównaniu z całym czasem funkcjonowania usług, tzn. gdy prawdopodobieństwo dwukrotnego umieszczenia tego samego klienta w kolejce jest bardzo niewielkie. Od czasu do czasu żądania części klientów zostaną umieszczone w kolejce (w odpowiedzi na nagły wzrost obciążenia), ale system jako całość będzie działał prawidłowo. Dławienie nie sprawdza się w sytuacjach, gdy obciążenie osiąga nowy, wysoki poziom i utrzymuje się na tym poziomie przez dłuższy czas (patrz rysunek 4.9). W takim przypadku dławienie powoduje tylko odłożenie problemu na później, prowadząc ostatecznie do wyczerpania limitów czasowych po stronie klientów. Taki system należy od podstaw zaprojektować z myślą o obsłudze większego obciążenia.



Rysunek 4.9. Nieprawidłowe zastosowanie dławienia

Dławienie stosuje się na poziomie typu usługi, zatem ma wpływ na wszystkie instancje tej usługi i wszystkie jej punkty końcowe. Mechanizm dławienia jest kojarzony z elementami rozdzielającymi dla wszystkich kanałów używanych przez daną usługę.

Technologia WCF umożliwia programiście sterowanie wybranymi lub wszystkimi poniższymi parametrami obciążenia usługi:

Maksymalna liczba równoczesnych sesji
Określa łączną liczbę klientów, którzy mogą jednocześnie dysponować sesją transportową dla danej usługi. W największym skrócie ten parametr reprezentuje maksymalną liczbę klientów jednocześnie korzystających z powiązań WS na bazie protokołu TCP i (lub) IPC (z niezawodnością, bezpieczeństwem lub oboma mechanizmami jednocześnie). Ponieważ bezpołączeniowy charakter podstawowych połączeń na bazie protokołu HTTP oznacza, że sesje transportowe są bardzo krótkie (istnieją tylko w czasie wykonywania wywołania), opisywany parametr nie wpływa na aplikacje klienckie używające podstawowych powiązań lub powiązań WS bez sesji transportowych. Obciążenie generowane przez te aplikacje klienckie można ograniczyć, określając maksymalną dopuszczalną liczbę równoczesnych wywołań. Parametr domyślnie ma wartość równą stokrotnej liczbie procesorów (lub rdzeni).

Maksymalna liczba równoczesnych wywołań
Ogranicza łączną liczbę wywołań, które mogą być jednocześnie przetwarzane przez wszystkie instancje usługi. Wartość tego parametru powinna mieścić się w przedziale od 1 do 3% maksymalnej liczby współbieżnych sesji. Parametr domyślnie ma wartość równą szesnastokrotności liczby procesorów (lub rdzeni).

Maksymalna liczba jednocześnie istniejących instancji
Decyduje o liczbie jednocześnie utrzymywanych kontekstów. Jeśli wartość tego parametru nie zostanie ustawiona przez programistę, domyślnie zostanie użyta suma maksymalnej liczby jednoczesnych wywołań i maksymalnej liczby jednoczesnych sesji (116-krotność liczby procesorów lub rdzeni). Ustawienie wartości tego parametru powoduje jej uniezależnienie od dwóch pozostałych właściwości. Sposób odwzorowywania instancji na konteksty zależy zarówno od stosowanego trybu zarządzania kontekstem instancji, jak i od obowiązującego modelu dezaktywacji kontekstu i instancji. W przypadku usługi sesyjnej maksymalna liczba instancji jest jednocześnie łączną liczbę istniejących jednocześnie aktywnych instancji i łączną liczbą współbieżnych sesji. W przypadku stosowania mechanizmu dezaktywacji instancji liczba instancji może być dużo mniejsza niż liczba kontekstów,

a mimo to aplikacje klienckie będą blokowane w razie osiągnięcia przez liczbę kontekstów maksymalnej liczby jednocześnie istniejących instancji. W przypadku usługi aktywowanej przez wywołania liczba instancji jest równa liczbie współbieżnych wywołań. Oznacza to, że dla usług aktywowanych przez wywołania maksymalna liczba instancji w praktyce jest równa mniejszej spośród dwóch innych wartości: maksymalnej liczby jednocześnie istniejących instancji i maksymalnej liczby współbieżnych wywołań. W przypadku usług singletonowych wartość tego parametru jest ignorowana, ponieważ takie usługi i tak mogą mieć tylko po jednej instancji.

Dławienie to jeden z aspektów hostingu i wdrażania. Podczas projektowania usługi nie należy przyjmować żadnych założeń dotyczących konfiguracji dławienia — programista powinien raczej zakładać, że jego usługa będzie musiała radzić sobie ze wszystkimi żądaniami klientów. Właśnie dlatego technologia WCF nie oferuje żadnych atrybutów sterujących mechanizmem dławienia, chociaż ich opracowanie nie stanowiłoby żadnego problemu.

Konfiguracja dławienia

Administratorzy zwykle konfigurują dławienie w pliku konfiguracyjnym. Takie rozwiązanie umożliwia stosowanie różnych parametrów dławienia dla tego samego kodu usługi zależnie od bieżących warunków i wdrożenia. Host może też programowo konfigurować dławienie na podstawie decyzji podejmowanych w czasie wykonywania.

Administracyjne konfigurowanie dławienia

Na listingu 4.20 pokazano, jak skonfigurować dławienie w pliku konfiguracyjnym hosta. Za pomocą znacznika `behaviorConfiguration` można dodać do usługi niestandardowe zachowanie ustawiające parametry dławienia.

Listing 4.20. Administracyjne konfigurowanie dławienia

```
<system.serviceModel>
   <services>
      <service name = "MyService" behaviorConfiguration = "ThrottledBehavior">
         ...
      </service>
   </services>
   <behaviors>
      <serviceBehaviors>
         <behavior name = "ThrottledBehavior">
            <serviceThrottling
               maxConcurrentCalls     = "500"
               maxConcurrentSessions  = "10000"
               maxConcurrentInstances = "100"
            />
         </behavior>
      </serviceBehaviors>
   </behaviors>
</system.serviceModel>
```

Programowe konfigurowanie dławienia

Proces hosta może też programowo sterować mechanizmem dławienia na podstawie bieżących parametrów wykonawczych. Dławienie można skonfigurować programowo wyłącznie przed otwarciem hosta. Mimo że host może przykryć zachowanie mechanizmu dławienia znalezione w pliku konfiguracyjnym (usuwając je i dodając własne), w większości przypadków programowe sterowanie dławieniem należy stosować tylko wtedy, gdy nie zdefiniowano odpowiednich ustawień w pliku konfiguracyjnym.

Klasa `ServiceHostBase` oferuje właściwość `Description` typu `ServiceDescription`:

```
public abstract class ServiceHostBase : ...
{
   public ServiceDescription Description
   {get;}
   // Pozostałe składowe…
}
```

Typ `ServiceDescription`, jak nietrudno się domyślić, reprezentuje opis usługi obejmujący wszystkie jej aspekty i zachowania. Klasa `ServiceDescription` zawiera właściwość nazwaną `Behaviors` (typu `KeyedByTypeCollection<T>`) z interfejsem `IServiceBehavior` w roli parametru uogólnionego.

Sposób programowego ustawiania parametrów zachowania z dławieniem pokazano na listingu 4.21.

Listing 4.21. Programowe konfigurowanie dławienia

```
ServiceHost host = new ServiceHost(typeof(MyService));

ServiceThrottlingBehavior throttle;
throttle = host.Description.Behaviors.Find<ServiceThrottlingBehavior>();
if(throttle == null)
{
   throttle = new ServiceThrottlingBehavior();
   throttle.MaxConcurrentCalls = 500;
   throttle.MaxConcurrentSessions = 10000;
   throttle.MaxConcurrentInstances = 100;
   host.Description.Behaviors.Add(throttle);
}
host.Open();
```

Kod hosta sprawdza najpierw, czy parametry zachowania dławiącego nie zostały ustawione w pliku konfiguracyjnym. W tym celu kod wywołuje metodę `Find<T>()` klasy `KeyedByTypeCollec` ↪`tion<T>`, przekazując klasę `ServiceThrottlingBehavior` w roli parametru typu:

```
public class ServiceThrottlingBehavior : IServiceBehavior
{
   public int MaxConcurrentCalls
   {get;set;}
   public int MaxConcurrentSessions
   {get;set;}
   public int MaxConcurrentInstances
   {get;set;}
   // Pozostałe składowe…
}
```

Jeśli zwrócona zmienna `throttle` ma wartość `null`, kod hosta tworzy nowy obiekt klasy `Service` ↪`ThrottlingBehavior`, ustawia jego parametry i dodaje go do zachowań reprezentowanych w opisie usługi.

Usprawnienie przy użyciu klasy ServiceHost<T>

W przypadku stosowania rozszerzeń frameworku C# 3.0 istnieje możliwość uzupełnienia klasy `ServiceHost` (lub dowolnej spośród jej podklas, na przykład `ServiceHost<T>`) o mechanizm automatyzujący kod z listingu 4.21 — przykład takiego rozwiązania pokazano na listingu 4.22.

Listing 4.22. Rozszerzenie klasy ServiceHost o obsługę dławienia

```
public static class ServiceThrottleHelper
{
   public static void SetThrottle(this ServiceHost host,
                                   int maxCalls,int maxSessions,int maxInstances)
   {
      ServiceThrottlingBehavior throttle = new ServiceThrottlingBehavior();
      throttle.MaxConcurrentCalls = maxCalls;
      throttle.MaxConcurrentSessions = maxSessions;
      throttle.MaxConcurrentInstances = maxInstances;
      host.SetThrottle(throttle);
   }

   public static void SetThrottle(this ServiceHost host,
                                   ServiceThrottlingBehavior serviceThrottle,
                                   bool overrideConfig)
   {
      if(host.State == CommunicationState.Opened)
      {
         throw new InvalidOperationException("Host jest już otwarty");
      }
      ServiceThrottlingBehavior throttle =
                  host.Description.Behaviors.Find<ServiceThrottlingBehavior>();
      if(throttle == null)
      {
         host.Description.Behaviors.Add(serviceThrottle);
         return;
      }
      if(overrideConfig == false)
      {
         return;
      }
      host.Description.Behaviors.Remove(throttle);
      host.Description.Behaviors.Add(serviceThrottle);
   }
   public static void SetThrottle(this ServiceHost host,
                                   ServiceThrottlingBehavior serviceThrottle)
   {
      host.SetThrottle(serviceThrottle,false);
   }
}
```

Klasa `ServiceThrottleHelper` udostępnia metodę `SetThrottle()`, która otrzymuje na wejściu zachowanie dławienia, i flagę typu `Boolean` określającą, czy należy przykryć ewentualne wartości odczytane z pliku konfiguracyjnego. Drugi parametr przeciążonej wersji metody `SetThrottle()` domyślnie ma wartość `false`. Metoda `SetThrottle()` używa właściwości `State` klasy bazowej `CommunicationObject` do sprawdzenia, czy host nie został jeszcze otwarty. Jeśli skonfigurowane parametry dławienia mają zostać przykryte, metoda `SetThrottle()` usuwa zachowanie dławienia z opisu usługi. Pozostały kod z listingu 4.22 bardzo przypomina rozwiązania zastosowane na listingu 4.21. Oto przykład użycia klasy `ServiceHost<T>` do programowego ustawienia parametrów dławienia:

```
ServiceHost<MyService> host = new ServiceHost<MyService>();
host.SetThrottle(12,34,56);
host.Open();
```

Także klasę `InProcFactory<T>` (wprowadzoną w rozdziale 1.) można w podobny sposób uzupełnić o kod upraszczający zarządzanie dławieniem.

Odczytywanie wartości parametrów dławienia

Programiści usług mogą odczytywać wartości parametrów dławienia w czasie wykonywania kodu (na przykład na potrzeby diagnostyczne lub analityczne). Warunkiem uzyskania dostępu do właściwości dławienia przez instancję usługi (za pośrednictwem obiektu rozdzielającego zadania) jest uzyskanie referencji do hosta z kontekstu operacji.

Klasa bazowa hosta, czyli `ServiceHostBase`, definiuje właściwość `ChannelDispatchers` dostępną tylko do odczytu:

```
public abstract class ServiceHostBase : CommunicationObject,...
{
   public ChannelDispatcherCollection ChannelDispatchers
   {get;}
   // Pozostałe składowe…
}
```

`ChannelDispatchers` to kolekcja obiektów klasy `ChannelDispatcherBase` ze ścisłą kontrolą typów:

```
public class ChannelDispatcherCollection :
                                 SynchronizedCollection<ChannelDispatcherBase>
{...}
```

Każdy element tej kolekcji jest obiektem klasy `ChannelDispatcher`. Klasa `ChannelDispatcher` udostępnia właściwość `ServiceThrottle`:

```
public class ChannelDispatcher : ChannelDispatcherBase
{
   public ServiceThrottle ServiceThrottle
   {get;set;}
   // Pozostałe składowe…
}
public sealed class ServiceThrottle
{
   public int MaxConcurrentCalls
   {get;set;}
   public int MaxConcurrentSessions
   {get;set;}
   public int MaxConcurrentInstances
   {get;set;}
}
```

Właściwość `ServiceThrottle` zawiera skonfigurowane wartości parametrów dławienia:

```
class MyService : ...
{
   public void MyMethod() // Operacja kontraktu
   {
      ChannelDispatcher dispatcher = OperationContext.Current.
                                    Host.ChannelDispatchers[0] as ChannelDispatcher;

      ServiceThrottle serviceThrottle = dispatcher.ServiceThrottle;
```

```
        Trace.WriteLine("Maksymalna liczba wywołań: " + serviceThrottle.MaxConcurrentCalls);
        Trace.WriteLine("Maksymalna liczba sesji: " + serviceThrottle.MaxConcurrentSessions);
        Trace.WriteLine("Maksymalna liczba instancji: " + serviceThrottle.MaxConcurrentInstances);
    }
}
```

Warto pamiętać, że usługa może tylko odczytać wartości parametrów dławienia i w żaden sposób nie może na nie wpływać. Każda próba ustawienia tych wartości spowoduje zgłoszenie wyjątku `InvalidOperationException`.

Także w tym przypadku proces odczytywania wartości parametrów można usprawnić za pomocą klasy `ServiceHost<T>`. Należy najpierw dodać właściwość `ServiceThrottle`:

```
public class ServiceHost<T> : ServiceHost
{
   public ServiceThrottle Throttle
   {
      get
      {
         if(State == CommunicationState.Created)
         {
            throw new InvalidOperationException("Host nie jest otwarty");
         }
         ChannelDispatcher dispatcher = OperationContext.Current.
         Host.ChannelDispatchers[0] as ChannelDispatcher;
         return dispatcher.ServiceThrottle;
      }
   }
   // Pozostałe składowe…
}
```

Od tej pory można użyć klasy `ServiceHost<T>` w roli hosta usługi, a za pośrednictwem właściwości `ServiceThrottle` można uzyskać dostęp do skonfigurowanego zachowania dławienia:

```
// Kod hosta
ServiceHost<MyService> host = new ServiceHost<MyService>();
host.Open();

class MyService : ...
{
   public void MyMethod()
   {
      ServiceHost<MyService> host = OperationContext.Current.
                                                    Host as ServiceHost<MyService>;
      ServiceThrottle serviceThrottle = host.Throttle;
      ...
   }
}
```

Dostęp do właściwości `Throttle` klasy `ServiceHost<T>` można uzyskać dopiero po otwarciu hosta, ponieważ kolekcja `dispatcher` jest inicjalizowana dopiero w momencie otwarcia.

Skorowidz

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

O'REILLY®

Programowanie usług WCF

Udostępnianie usług sieciowych to już nie tylko widzimisię programistów lub projektantów — w dzisiejszych czasach to obowiązek! Dzięki temu ułatwiasz integrację innych aplikacji z Twoim produktem, ale też z łatwością korzystasz z funkcjonalności dostarczanych przez innych producentów. Najważniejsze jest jednak to, że najwięcej zyskuje Twój klient. A jego zadowolenie zapewni Ci sukces i byt na rynku!

Jeżeli podjąłeś decyzję, że Twoja kolejna aplikacja będzie wspierała WCF, to wybierając tę książkę, nie mogłeś trafić lepiej. *Programowanie usług WCF. Wydanie III* to doskonały, cieszący się ogromną popularnością przewodnik poświęcony spójnej, jednolitej platformie firmy Microsoft, którą zaprojektowano z myślą o programowaniu aplikacji na bazie usług dla systemu Windows. Jej autor Juval Löwy jest wybitnym specjalistą w dziedzinie platformy .NET i technologii WCF. W trakcie lektury poznasz architekturę technologii WCF, jej elementy składowe oraz zagadnienia związane z jej niezawodnością. Ponadto dowiesz się, jak zagwarantować bezpieczeństwo swoim usługom sieciowym, oraz sprawdzisz możliwości magistrali usług Azure AppFabric Service Bus. Wiedza, którą zdobędziesz, pozwoli Ci na tworzenie jeszcze lepszych i bardziej elastycznych projektów informatycznych. Sprawdź sam!

- Poznaj architekturę technologii WCF i jej podstawowe elementy składowe, w tym tak ważne pojęcia jak niezawodność czy sesja transportowa.
- Naucz się używać wbudowanych elementów, takich jak hosty usług, mechanizmy zarządzania instancjami i współbieżnością, transakcje, kolejkowane wywołania rozłączonych usług, zabezpieczenia czy odkrywanie.
- Opanuj sztukę korzystania z magistrali usług Azure AppFabric Service Bus, czyli najbardziej rewolucyjnego elementu nowego projektu chmury obliczeniowej.
- Podnieś swoją produktywność i jakość tworzonych usług WCF dzięki odpowiednim opcjom projektowym, wskazówkom i zalecanym praktykom, zawartym we frameworku ServiceModelEx autorstwa Juvala Löwy'ego.
- Poznaj uzasadnienie szczegółowych decyzji projektowych i odkryj najtrudniejsze, rozumiane przez niewielu programistów aspekty programowania usług WCF.

Najlepszy podręcznik poświęcony WCF!

ISBN 978-83-246-3617-4

9 788324 636174

Cena 129,00 zł